Gimme:
A simple versioning system for shared folders

February 3, 2019

1 Introduction

We present here gimme, a simple versioning system for shared folders. It uses an assymetric model, in which one of the parties acts as the "master" or "real truth", and the other is a dirty local copy.

1.1 Possible names

The names to be used to reference the two parties involved are not fixed yet. Possible pairs include :

- master version vs local copy
- truth vs dirty copy
- remote vs local
- shared vs local
- accepted version vs pending changes

2 Model

We describe here the model used by gimme.

2.1 Master version vs local copy

gimme uses an assymetric model, where one of the two parties involved acts as a "master", and the other is seen as a local copy.

The master version holds the shared truth for all the systems. It holds all the information, in the latest *accepted* state. Information (conceptually speaking) should never disappear from the master version. its goal is to aggregate all the information present on all the systems.

The local copy is a working directory for a single machine. Files can be modified, copied, added, removed, etc... without affecting the master version. In this case, the copy is called *dirty*.

Changes to the local copy are shared with all systems by calling gimme. These changes then become part of the shared truth. We say these changes are *accepted*. This is called an *upload*, or *update*.

The other way around, changes that are part of the shared truth but are not present in the local copy are also synchronized upon calling gimme. This process is called *download* or *upgrade*.

2.2 Subscription model

One of the big differences between the master version and local copies is that local copies do not necessarily have all the information, and maybe they do not want it.

We model this using *subscriptions*. Local copies explicitely subscribe to a specific set of folders and files. A local copy is not interested in updates or removal of folders or files it does not subscribe to. When a new folder or file is added in a folder that the local copy is subscribed to, the user it prompted to know whether this copy should subscribe to this new element.

local \ master	(ADD, SAM)	(ADD, UPD)	(ADD, RM)
(SUB, SAM)	(SUB, nothing, ADD, nothing)	(SUB, cp, ADD, upd-hash)	(RM, rm, RM, nothing)
(SUB, UPD)	(SUB, upd-hash, ADD, cp)	(SUB, latest, ADD, latest)	[1][SUB, ADD] / [RM, RM]
(SUB, RM)	[2][UNSUB, ADD] / [RM, RM]	[2][UNSUB, ADD] / [RM, RM]	(RM, nothing, RM, nothing)

Table 1: State machine for tracked files

local \ master	(RM, ADD)	(RM, RM)
(RM, ADD)	(SUB, latest, ADD, latest)	(SUB, upd-hash, ADD, cp)
(RM, RM)	[3][SUB, ADD] / [UNSUB, ADD]	(RM, nothing, RM, nothing)

Table 2: State machine for untracked files

It is still unclear what the exact granularity of subscriptions is. It should be possible to choose which folders to subscribe to, and which not too. There could be a mechanism to tell that a folder is subscribed to all its (future) content. This can be bound to the folder, or only to the current update.

The only reason to have a granularity at the level of single files would be to allow deleting a file locally because it is not necessary, but keeping it in the master version. That use-case seems very obscure. When speaking about music files, a similar result can be achieved by removing the file from the library.

To subscribe, the folder only needs to be present in the local copy. New folders added in the master version can be proposed for copy in the local copy, but this process should not be the default. It is always possible to copy a folder from the master file system to subscribe to it.

To unsubscribe, the folder needs to be deleted on the local copy. Then, when computing the changelog, the user should tell explicitely that this deletion is an unsubscribe, and should not be reflected in the master version. There is no default between the two behaviours, the choice must be made explicit anyway.

2.3 Data considered

We consider folders and files.

2.4 Possible actions

For a given element (either file or folder), the possible actions can be :

1. Add
2. Delete
3. Update

Note that folders can only be updated if their content has been modified (any of the changes listed above).

3 The synchronization process

We describe here the different steps that must be performed when synchronizing two devices.

3.1 Workflow

3.2 Computing the changelogs

3.3 State machine

3.4 Simple merge

3.5 Merge conflicts

	Add	Update	Delete	Unsubscribe
Add	3.5.5			
Update		3.5.6	3.5.4	
Delete	3.5.2	3.5.1		

local \ master	(ADD, ADD)	(ADD, RM)
(SUB, ADD)	(SUB, nothing, ADD, nothing)	(RM, rm, RM, nothing)
(SUB, RM)	[2][UNSUB, ADD] / [RM, RM]	(RM, nothing, RM, nothing)
(UNSUB, ADD)	(SUB, merge, ADD, merge)	[1][SUB, ADD] / [RM, RM]
(UNSUB, RM)	(UNSUB, nothing, ADD, nothing)	(RM, nothing, RM, nothing)

Table 3: State machine for tracked folders

local \ master	(RM, ADD)	(RM, RM)
(RM, ADD)	(SUB, merge, ADD, merge)	(SUB, nothing, ADD, cp)
(RM, RM)	[3][SUB, ADD] / [UNSUB, ADD]	(RM, nothing, RM, nothing)

Table 4: State machine for untracked folders

state	action
(SUB, SAM)	rm
(SUB, UPD)	Conflict
(SUB, RM)	nothing
(RM, ADD)	Conflict
(RM, RM)	nothing

Table 5: rm mechanism for files. In case of conflicts, keep file and print warning

state	action
(SUB, ADD)	recurse then rmdir
(SUB, RM)	nothing
(UNSUB, ADD)	Conflict
(UNSUB, RM)	nothing
(RM, ADD)	Conflict
(RM, RM)	nothing

Table 6: rm mechanism for folders. In case of conflicts, keep folder and print warning

3.5.1 Master delete and local update

3.5.2 Master delete and local add

Folder subscribed to or not before ?

3.5.3 Master add and local delete

3.5.4 Master update and local delete

Unsubscribe, except if on file

3.5.5 Master add and local add

3.5.6 Master update and local update

4 Commit

4.1 Commit point

4.2 Elements part of the commit / changelog