Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

# Σχεδιασμός και Υλοποίηση ενός Φορητού Μηχανισμού Συγχρονισμού Αρχείων σε Περιβάλλον Αποθηκευτικού Νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΒΑΣΙΛΕΙΟΣ ΓΕΡΑΚΑΡΗΣ**

**Επιβλέπων :** Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Αύγουστος 2015

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

# Σχεδιασμός και Υλοποίηση ενός Φορητού Μηχανισμού Συγχρονισμού Αρχείων σε Περιβάλλον Αποθηκευτικού Νέφους

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

**ΒΑΣΙΛΕΙΟΣ ΓΕΡΑΚΑΡΗΣ**

**Επιβλέπων :** Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 28η Αυγούστου 2015.

| | | |
|---|---|---|
| ........................................... | ........................................... | ........................................... |
| Νεκτάριος Κοζύρης<br>Καθηγητής Ε.Μ.Π. | Νικόλαος Παπασπύρου<br>Αν. Καθηγητής Ε.Μ.Π. | Γεώργιος Γκούμας<br>Λέκτορας Ε.Μ.Π. |

Αθήνα, Αύγουστος 2015

...........................................

**Βασίλειος Γερακάρης**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

## Περίληψη

-Περίληψη-

**Λέξεις κλειδιά**

Αποθηκευτικό Νέφος, Συγχρονισμός αρχείων

# Abstract

Abstract

## Key words

Cloud storage, File synchronisation

# Ευχαριστίες

Η παρούσα διπλωματική εργασία σημαίνει την ολοκλήρωση ενός σημαντικού κεφαλαίου της ακαδημαϊκής μου πορείας. Θα ήθελα στο σημείο αυτό να ευχαριστήσω ορισμένους ανθρώπους που με βοήθησαν στη διαδρομή αυτή. -Ευχαριστίες-

Βασίλειος Γερακάρης,

Αθήνα, 28η Αυγούστου 2015

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά **TODO: ask nkoziris**, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Υπολογιστικών Συστημάτων, Αύγουστος 2015.

# Contents

# List of Tables

# List of Figures

Chapter 1

# Εισαγωγή

## 1.1 Κίνητρο

TODO: Κίνητρο

## 1.2 Συνεισφορά της εργασίας

TODO: Κύρια σημεία της εργασίας

## 1.3 Οργάνωση κειμένου

TODO: Οργάνωση κειμένου

## 1.4 Συνοπτική παρουσίαση του framework

TODO: Σύνοψη framework

## 1.5 Συνοπτική παρουσίαση των πειραματικών αποτελεσμάτων

TODO: Σύνοψη αποτελεσμάτων

# Chapter 1

# Introduction

- What is the problem?
- Why is it interesting and important?
- Why is it hard? (E.g., why do naive approaches fail?)
- Why hasn't it been solved before? (Or, what's wrong with previous proposed solutions? How does mine differ?)
- What are the key components of my approach and results? Also include any specific limitations.

## 1.1 Motivation

**TODO: Motivation**

## 1.2 Thesis contribution

**TODO: Thesis contribution**

## 1.3 Chapter outline

**TODO: Chapter outline**

## 1.4 Brief description of the framework

**TODO: Brief framework description**

Chapter 2

# Background

## 2.1 Data Synchronisation

Data synchronisation is the process of establishing consistency among data from a source to a target data storage and vice versa and the continuous harmonisation of the data over time. File Synchronisation (or syncing) is the process of ensuring that files in two or more locations are updated by certain rules. In *one-way file synchronisation*, also called mirroring, updated files are copied from a ‘source’ location to one or more ‘target’ locations, but no files are copied back to the source location. In *two-way file synchronisation*, updated files are copied in both directions, usually with the purpose of keeping the two locations identical to each other

## 2.2 File Hosting Service

A file hosting service[16] or cloud storage service, is an Internet hosting service specifically designed to host user files. It allows users to upload files that could then be accessed over the internet from a different computer, tablet, smart phone or other networked device, by the same user or possibly by other users, after a password or other authentication is provided. File hosting services often offer file sync and sharing services, most notable consumer products being Dropbox and Google Drive.

## 2.3 Application programming interface

Application programming interface (API) is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. APIs often come in the form of a library that includes specifications for routines, data structures, object classes, and variables. In other cases, such as REST services, an API is simply a specification of remote calls exposed to the API consumers.

### 2.3.1 Representational state transfer

Representational State Transfer (REST) is a software architecture style for building scalable web services[18] REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a more performant and maintainable architecture[19]. RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers.

## 2.4 OpenStack

OpenStack[4] is a free and open source cloud operating system that controls large pools of compute, storage, and networking resources throughout a data centre. Users can manage those resources through a web-based dashboard, command-line tools, or a RESTful API. It is primarily being deployed as an infrastructure-as-a-service (IaaS). OpenStack offers support for both Object Storage and Block Storage.Object Storage is ideal for cost effective, scale-out storage. It provides a fully distributed, API-accessible storage platform that can be integrated directly into applications or used for backup, archiving and data retention. Block Storage allows block devices to be exposed and connected to compute instances for expanded storage, better performance and integration with enterprise storage platforms.

### 2.4.1 Object Storage - Swift

OpenStack Object Storage (Swift) is a scalable redundant storage system. Objects and files are written to multiple disk drives spread throughout servers in the data centre, with the OpenStack software responsible for ensuring data replication and integrity across the cluster. Because OpenStack uses software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives and servers can be used.

## 2.5 Synnefo

Synnefo[11] is an open source cloud stack, which offers Compute, Network, Image, Volume and Storage services, similar to the ones offered by OpenStack. Synnefo is written in Python and to improve third-party compatibility, it exposes the OpenStack APIs to users[12]. It is the software used for ~okeanos[10], an Infrastructure as a Service (IaaS) cloud service, provided by the Greek Research and Technology Network (GRNET) for the Greek Research and Academic Community. ~okeanos offers a virtual compute/network service called Cyclades as well as a virtual storage service, called Pithos+.

### 2.5.1 Pithos+

Pithos+ is the Virtual Storage service of ~okeanos, featuring cloud storage as well as file synchronisation and sharing services. Files stored in Pithos+ are accessible via the web UI or with the client software, which exists for Windows, MacOS and iOS systems. Linux users can access the files using *kamaki*, the command line client for ~okeanos resources. It is powered by the Pithos (File/Object Storage) services of synnefo.

## 2.6 Amazon Web Services

Amazon Web Services (AWS) is a collection of remote computing services, also called web services, that make up a cloud-computing platform offered by Amazon.The most central and well-known of these services arguably include Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3).

### 2.6.1 Amazon S3

Amazon S3[5] (Simple Storage Service) is an online file storage web service offered by Amazon Web Services. Amazon S3 provides storage through web services interfaces (REST, SOAP, and BitTorrent). It provides developers and IT teams with secure, durable, highly-scalable object storage, for a wide variety of use cases including cloud applications, content distribution, backup and archiving, disaster recovery, and big data analytics. Amazon S3 stores arbitrary objects up to 5 terabytes in size, each accompanied by up to 2 kilobytes of metadata. Objects are organised into buckets (each owned by an AWS account), and identified within each bucket by a unique, user-assigned key.

## 2.7 Dropbox

Dropbox offers cloud storage, file synchronisation, personal cloud, and client software. Dropbox synchronises a directory so that it appears to be the same (with the same contents) regardless of which computer is used to view it. Files placed in this folder are also accessible via the Dropbox website. Dropbox is multi-platform, and is working on all major desktop and mobile OS. Originally, both the server and client software were primarily written in Python; since 2013 Dropbox has began migrating its backend infrastructure to Go. Dropbox depends on rsync, ships the librsync binary-delta library (which is written in C) and utilises delta encoding technology. When a file in a user's Dropbox folder is changed, Dropbox only uploads the pieces of the file that are changed when synchronising, when possible. It currently uses Amazon's S3 storage system to store the files. Dropbox also provides a technology called LAN sync, which allows computers on a local area network to securely download files locally from each other instead of always hitting the central servers, improving syncing speed.

## 2.8 Google Drive

Google Drive is a file storage and synchronisation service created by Google. The Google Drive client communicates with Google Drive to cause updates on one side to be propagated to the other so they both normally contain the same data. Google Drive is also multi-platform, though there is no official Linux client software. The implementation and syncing algorithm underlying Google Drive are mostly unknown, due to the software being closed source.

## 2.9 ownCloud

ownCloud[6] is a suite of client-server software for creating file hosting services and using them. ownCloud allows synchronisation of directories, similar to the way Dropbox operates. It is a free and open-source software and is multi-platform, with clients available for all major desktop and mobile OS. The server software is written in PHP and JavaScript languages. ownCloud's desktop syncing client depends and ships with csync[7], which is a lightweight utility to synchronise files between two directories on a system or between multiple systems. The software does not currently support delta-sync (syncing only file changes).

## 2.10 Hash function

A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. Hash functions accelerate table or database lookup by detecting duplicated records in a large file. Good hash functions should satisfy certain properties. Firstly, the function must be deterministic, meaning that for a given input value it must always generate the same hash value. A good hash function should map the expected inputs as evenly as possible over its output range - this property is called uniformity. This property minimises the chance of hash collisions (pairs of inputs that are mapped to the same hash value). For hash functions used in data search, it is desirable that the output of the function has fixed size, measured in bits.

### 2.10.1 SHA-256

The Secure Hash Algorithm is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS). *SHA-2* is a family of two similar hash functions, with different block sizes, known as SHA-256 and SHA-512. They differ in the word size, with SHA-256 using 32-bit words where SHA-512 uses 64-bit words and have a hash value (digest) of 256 and 512 bits, respectively. They are considered to be secure and collision resistant, with SHA-256 having a collision probability of about $4.3 * 10^{-60}$ when digesting one billion ($10^9$) different messages.

### 2.10.2 xxhash

xxHash[9] is a non-cryptographic hash function designed around speed by Yann Collet. It successfully completes the SMHasher test suite which evaluates collision, dispersion and randomness qualities of hash functions. xxHash's digests can be returned as bytes, integers or hex numbers and can be of 32 or 64 bit size.

## 2.11 ETag

The ETag or Entity Tag is a string identifier assigned to a resource, usually a file or block, that describe exactly one specific version of it. Whenever there is a change on the file, the ETag should be changed as well.ETags can be used for optimistic concurrency control[2], which is a method where shared data resources are being used without a transaction acquiring locks on them; before the transaction commits, it verifies that no other transaction has modified the data, otherwise it rolls back the changes. SHA-256 digests are commonly used as ETag identifiers, since the algorithm is secure and collision resistant, in contrast to MD5 digests, where collisions can be computed.

## 2.12 Containers

**TODO: Containers, VMs, virtualisation**

## 2.13 Database

A database-management system (DBMS) [17] is a collection of interrelated data and a set of programs to access those data. The collection of data, is referred to as the *database.*

### 2.13.1 Relational database

A relational database uses a collection of tables to represent both data and the relationships among those data. Each table represents a relation variable, has multiple columns and each column has a unique name. The columns define the attributes and each row is an instance of the variable. The rows are uniquely identified by a certain attribute, called the *primary key.*

### 2.13.2 Transactional database

A transactional database is one in which all changes and queries have the **ACID** [1] (Atomicity, Consistency, Isolation, Durability) set of properties. Those properties guarantee that database transactions are processed reliably even in the event of a transaction interruption.

**Atomicity**

The atomicity property ensures that in a transaction, a series of database operations either all occur, or nothing occurs. An atomic system must guarantee atomicity in every situation, including power failures, errors, and crashes. To the outside world, a committed transaction appears (by its effects on the database) to be indivisible (”atomic”), and an aborted transaction does not happen at all.

**Consistency**

The consistency property ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This does not guarantee correctness of the transaction in all ways the application programmer might have wanted (that is the responsibility of application-level code) but merely that any programming errors cannot result in the violation of any defined rules.

**Isolation**

The isolation property ensures that the concurrent execution of transactions results in a system state that would be obtained if transactions were executed serially. Providing isolation is the main goal of concurrency control. Depending on concurrency control method, the effects of an incomplete transaction might not even be visible to another transaction.

**Durability**

Durability means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors. In a relational database, once a group of SQL statements execute, the results need to be stored permanently (even if the database crashes immediately thereafter). To defend against power loss, transactions (or their effects) must be recorded in a non-volatile memory.

### 2.13.3 Structured Query Language

Structured Query Language (SQL) is a special-purpose programming language designed for managing data held in a relational database management system (RDBMS). The most important elements of the SQL language are the *Statements*, which may have a persistent effect on schemata and data, or may control transactions, program flow, connections, sessions, or diagnostics and the *Queries*, which retrieve data from the database, based on specific criteria.

### 2.13.4 SQLite

SQLite[3] is an open source, cross-platform RDBMS contained in a C programming library that offers a full SQL implementation. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program and reads and writes directly to ordinary disk files. SQLite is transactional and as such, ACID-compliant. SQLite has a small code footprint and is widely used on memory and disk space constrained cases.

Chapter 3

# Design & Implementation

## 3.1 Syncing Algorithm

### 3.1.1 Known algorithms

The process of synchronising two filesystem trees (henceforth called “A” and “B”) can be complex to perform correctly, if some needed information is missing. Differences in files can be detected by comparing their hash digests, which is often used as an ETag. While this is the most secure and reliable way to detect changes, computing the hash digest of a file, especially when using a cryptographically secure function) is a computationally expensive procedure that takes significant time for large files. The last modification time is a fast and cheap way to detect file changes, but since it is a property that can be manipulated by software, improper or malicious manipulation can result in failure to detect changes.

First of all, history data for those two trees are very important, as illustrated in the following example. There are generally three cases when synchronising two trees:

1. File exists on both and is identical

2. File exists on both and is different

3. File exists on A but not on B (or vice-versa)

Case 1 is easily handled, since there is no action to be taken. Without history information, the other two cases would require user input in order to be handled correctly. In case 2, the files should be merged, but cannot be done automatically - and asking regular users how to merge files is undesirable. In case 3, the file might be a newly created, or a recently deleted one and should be copied to tree B or deleted from A, respectively. While the safe choice is to assume the file is new and copy it to tree B, it is often **not** the right thing to do. It is therefore clear that without file history data the syncing algorithm makes wrong assumtions and fails to handle correctly the most common cases.

With file history present in the form of metadata, a more proper synchronisation is achievable. By checking what changes occured and when between the times $T_1$ and $T_2$, it is possible to determine the action that should be taken, based on table 3.1:

| **File A** | **File B** | **Action** |
|---|---|---|
| No Change | No Change | No Action |
| Created (ETag = J) | Created (Etag = J) | No Action |
| Created (ETag = J) | Created (Etag = K) | *Merge** |
| Deleted | Deleted | No Action |
| Deleted | No Change | Delete B |
| Modified | No Change | Update B |
| Modified (ETag = J) | Modified (ETag = K) | *Merge** |

**Table 3.1:** Syncing actions based on file states

[*] In this table, *Merge* refers to a situation where files A and B become identical. One way this can be achieved is by generating diffs and patching the files, requiring user input if a conflict arises - this is the way most Version Control Systems (VCS), like git and Mercurial, work. A different way, that requires no immediate user interaction is to accept one file version (e.g. File A) and propagating its changes to all other trees, while also renaming the conflicting files, so the conflicts can be manually merged later.

For any given time, detecting what happened between the times $T_1$ and $T_2$, is straightforward, and described in 3.2:

| **Time $T_1$** | **Time $T_2$** | **Change** |
|---|---|---|
| Does not Exist | Exists | Created |
| Exists | Does not Exist | Deleted |
| Exists (ETag = J) | Exists (Etag = J) | No Change |
| Exists (Etag = J) | Exists (Etag = K) | Modified |

**Table 3.2:** File change detection between two points in time

While this algorithm is significantly better than the previous one, it still has limitations, the most important of which is the failure to detect renames. This can become possible by comparing file digests, but doing so for all files in a directory or for very large files is computationally expensive and slows down the sync process. An even harder problem is the detection of a file that has been renamed and modified, hence having a different hash digest than the original one, a case most syncing algorithms fail to handle efficiently.

### 3.1.2 Proposed Algorithm

At this point we propose a new sychronisation algorithm, one which is efficient, fast and reliable. We assume a service that uses a central metadata server, which maintains information about each version of an uploaded object. We also assume the usage of a local state database, henceforth called **StateDB**, which locally stores the metadata of all files in the local directory, as they were during the last synchronisation with the server. More specifically, a path hash is used as a file identifier, and other important metadata saved are the file name ("path"), the inode, last modification time ("modtime"), and the file's hash digest("Etag"). Now the problem of reconcilation between the local directory replicas ("Local") and the remote server replicas ("Remote") can be now handled in three steps.

#### Step 1: Detect updates from Local Directory

For each file in the local directory, the necessary action can be derived from the decision tree in Figure 3.1.

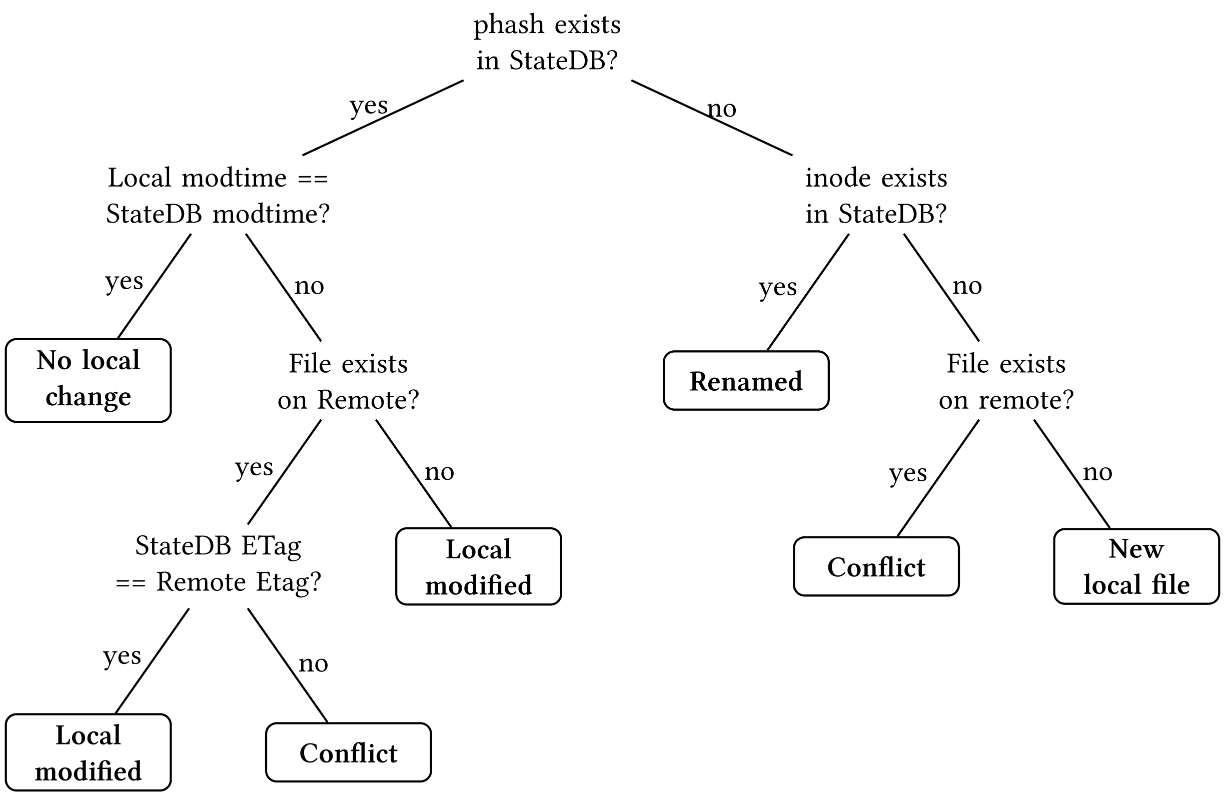


**Figure 3.1**: Decision tree for local directory files

**Step 2: Detect updates from StateDB**

For each entries in the StateDB, the necessary action can be derived from the decision tree in Figure 3.2.

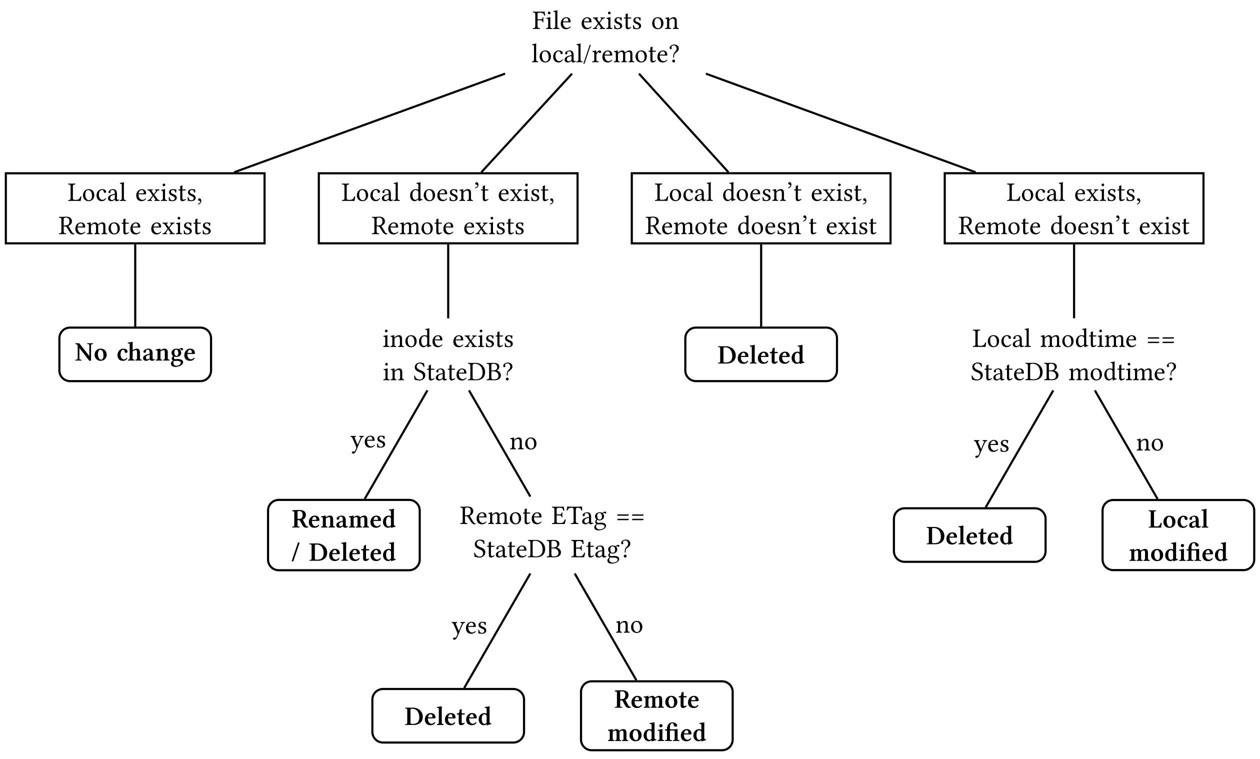

File exists on local/remote?
Local exists, Remote exists
Local doesn't exist, Remote exists
Local doesn't exist, Remote doesn't exist
Local exists, Remote doesn't exist
No change
inode exists in StateDB?
Deleted
Local modtime == StateDB modtime?
yes
no
yes
no
Renamed / Deleted
Remote ETag == StateDB Etag?
Deleted
Local modified
yes
no
Deleted
Remote modified

**Figure 3.2**: Decision tree for StateDB entries

### Step 3: Detect updates from Remote Server

In accordance to the previous steps, the necessary action for each file in the remote server can be derived from the decision tree in Figure 3.3

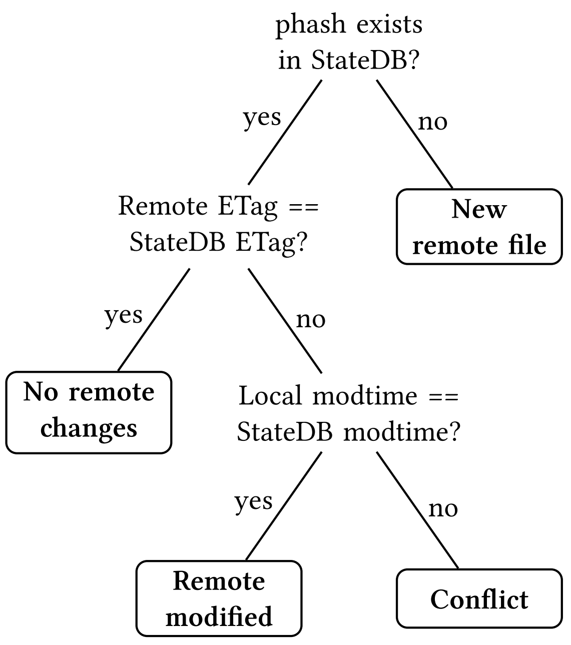


**Figure 3.3:** Decision tree for remote server files

### Notes on the syncing algorithm

- This algorithm offers a resilient synchronisation process. Since it completely processes each file, there is no problem if a partial sync is made because of an interruption.
- For update detection between Local and StateDB, we use modtime, since it is faster. For update detection between StateDB and Remote, we use the ETag, since it is the most reliable way and is readily available without hashing the file - it is stored in both cases and available with only a lookup.
- When the file is updated on Local but deleted on Remote, we decide to upload the modified file again, since it is the safe option.
- If conflict is detected, that means that the both the Local file and the Remote one have been updated since the last sync was completed. In that case we propose to rename the local one to "<filename>-conflicting_copy.<extension>" and download the remote one. Filenames ending in "-conflicting_copy" are excluded from Step 1 of the syncing algorithm.
- In step 2, figure 3.2, "**No change**" means that there is no action to be taken at this point. There might be changes in the files, but they will be handled by steps 1 and 3.
- In step 2, figure 3.2, "**Renamed / Deleted**" means that we are unsure whether the file has been renamed or deleted. If a FileID info (a string, created once at the creation time of the file that changes over the file's lifetime) is supported and available from the remote server (as ownCloud does), it should be easy to decide which of the two possible actions should be taken. Without this information, we suggest an alternative way to handle this case, without performing a costly reverse-inode lookup. When a file falls in this category during step 2, we add its inode to a *delete_set*. During step 1, for each file in the local directory, we discard its

inode from the *delete_set*, if it exists. When the algorithm reaches step 2 again, all remaining files in the *delete_set* should be deleted.

## 3.2 Basic Classes / API

### 3.2.1 FileStat

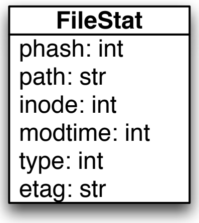


**Figure 3.4**: FileStat UML class description

FileStat is the core class used in this framework to represent a file object's status. The StateDB stores entries with this format and the other classes and methods use instances of this class to refer to files.

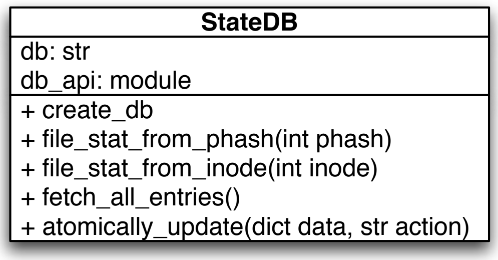
- **phash (int)**: The hash digest of the relative path string. Currently uses the xxh64 hash function, for the reasons explained below. It is the main identifier of a file.
- **path (str)**: The path of the file, relative to the root synchronisation directory.
- **inode (int)**: The index node of the file on the file system. Used mostly for rename detection.
- **modtime (int)**: The last modification time of a file. It is stored in the POSIX format (UNIX epoch), for accurate and timezone-independent representation.
- **type (int)**: The type of the file. Added for possible future features, depending on the file type (document, image, binary object, etc).
  Currently only the following two values are used: {0 = Regular file, 1 = Directory}.
- **etag (str)**: The ETag of the file. Current implementation uses the SHA-256 digest of the file. Hashing is not done locally, but uses the hashed value produced and stored on the server.

#### Path hash algorithm selection

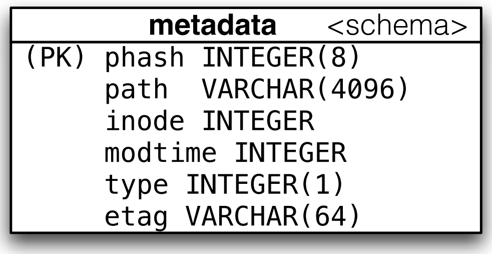
We needed the path hash (phash) algorithm to be consistent and provide hash values that are both collision resistant and large enough (in bits) as to provide a large enough set of permissible outputs and prevent collisions occuring from the birthday problem. Generating the hash of a path occurs often, so a fast hash function is preferable to a cryptographic but slower one. The most notable hash functions that fit this description were MurmurHash 3 and xxHash[8], and we decided to use the latter, because it was faster and a better Python library was available to use in the framework.

### 3.2.2 StateDB

StateDB is the class that manages the state database, which stores the metadata of the files during the time each was last synchronised. The current implementation uses the SQLite library with the sqlite3 module, because it is reliable, fast and lightweight, properties that are important for the framework. SQLite also offers a *row_factory* attribute, enabling us to return the results as FileStat objects, instead of rows (tuples). Nevertheless, the class has been designed to be easily used


StateDB
db: str
db_api: module
+ create_db
+ file_stat_from_phash(int phash)
+ file_stat_from_inode(int inode)
+ fetch_all_entries()
+ atomically_update(dict data, str action)

**(a)** StateDB UML class description



**(b)** StateDB schema

**Figure 3.5:** StateDB

with a different database engine, and just needs any python module that conforms to the Python Database API Specification v2.0 (PEP-249)[13]. The schema used for the database, as seen in figure 3.5b, has the same attributes as the FileStat object. A brief explanation of StateDB attributes and methods follows:

- **db**: The full path of the database file (sqlite3 uses files to store the database)
- **db_api**: Any PEP 249-compliant DB API module.

- **create_db**: Creates the metadata database, if it doesn't already exist, using the schema in figure 3.5b.
- **file_stat_from_phash**: Returns a FileStat object for the given *phash* if it exists, else returns *None*.
- **file_stat_from_inode**: Returns a FileStat object for the given *inode* if it exists, else returns *None*.
- **fetch_all_entries**: Returns all entries in the StateDB as FileStat objects. Uses a generator that fetches 1000 entries at a time, in order to reduce memory footprint in cases of large databases.
- **atomically_update**: Atomically updates the StateDB, using the *data* and *action* specified in the arguments. For sqlite, we use a 2-phase commit mechanism, copying the database to a temporary location, updating the copy and then moving the updated copy back to the original's location, in order to emulate an atomic commit.

### 3.2.3 LocalDirectory

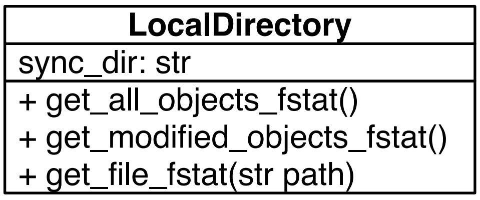

LocalDirectory
sync_dir: str
+ get_all_objects_fstat()
+ get_modified_objects_fstat()
+ get_file_fstat(str path)

**Figure 3.6:** LocalDirectory UML class description

LocalDirectory is the base Class that represents the local sync directory. A brief explanation of LocalDirectory attributes and methods follows:

- **sync_dir**: The full path of the root directory to be synchronised.

- **get_all_objects_fstat**: Returns all local files' metadata as FileStat objects. Uses a generator to reduce memory footprint.

- **get_modified_objects_fstat**: Return file metadata only for the files that were modified since the last sync. On the base implementation it returns all files in the directory, as if **get_all_objects_fstat()** was called.

- **get_file_fstat**: Returns the FileStat object for the file *path* if it exists, else returns *None*.

### 3.2.4 CloudClient

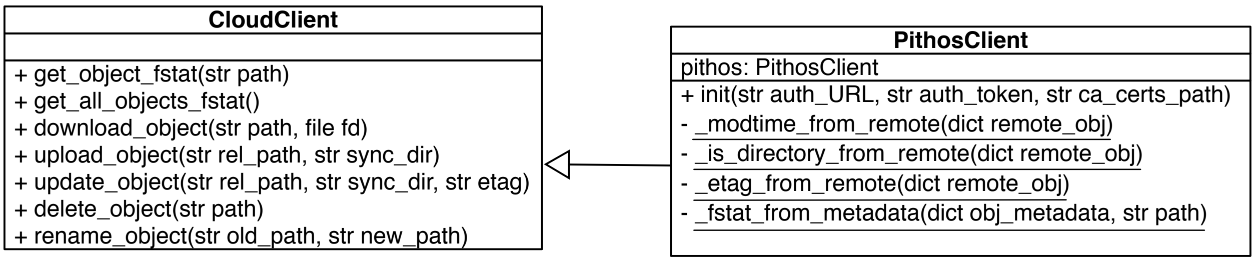

CloudClient
+ get_object_fstat(str path)
+ get_all_objects_fstat()
+ download_object(str path, file fd)
+ upload_object(str rel_path, str sync_dir)
+ update_object(str rel_path, str sync_dir, str etag)
+ delete_object(str path)
+ rename_object(str old_path, str new_path)
PithosClient
pithos: PithosClient
+ init(str auth_URL, str auth_token, str ca_certs_path)
- _modtime_from_remote(dict remote_obj)
- _is_directory_from_remote(dict remote_obj)
- _etag_from_remote(dict remote_obj)
- _fstat_from_metadata(dict obj_metadata, str path)

**Figure 3.7:** CloudClient - PithosClient UML class description

CloudClient is the base class for representing an Object Storage service client. None of the methods shown in 3.7 are implemented on the base class and just serve as an API designation. Derived classes should implement those methods and extend the class with their own, where necessary. In this dissertation, we show an example subclass with Pithos (File/Object Storage) services of synnefo, as used in the ~okeanos IaaS. A brief explanation of CloudClient and PithosClient attributes and methods follows:

#### CloudClient

- **get_object_fstat**: Returns the metadata of the file *path* stored on the remote server as a FileStat object.

- **get_all_objects_fstat**: Returns the metadata of all files stored on the remote server as FileStat objects.

- **download_object**: Downloads the file *path*, saving its contents to the file desciptor *fd*.

- **upload_object**: Uploads the local file from *rel_path* to the remote server. Fails if the file already exists on the server.

- **update_object**: Updates the the remote server replica with the changes in local file *rel_path*. Checks for race condition, updating only if the *etag* matches the one stored in StateDB, else fails.

- **delete_object**: Deletes the remote server file *path*.

- **rename_object**: Renames the remote server file *old_path* to *new_path*. Fails if *new_path* exists on the server.

**PithosClient**

- **pithos**: An authenticated client which calls the Pithos API functions of the service, when needed.

- **init**: Uses *auth_URL* and *auth_token* to authenticate a pithos client with Astakos (the Identity Management Service), then prepares it for use.
- **_modtime_from_remote**: Static method that returns the modtime in POSIX format (UNIX epoch timestamp), given the remote file metadata response *remote_obj*. Used for disambiguation, since the json responses of pithos service follow two different formats.
- **_is_directory_from_remote**: Static method that returns *True* if the object is a folder, given the remote object metadata response *remote_obj*. Used for disambiguation.
- **_etag_from_remote**: Static method that returns the etag, given the remote file metadata response *remote_obj*. Used for disambiguation.
- **_fstat_from_metadata**: Returns the FileStat object of a file, given its metadata response *remote_obj*.

### 3.2.5 Syncer

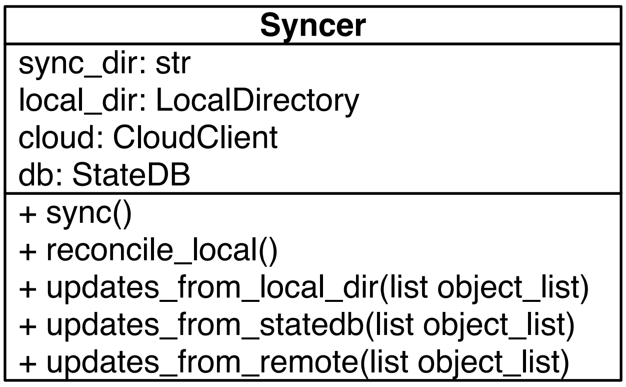

Syncer
sync_dir: str
local_dir: LocalDirectory
cloud: CloudClient
db: StateDB
+ sync()
+ reconcile_local()
+ updates_from_local_dir(list object_list)
+ updates_from_statedb(list object_list)
+ updates_from_remote(list object_list)

**Figure 3.8:** Syncer UML class description

Syncer is the class that manages the synchronisation process between a local directory and a remote object storage service. A brief explanation of Syncer attributes and methods follows:

- **sync_dir**: The full path of the root directory to be synchronised.
- **local_dir**: The LocalDirectory object to be used during sync.
- **cloud**: The authenticated CloudClient object to be used during sync.
- **db**: The StateDB object to be used during sync.

- **sync**: Executes a full local-remote sync, using the 3-step algorithm described in section 3.1.2.
- **reconcile_local**: Checks all files in the local directory for updates and performs the necessary actions for their synchronisation, as described in figure 3.1.

- **updates_from_local_dir**: Checks all files in the *object_list* for updates and performs the necessary actions for their synchronisation, as described in figure 3.1.
- **updates_from_statedb**: Checks all files in the *object_list* for updates and performs the necessary actions for their synchronisation, as described in figure 3.2. *object_list* is generated by fetching all entries in the StateDB.
- **updates_from_remote**: Checks all files in the *object_list* for updates and performs the necessary actions for their synchronisation, as described in figure 3.3. *object_list* is generated by fetching all remote files' metadata.

Chapter 4

# Syncer Optimisations

In this chapter, we improve on the initial design of the framework described in chapter 3. We run benchmarks on the optimisations proposed, and present the results. The specifications of the two setups used for testing are shown in table 4.1.

| MacBook Pro 2011 | |
|---|---|
| **Operating System** | OS X 10.10.4 (Yosemite) |
| **Processor** | 2.3 GHz Intel Core i5 |
| **Memory** | 8 GB 1333 MHz DDR3 |
| **Storage** | 256GB SSD Crucial m4 |
| **Network Speed** | 24.4/2.5 Mbit/s |

**(a)** MBP

| ~okeanos Virtual Machine | |
|---|---|
| **Operating System** | Ubuntu Linux 14.04.2 LTS (Trusty) |
| **Processor** | 2.1 GHz Virtual CPU QEMU v2.1.2 |
| **Memory** | 4 GB Virtual RAM |
| **Storage** | 80 GB (DRBD) |
| **Network Speed** | 344.6/137.3 Mbit/s |

**(b)** VM

**Table 4.1:** Benchmark setups specs

## 4.1 Request queuing

After using the framework with the help of a profiler, it became apparent that a bottleneck existed whenever there was a need for multiple requests on the remote server, due to the latency and the round-trip times. Additionally, during the transfer (download or upload) of a large file, the synchronisation process was being unnecessarily stalled until the tranfer finished. To overcome these issues, a request queuing system was implemented, dispatching the requests to different threads, while the main thread continued the execution of the sync. There was also a dramatic speedup when step 2 of the syncing algorithm was modified to request all objects' metadata from the remote server together, instead of individually for each file. To ensure correctness in the synchronisation process, the program should wait until all threads executing requests for a step of an algorithm (as described in section 3.1.2) have finished, before starting a different step. A more efficient solution would be to implement a locking mechanism and disallow actions on files already being processed by a spawned thread, but this would require substantial changes in the framework code.

### 4.1.1 Benchmarks

#### Upload time relative to number of threads

For the first benchmark, we tried to upload 100 files of 150 bytes each sequentially (denoted as "0" threads in table 4.2) and again by using a different number of threads. The files were always being randomly generated, because Pithos+ stores the blocks that are uploaded, and we needed to evade that caching for accurate measurements. Each batch of uploads was executed 10 times, and the results presented are the average of those tries. The results are statistically accurate, having a standard deviation of $\sigma_A = 1.1$.

| | # of threads | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
| **time (s)** | 92.55 | 91.51 | 48.33 | 33.42 | 29.79 | 29.80 | 30.85 | 30.79 | 30.95 | 30.68 | 31.23 |
| **speedup (%)** | N/A | 1.51 | 47.78 | 63.89 | 67.81 | 67.80 | 66.67 | 66.73 | 66.56 | 66.85 | 66.25 |

**(a)** MBP

| | # of threads | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 |
| **time (s)** | 76.24 | 72.79 | 43.92 | 33.82 | 29.90 | 33.52 | 34.85 | 33.18 | 33.01 | 33.98 | 32.26 |
| **speedup (%)** | N/A | -0.21 | 39.54 | 53.44 | 58.84 | 53.85 | 52.03 | 54.31 | 54.55 | 53.21 | 55.58 |

**(b)** VM

**Table 4.2**: Upload speedup by queuing, relative to # of threads

As seen in table 4.2, there is a considerable speedup when using multiple threads to upload the files, with the uploads completing in less than half the time on the VM, when 4 or more threads were used. It is worth mentioning that the pithos service used raised exceptions on some requests when using 8 or more threads. Since reliability is a core element of a synchronisation framework, the use of 4 or less threads is recommended when using this feature.

| | File Size | | |
|---|---|---|---|
| | 150 B | 150 KB | 1.5 MB |
| **Sequential upload time (s)** | 92.55 | 153.32 | 636.48 |
| **4 threads upload time (s)** | 33.82 | 68.12 | 569.43 |
| **speedup (%)** | 63.46 | 55.57 | 10.54 |

**(a)** MBP

| | File Size | | |
|---|---|---|---|
| | 150 B | 150 KB | 1.5 MB |
| **Sequential upload time (s)** | 76.24 | 86.71 | 106.54 |
| **4 threads upload time (s)** | 33.82 | 38.15 | 38.83 |
| **speedup (%)** | 55.64 | 56.01 | 63.55 |

**(b)** VM

**Table 4.3**: Upload speedup by using queue with 4 threads, relative to file size

From table 4.3 we observe that the percentage of speed improvement gained using threads is dependent on the network bandwidth. As the sequential upload of files get closer to the max throughput of the network, the speed improvement of request queuing becomes less significant. It still remains a considerable improvement when synchronising smaller files or using networks with large bandwidth, but the performance gain while transfering VM images or snapshots is relatively small.

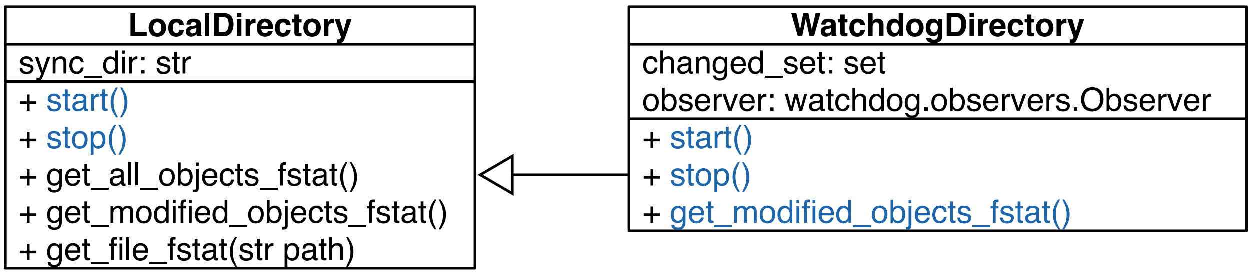
## 4.2 Directory monitoring

Detecting file changes on the original algorithm meant that each file in the directory would have to be individually checked for updates, a process that scales linearly with the number of files present. Even at a speed of about 1000 files scanned per second on an SSD (MBP), each sync would need over 100 seconds for the local directory only, which is highly undesirable. The solution to this problem is to use the filesystem monitoring mechanisms that exist for the Operating Systems (OS), to quickly produce a substantially smaller set of files that have potentially changed. Such utilities and functions exist for the most common OS and are:

- inotify (Linux 2.6 or later)
- FSEvents (Mac OS X)
- kqueue (FreeBSD, BSD, OS X)
- ReadDirectoryChangesW function (Windows)

To implement this in the framework, the *watchdog* Python library was used, which provides easy access to the aforementioned utilities. Two functions, *start()* and *stop()* were added to the

LocalDirectory class, and then inherited to the WatchdogDirectory subclass. Those two functions are a no-op in the original implementation. To avoid the risk of an incorrect synchronisation if some files were modified while the monitoring was not active, the framework always perform a full local scan on startup, using the *get_all_objects_fstat()* function, and uses the more efficient *get_modified_objects_fstat()* for all subsequent scans, while the application remains active.



LocalDirectory
sync_dir: str
+ start()
+ stop()
+ get_all_objects_fstat()
+ get_modified_objects_fstat()
+ get_file_fstat(str path)
WatchdogDirectory
changed_set: set
observer: watchdog.observers.Observer
+ start()
+ stop()
+ get_modified_objects_fstat()

**Figure 4.1**: WatchdogDirectory

- **changed_set**: The set containing all files that were created, modified or renamed, since the last time *get_modified_objects_fstat()* was called.
- **observer**: The thread that monitors the sync directory for changes.

- **start**: Starts the observer thread.
- **stop**: Stops the observer thread.
- **get_modified_objects_fstat**: Returns the metadata of the files in the *changed_set* as FileStat objects. Clears the *changed_set*.

### 4.2.1 Benchmarks

For this benchmark, we created a directory containing 1.000.000 files (1000 directories of 1000 files each). A LocalDirectory and a WatchdogDirectory instance were created and started and then a number of files were modified. Immidiately afterwards, a list of the modified files was requested and we the response time was timed. Each test was executed 5 times, and the average of the results are shown on the tables 4.4, 4.5 and graphed on figures 4.2, 4.3.

| | **# files modified** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **0** | **10** | **100** | **1000** | **10000** | **100000** | **1000000** | **default** |
| **time (s)** | 1.06E-5 | 0.004 | 0.038 | 0.339 | 1.618 | 12.907 | 90.003 | 108.110 |
| **speedup (%)** | 100.000 | 99.996 | 99.965 | 99.687 | 98.503 | 92.825 | 16.749 | N/A |

**Table 4.4**: MBP local directory *get_modified_objects_fstat()* times, relative to # of files modified

| | **# files modified** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **0** | **10** | **100** | **1000** | **10000** | **100000** | **1000000** | **default** |
| **time (s)** | 9.00E-5 | 5.85E-4 | 0.004 | 0.039 | 0.383 | 4.200 | 41.230 | 48.600 |
| **speedup (%)** | 100.000 | 99.999 | 99.992 | 99.919 | 99.212 | 91.358 | 15.164 | N/A |

**Table 4.5**: VM local directory *get_modified_objects_fstat()* times, relative to # of files modified

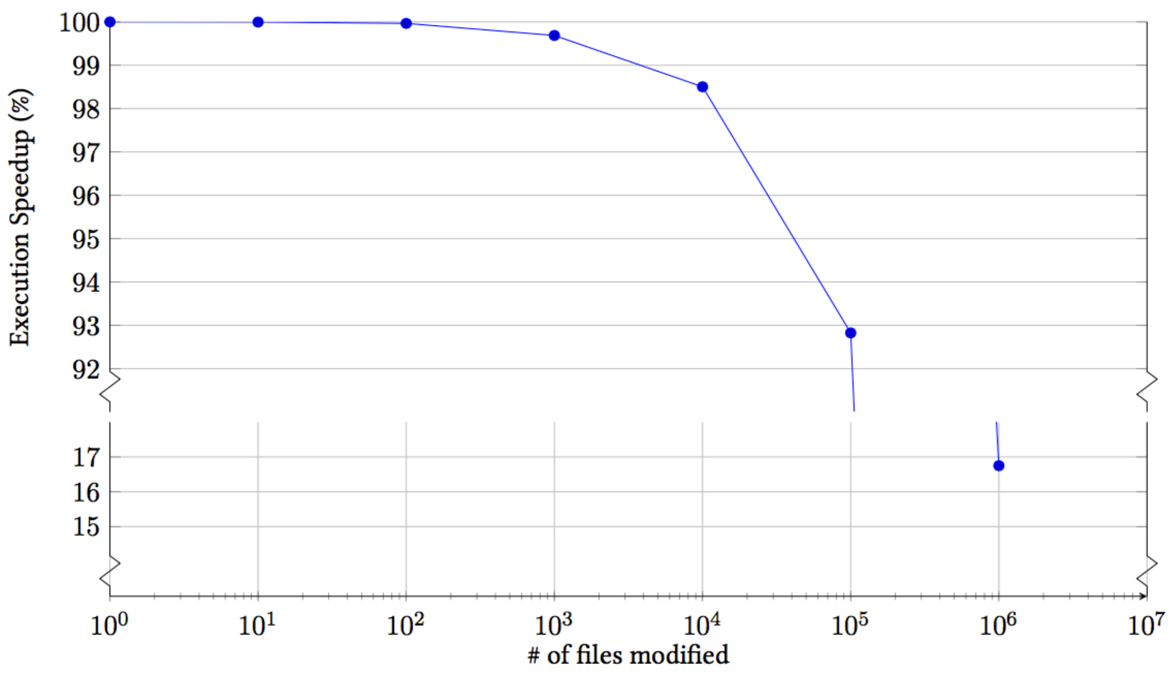


**Figure 4.2**: MPB Speedup relative to number of modified files

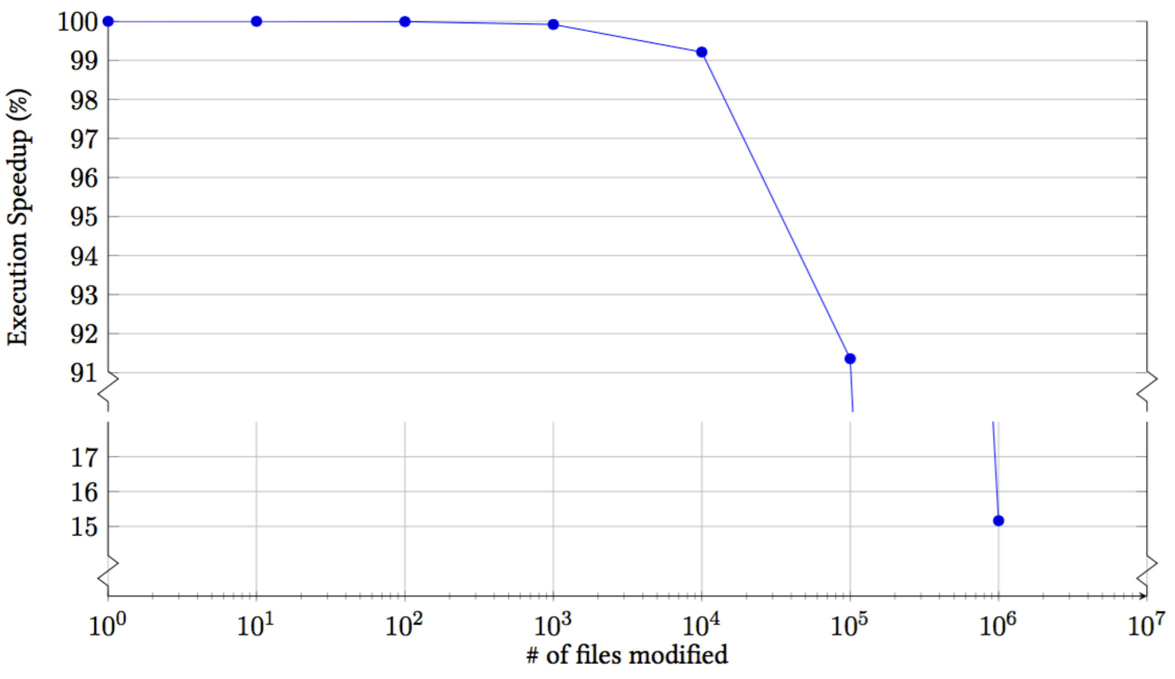


**Figure 4.3**: VM Speedup relative to number of modified files

From the results displayed in tables 4.4, 4.5, it is obvious that using a directory monitoring mechanism can result in very significant speed gains. This is more apparent in directories with a large number of files; by using this optimisation the time to detect file changes scales linearly with the number of modified files, rather than the total number of files present in the directory.

## 4.3 Local block storage

As mentioned before, the motivation behind the creation of this framework was the synchronisation of Virtual Machine images and snapshots. All those files are relatively large in size, usually several GB, and they have many similarities in their data; they are often referred to as *large similar files*. Those similarities can be exploited from local clients in order to improve download speeds, if such a file is already present on the system.

In this framework, we propose and implement a way to benefit from that characteristic of large similar files. We use a directory on the local filesystem to save the blocks of all the files present in the sync directory. To further improve speed, we take advantage of the caching capabilities of the OS, by creating a structure of 65536 directories, 256 folders containing 256 folders each, named using a hex number from "00" to "ff". A block is stored at the directory indicated by the first four characters of its hash value. For example, a block with hash value **abcdef1234567890** would be saved to <**block_directory**>**/ab/cd/ef1234567890**.

Whenever a file should be downloaded from the remote server, the client first asks for a list containing the hash values of its blocks, and downloads only the ones that do not already exist in the block directory. The file is constructed afterwards, using the stored blocks. To ensure that the block directory contains all the new blocks when a file modification occurs on local, the blocks are stored immediately after a successful object upload or update request at the server. The changes on the CloudClient class are shown in figure 4.4, hilighting the methods that should be modified on derived classes to implement this feature.

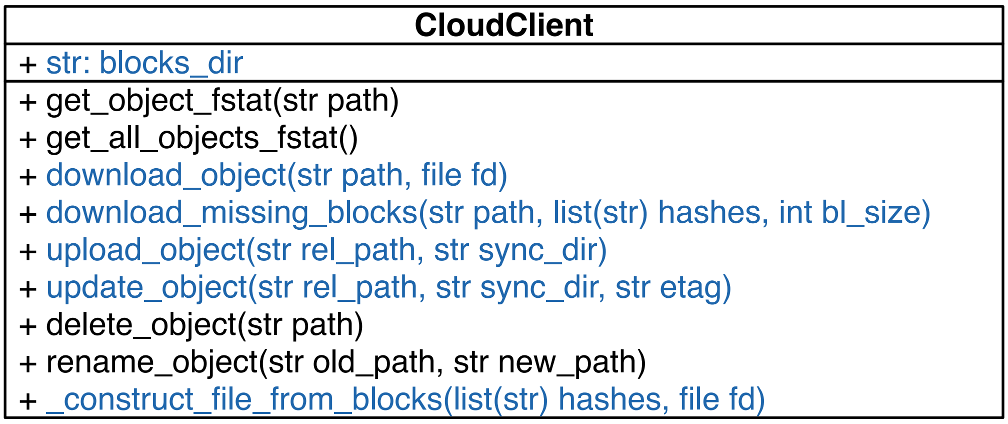

CloudClient
+ str: blocks_dir
+ get_object_fstat(str path)
+ get_all_objects_fstat()
+ download_object(str path, file fd)
+ download_missing_blocks(str path, list(str) hashes, int bl_size)
+ upload_object(str rel_path, str sync_dir)
+ update_object(str rel_path, str sync_dir, str etag)
+ delete_object(str path)
+ rename_object(str old_path, str new_path)
+ _construct_file_from_blocks(list(str) hashes, file fd)

**Figure 4.4:** CloudClient with local block storage feature

- **blocks_dir**: The full path of the directory where file blocks are stored. Optional argument, whose existence denotes the use of the feature.

- **download_object**: Checks the hash list of the remote object and downloads only the blocks that are missing, constructing the file from its hashmap afterwards.
- **download_missing_blocks**: Downloads the blocks in 'hashes' list that do not already exist on the block directory.
- **upload_object**: Uploads a file to the remote server and stores its blocks to the block directory.
- **update_object**: Updates a file at the remote server and stores its blocks to the block directory.
- **_construct_file_from_blocks**: Static method that constructs and saves a file to the file descriptor 'fd', given its hash list 'hashes'.

### 4.3.1 Benchmarks

To test the performance of this optimisation, we created a text file of 40 MiB (41,943,040 Bytes) in size which is exactly equal to 10 blocks of 4MiB (4194304 Bytes) each, and uploaded it on the remote server. We then modified some blocks, uploaded the change on the remote, and measured the time needed to download the file. Each benchmark ran five times, the average of which is presented on the results, shown in the table 4.6 and graphed on figure 4.5.

| | **# of modified blocks** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
| **time (s)** | 0.37 | 2.59 | 4.49 | 6.44 | 8.98 | 10.12 | 12.23 | 13.60 | 15.65 | 17.59 | 19.61 |
| **speedup (%)** | 98.1 | 86.8 | 77.1 | 67.2 | 54.2 | 48.4 | 37.7 | 30.7 | 20.2 | 10.3 | N/A |

**Table 4.6:** MBP file download times, relative to # of modified blocks

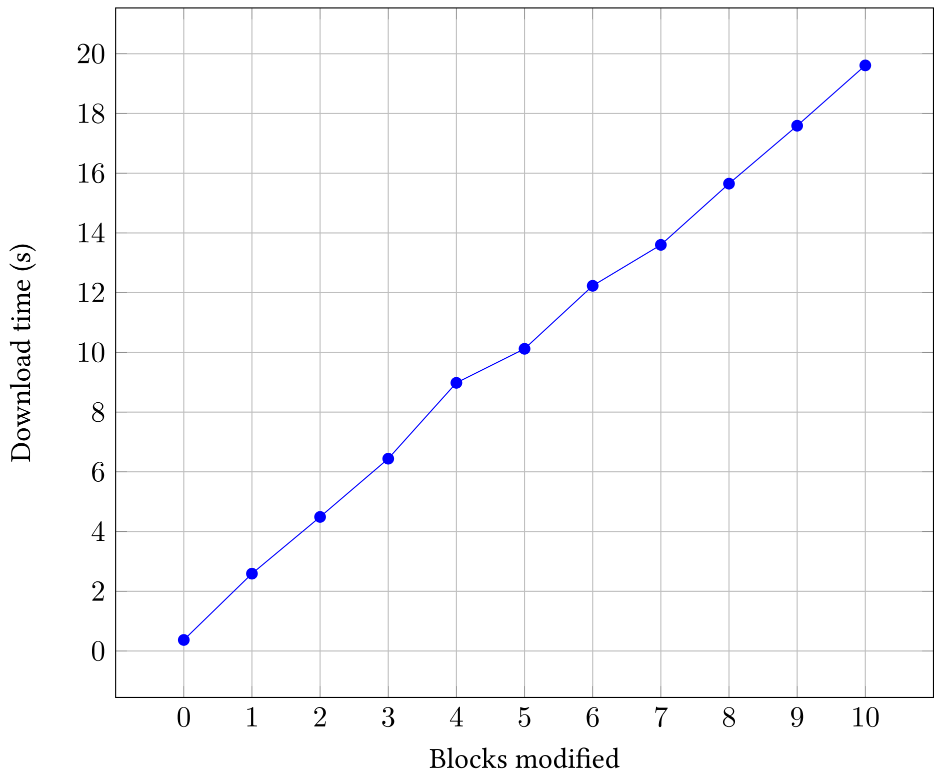


**Figure 4.5:** MBP file download times, relative to # of modified blocks

We can see from figure 4.5 that the results of the benchmarks very closely approximate a linear correlation. Therefore, the results confirm the hypothesis that the the time needed to download a file is proportional to the number of the blocks that do not exist in the local block directory (and hence, have to be downloaded). This is very important for the main use case that this framework was created for, since it allows considerably faster downloads of large similar files. The performance gain achieved on a file download by this optimisation is approximately:

$$\text{Performance gain \%} = \left(1 - \frac{\text{\# of new blocks}}{\text{Total \# of blocks}}\right) \times 100$$

## 4.4 Local deduplication - FUSE

As mentioned on section 4.3, we can implement local block storage, to improve download speeds for large similar files. Even using this method, storing many VM snapshots or images requires large amounts of disk space, since the files are being constructed from their blocks, taking space in the file system. Since those files consist of a significant amount of same blocks, very efficient data compression can be achieved by using deduplication techniques on the local file system.

The solution proposed is to only store the blocks in the block directory and not reconstruct the files in the syncing directory, but link a file's hash list to the corresponding blocks in the block directory instead. This solution requires modifications to most existing file systems, so either a kernel module or a Filesystem in Userspace (FUSE) mechanism is required, of which we propose the latter. This mechanism should modify the calls to *fstat(), open(), read()* and *write()* system calls, to use the blocks a file is consisted of. The design should follow the "*Write once, Read many, Update never*" practice, which suggests never modifying a block in the block directory, but instead use the Copy on Write (CoW) strategy, to create new blocks when changes are required. This ensures that the original resources remain unchanged on the disk, so different files sharing this block will not get corrupted.

This method effectively implements deduplication on the local file system, reducing storage space required by approximately

$$block\_size \times \sum_{i=1}^{n} [(\text{\# of files sharing block i} - 1]$$

which can be a significant amount of space, when storing large similar files.

Apart from the space reduction, a FUSE mechanism implementation provides additional benefits for a file synchronisation framework. File copying within the directory managed by FUSE becomes a very fast and computationaly light procedure, since only a file's hash list needs to be copied to the new location. Furthermore, this optimisation works well in tandem with the file monitoring optimisation described in section 4.2. Being in control of the filesystem means we are able to immediately detect changes to files and process only those during the synchronisation. At the time of writing, the FUSE mechanism has been extensively design, but not yet implemented.

Chapter 5

# Comparisons with existing software

To the extent of my knowledge and research, there is no publicly available application, commercial or free, that was designed specifically for the sychronisation of large similar files, such as VM images and snapshots. Therefore, the following comparisons are with the most commonly used file synchronisation software for regular files. For the proprietary software, since there is no way to access the source code, empirical comparisons will be made, based on the results of some benchmarks aimed at feature detection.

## 5.1 Rsync

TODO: Rsync

## 5.2 ownCloud

TODO: Owncloud

## 5.3 OneDrive

TODO: OneDrive

## 5.4 Dropbox

TODO: Dropbox

## 5.5 Google Drive

TODO: GDrive

Chapter 6

# Future Work

## 6.1 Local deduplication - FUSE Implementation

TODO

## 6.2 Peer-to-peer syncing with direct L2 frame exchange

TODO: Dropbox's idea, but sync blocks, not files

# Bibliography

[1] T. Haerder and A. Reuter, “Principles of transaction-oriented database recovery,” *ACM Comput. Surv.*, vol. 15, pp. 287–317, Dec. 1983.

[2] H. Kung and J. T. Robinson, “On optimistic methods for concurrency control,” *ACM Transactions on Database Systems*, vol. 6, June 1981.

[3] “About sqlite.” https://sqlite.org/about.html. [Online, accessed July 2015].

[4] “Openstack: The open source cloud operating system.” https://www.openstack.org/software/. [Online, accessed July 2015].

[5] “Amazon s3.” https://aws.amazon.com/s3/. [Online, accessed July 2015].

[6] “Owncloud.” https://owncloud.org/faq. [Online, accessed August 2015].

[7] “csync.” https://www.csync.org/about/. [Online, accessed August 2015].

[8] Y. Collet, “Selecting a checksum algorithm.” http://fastcompression.blogspot.gr/2012/04/selecting-checksum-algorithm.html. [Online, accessed August 2015].

[9] Y. Collet, “xxhash.” https://github.com/Cyan4973/xxHash. [Online, accessed August 2015].

[10] “~okeanos.” https://okeanos.grnet.gr/about/what/. [Online, accessed August 2015].

[11] “Synnefo.” https://www.synnefo.org/about/. [Online, accessed August 2015].

[12] “Synnefo rest api guide.” https://www.synnefo.org/docs/synnefo/latest/api-guide.html. [Online, accessed August 2015].

[13] M.-A. Lemburg, “Python database api specification v2.0.” https://www.python.org/dev/peps/pep-0249/. [Online, accessed June 2015].

[14] “Wikipedia: Openstack.” https://en.wikipedia.org/wiki/OpenStack. [Online, accessed July 2015].

[15] “Wikipedia: Amazon s3.” https://en.wikipedia.org/wiki/Amazon_S3. [Online, accessed July 2015].

[16] “Wikipedia: File hosting service.” https://en.wikipedia.org/wiki/File_hosting_service. [Online, accessed July 2015].

[17] H. K. Abraham Silberschatz and S. Sudarshan, *Database System Concepts*. McGraw-Hill Higher Education, 6th ed., 2010.

[18] R. T. Fielding and R. N. Taylor, “Principled design of the modern web architecture,” in *Proceedings of the 22nd international conference on Software engineering*, ICSE ’00, pp. 407–416, 2000.

[19] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[20] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, The Australian National University, Irvine.