National Technical University of Athens

Design & Implementation of a Portable File Synchronisation Mechanism for a Cloud Storage Environment

Vasilis Gerakaris

2/9/2015

Table of Contents

Table of Contents

Introduction

(i) - The problem

File Synchronisation: The process of updating files in two or more different locations, following certain rules.

Why is it needed?

- Copying files between different computers
- Backups

Important Qualities

✓ Needs to detect & handle update conflicts/renames/deletions

✓ Needs to be reliable (no errors)

✓ Needs to be efficient

Introduction

(i) - The problem (cont)

File Synchronisation: The process of updating files in two or more different locations, following certain rules.

Software designed for that purpose already exists, namely:

- rsync
- ownCloud
- Dropbox
- Google Drive

We focus on a more specific aspect of the problem.

Large Similar Files

(i) - Definition

What are they?

Files that satisfy the following two requirements:

- Are large in size (several GBs)
- Have a lot of their data in common

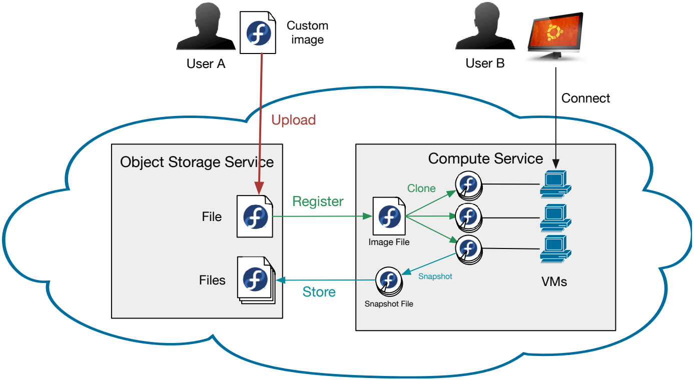
Examples: VM images, VM snapshots

Why are they important?

Many VMs are being used on cloud service providers (Amazon AWS, ~okeanos, etc) and there should be a way to efficiently synchronise their images and snapshots.

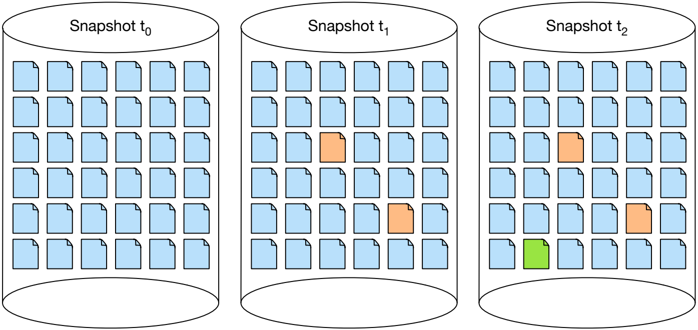
Large Similar Files

(ii) - Definition (cont)



Large Similar Files

(iii) - Definition (cont)



We can use these similarities to optimise the synchronisation!

Table of Contents

Syncing algorithm

(i) - Modification detection

Modification detection: Comparison of hash digests

✓ Reliable

✗ Very slow, especially on large files

Faster alternative: Use last modification time as an indicator.

Why we need history data:

Need to know what to do in the following cases:

- File exists on both locations and is different
- File exists on A but not on B (or vice-versa)

Syncing algorithm

(ii) - Initial algorithm

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

(a) File change detection between two points in time

File replica A	**File replica 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**

(b) Syncing actions based on file states

Syncing algorithm

(iii) - What we propose

Limitations

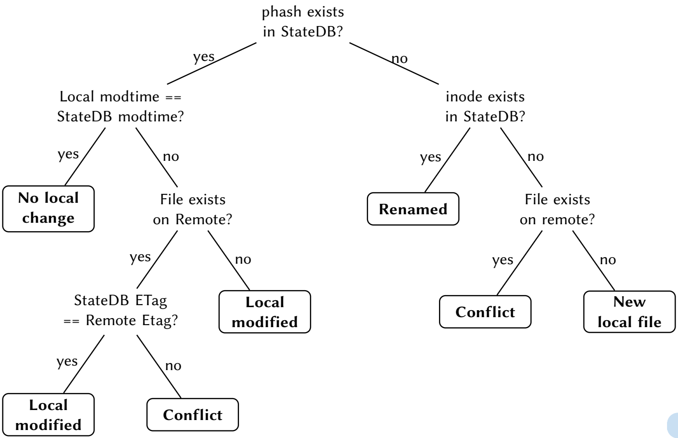
✗ Can't detect renames (or worse, renames & modifications)

Our solution for syncing with a central metadata server

- Store the metadata of all files, as they were during the last successful sync on a local state database (StateDB).
- Reconcile local directory replicas (Local) and remote server replicas (Remote) in three steps:
  1. Detect updates from Local Directory
  2. Detect updates from StateDB
  3. Detect updates from Remote Directory
- FCFS updates on conflicts, with conflicting copies being renamed.

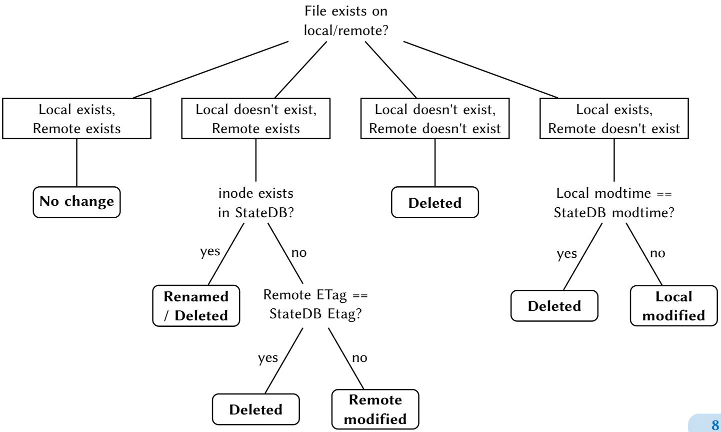
3-step synchronisation

(i) - Updates from Local Directory



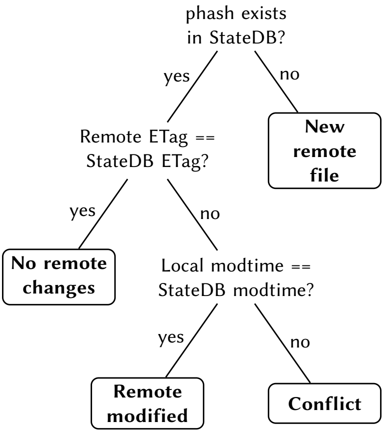
3-step synchronisation

(ii) - Updates from StateDB



3-step synchronisation

(iii) - Updates from Remote Directory



Core Classes / API

What we have done:

- Built a cross-platform framework in Python that can be used to synchronise files with any cloud storage service, as long as some API functions are implemented.
- Created abstract classes for representations of files, filesystem directories and cloud storage services.
- Implemented a class that uses the Synnefo (Pithos) API as an example.
- Created a proof-of-concept application that syncs a local directory with the Pithos+ service offered by ~okeanos.

Core Classes / API

(i) - FileStat

FileStat
phash: int
path: str
inode: int
modtime: int
type: int
etag: str

The core class used in this framework to represent file objects

- **phash**: The (integer) hash digest of the relative path string. It is used for fast indexing in the StateDB. Assumed unique for each file path.
- **etag**: The ETag (sha-256 digest) of the file. Assumed unique for each file version.

Core Classes / API

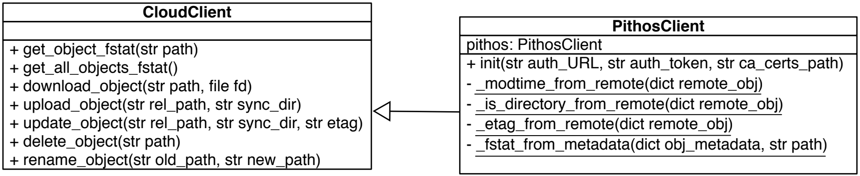
(ii) - LocalDirectory

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

- **get_all_objects_fstat**: Returns all local files' metadata as FileStat objects.
- **get_modified_objects_fstat**: Return file metadata only for the files that were modified since the last sync.
- **get_file_fstat**: Returns the FileStat object for the file *path* if it exists, else returns *None*.

Core Classes / API

(iii) - CloudClient



Closely resembles the OpenStack API (used by synnefo as well).

To properly handle race conditions:

upload_object() is used for new files

update_object() is used for existing files.

Table of Contents

Optimisation: Request Queuing

(i) - Description

Multiple requests are slow!

- ✓ Batch them wherever possible (get_all_objects_fstat())
- ✓ Use threads and queues to send requests without waiting for others to complete.

- ✗ Need to wait for completion of all threads at a step of the sync algorithm before proceding to the next.
  - Can be further optimised using a locking mechanism

Optimisation: Request Queuing

(ii) - Benchmark results

	# 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) Upload speedup by queuing, relative to # of threads

	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

(b) Upload speedup, relative to file size (4 threads)

Considerable speedup for smaller files, but less effective when network gets close to maximum throughput.

Optimisation: Directory Monitoring

(i) - Description

Checking all files for changes is slow!

- ~1000 files/s on an SSD
- 1M files $\Rightarrow$ 16.7 minutes!

✓ Operating Systems can have modification information available - directory monitoring mechanisms (inotify, FSEvents, kqueue, etc)

We use the watchdog Python module to access those utilities, extending the LocalDirectory class to support the feature.

- Constantly runs in the background (daemon). Offline changes/crashes/reboots are handled by performing a full local directory scan on startup.

Optimisation: Directory Monitoring

(ii) - Benchmark results

Setup: Directory with 1M files, modify some of them and measure time of update detection.

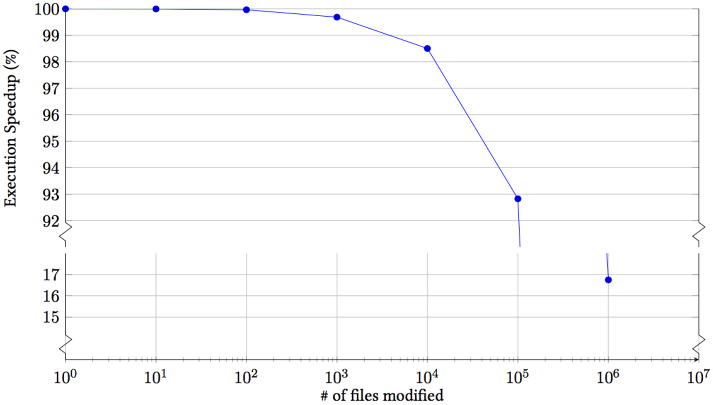
	# files modified							
	0	**10**	**100**	**1000**	**10000**	**100000**	**1000000**	**default**
time (s)	1.06E-5	0.004	0.038	0.339	1.618	12.907	90.003	108.110
speedup (%)	100.000	99.996	99.965	99.687	98.503	92.825	16.749	N/A

- ✓ Significant speedup when a small number of files has changed (most common scenario)
- ✓ Small speedup even in the cases where many files have changed

Graphical representation of the results on the next slide
(**Note:** Lin-Log scale)

Optimisation: Directory Monitoring

(iii) - Graphical representation of results



Optimisation: Local Block Storage

(i) - Description

Downloading whole large files for small changes is slow!

Implement delta-sync:

✓ Keep a local copy of all files' blocks

✓ Detect what parts of files have been changed

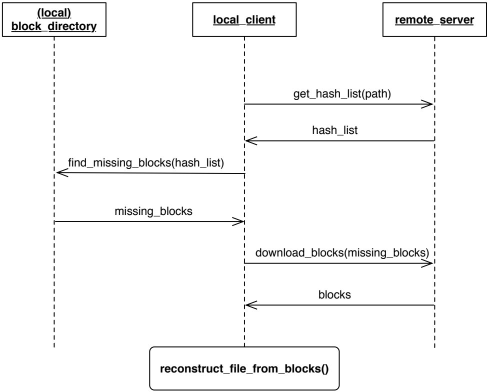
✓ Download only the missing blocks and create the file

✗ Needs extra storage space to store all the different blocks

- Extend the CloudClient class to handle downloads using blocks.
- Use hierarchical structure to improve block lookup speed.
- Save local modified blocks after uploads/updates.

Optimisation: Local Block Storage

(ii) - Sync Process



Optimisation: Local Block Storage

(iii) - Benchmark results

Setup: Create and upload a 40 MiB (41,943,040 B) file (exactly 10 blocks of 4 MiB), modify some blocks, manually reupload to server, measure download times.

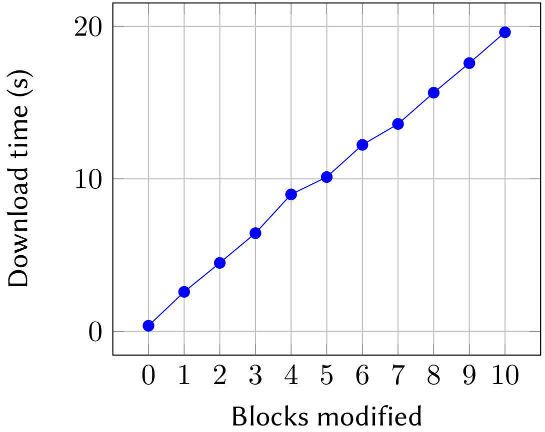
	# of modified blocks										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
time (s)	0.37	2.59	4.49	6.44	8.98	10.12	12.23	13.60	15.65	17.59	19.61
speedup (%)	98.1	86.8	77.1	67.2	54.2	48.4	37.7	30.7	20.2	10.3	N/A

- ✓ Linear correlation
- ✓ Significant improvement for large similar files, since very few blocks need to be downloaded each time

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

Optimisation: Local Block Storage

(iv) - Graphical representation



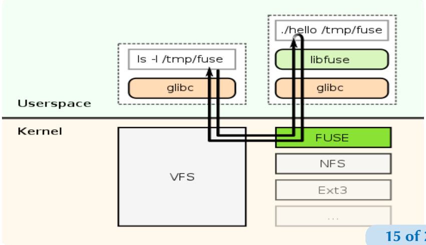
Local Deduplication - FUSE

(i) - Description

Storing so many large files is expensive!

- ✓ Those files have the majority of their blocks in common
- ✓ We only need to store each block once, in the block directory
- ✓ Need to control the FS, so we can "virtually" create the files

Solution:
Filesystem in Userspace (FUSE) mechanism.



Local Deduplication - FUSE

(ii) - Design

- Modify *fstat(), open(), read(), write()* system calls to use the blocks a file consists of.
- "Write once, Read many, Update never" practice
- Copy-on-Write (CoW) strategy, to preserve possibly shared blocks when changes are made

Effectively implements deduplication on the local file system. Storage space reduction of approximately:

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

Also offers other benefits (cheap file copies, immediate modification detection)

Table of Contents

Comparison with existing software

rsync

- ✓ Rolling hash algorithm performs exceptionally on detecting modified parts.
- ✓ Does not need files to be aligned to blocks
- ✓ One round-trip, works well on high latency connections
- ✓ Free & Open source software
- ✗ Not automated
- ✗ Needs third-party applications to handle synchronisation
- ✗ No directory monitoring
- ✗ Uses MD5 for checksum comparison (potentially unsafe - collisions can be computed)
- ✗ No local file deduplication

Comparison with existing software

ownCloud

- ✓ Most famous open source synchronisation software suite
- ✓ Cross-platform
- ✓ Directory monitoring
- ✗ No delta-sync - Transfer whole files
- ✗ Full local directory scan every few minutes
- ✗ No local file deduplication
- ✗ Silently ignores files containing special characters which are not allowed in Windows ('|', ':', '>', '<' and '?')

Comparison with existing software

Dropbox

✓ Uses librsync - rolling checksum algorithm benefits
✓ Remote deduplication with blocks of 4 MiB - Fast uploads of similar files
✓ Benchmarks indicated the existance of a local block cache - fast downloads if blocks are cached
✓ Directory Monitoring
✓ Streaming Sync for multiple clients (Prefetching blocks)

✗ Commercial, closed source software
✗ Cannot be deployed on personal cloud storage infrastructures or other cloud storage services
✗ No local file deduplication

Comparison with existing software

Google Drive

- ✓ Directory monitoring
- ✗ No delta-sync - Transfer whole files
- ✗ Commercial, closed source software
- ✗ Cannot be deployed on personal cloud storage infrastructures or other cloud storage services
- ✗ No local file deduplication

Table of Contents

Future Work

Peer-to-Peer L2 block exchange

Idea: LAN transfers are faster than over the WAN.

Request missing resources from the LAN, before asking the server.

- Have clients monitor a Link Layer (L2) broadcast address for requests.
- Send missing block requests to the network and wait for responses.
- When asked, clients check their respective block directories and respond with block availability.
- Only request blocks not found in the block directory or the local network from the remote server.
- **ALWAYS verify blocks downloaded from LAN** - Avoid corruption or compromise from blocks sent by malicious users.

Q & A

Any Questions?

The End.

Thank you for your time!