Magic-Folder Documentation

Release 1.x

The Tahoe-LAFS Developers, The Magic-Folder Developers

May 21, 2021

Contents

Magic Folder is a Tahoe-LAFS front-end that synchronizes local directories on two or more clients. It uses a Tahoe-LAFS grid for storage. Whenever a file is created or changed under the local directory of one of the clients, the change is propagated to the grid and then to the other clients.

The implementation of the "drop-upload" frontend, on which Magic Folder is based, was written as a prototype at the First International Tahoe-LAFS Summit in June 2011. In 2015, with the support of a grant from the Open Technology Fund, it was redesigned and extended to support synchronization between clients. It should work on all major platforms.

Warning: At the time of this writing, we are in the process of refactoring Magic Folder out of Tahoe-LAFS source tree and re-writing it. Because of this state of development, the documentation you read here may not be up-to-date, and subject to changes.

Magic Folder is not currently in as mature a state as the other Tahoe-LAFS frontends (web, CLI, SFTP and FTP). This means that you probably should not rely on all changes to files in the local directory to result in successful uploads. There might be (and have been) incompatible changes to how the feature is configured.

We are very interested in feedback on how well this feature works for you. We welcome suggestions to improve its usability, functionality, and reliability. Please file issues you find with Magic Folder at the GitHub project, or chat with us on IRC in the channel `#tahoe-lafs` on `irc.freenode.net`.

CHAPTER 1

Contents

1.1 Using Magic Folder

Magic-Folder is used in two ways. To interact with configuration, the `magic-folder` command line tool is used. For details of this, see the section on config-file. For additional information see Magic Folder CLI design.

To interact with content, use your normal filesystem-based tools. The folder which Magic-Folder synchronizes is a normal folder in the filesystem. The platform's filesystem change notification features are used to detect changes.

We think of participants in the system as "devices". A single human may control many devices (in case they are synchronizing files between them, for example). It is possible that a magic-folder setup serves just a single human or it may serve many. Each human may have one or many devices. When we talk about "an author" below (equivalent to the `--author` CLI option, usally) this means a single device. A single human may control many "authors".

1.1.1 Prerequisites

You must have one or more Tahoe-LAFS client nodes configured to be able to store objects somewhere. They must be able to reach their configured storage nodes. The client nodes must all share the same storage nodes. The nodes must be running.

1.1.2 Creating a Magic Folder Daemon Configuration

The "Magic Folder Daemon" is a long-running process that accomplishes the task of synchronizing all the "magic folders" that are configured in it.

A Magic Folder Daemon needs to have some configuration to start. There are two ways to create this: from scratch; or from a "legacy" magic-folder (when `magic-folder` was a sub-command of `tahoe` in version 1.14.0 and earlier).

No matter the method used, there are some pieces of information required and two decisions to be made. You need to know:

- the `node-directory` of the Tahoe-LAFS client this magic folder daemon shall use.

You need to decide:

- The network endpoint the API will listen on. This is how CLI commands and front-ends communicate to the magic-folder daemon and is expressed as a Twisted "server endpoint string". For example, `tcp:4321:interface=localhost` to listen locally on port 4321.
- If you wish to specify where to store the configuration. By default it will be in an appropriate location for your OS (like `~/.config/magic-folder` on Debian).

To create a magic folder daemon configuration from scratch:

```
$ magic-folder init --config ./foo --listen-endpoint tcp:4321:interface=localhost --
↪node-directory ./tahoe-client
```

This will store configuration in `./foo`; listen on `localhost` port `4321` for API commands; and talk to the Tahoe-LAFS client in `./tahoe-client` (which must itself be running).

To create a magic folder daemon configuration by migrating an existing Tahoe-LAFS-based magic-folder, do:

```
$ magic-folder migrate --config ./foo --listen-endpoint tcp:4321:interface=localhost -
↪-node-directory ./tahoe-client --author alice
```

The main difference is the `--author` argument. This is required when creating signing keys for each configured magic-folder that is migrated over to the new daemon. Note that the Tahoe-LAFS configuration (`./tahoe-client` in the example) will be left alone.

From now on, we will assume there is a valid magic folder daemon configuration in `./foo`. This is usually provided to all sub-commands like so: `magic-folder --config ./foo <subcommand>`. The default location is used if `--config` is not specified.

1.1.3 Running a Magic Folder process

Our magic folder daemon has a configuration location so now it can be started. It must be running for most of the other magic-folder commands to work as they use the API.

```
$ magic-folder --config ./foo run
```

Remember that the Tahoe-LAFS node which the daemon uses to upload and download items from the Grid must also be running.

1.1.4 Creating Magic Folders

A new magic folder is added using the `magic-folder add` command.

```
$ magic-folder --config ./foo add --author alice-laptop --name example ~/Documents
```

There are some other options that can be specified. The above will create a new magic-folder named `example` (we could decide differently with `--name docs` for example). Any changes we make locally will be signed as `alice-laptop`. Files from other devices are downloaded into `~/Documents` and any files we add or change in that local directory will be uploaded. Note that deleting a file in `~/Documents` will record a new "deleted" version in Tahoe Grid and not actually remove data.

It is also possible to specify `--poll-interval` to control how often the daemon will check for updates if the default seems wrong.

This device will be the administrator for a magic folder created in this manner (that is, only this device can invite new participants).

See `magic-folder create --help` for specific usage details.

1.1.5 Listing Magic Folders

Existing magic folders can be listed using the `magic-folder list` command:

```
$ magic-folder --config ./foo list
This client has the following magic-folders:
example:
    location: /home/alice/Documents
   stash-dir: /home/alice/foo/example/stash
      author: alice-laptop (public_key:␣
↪KSYPPXN3HTCSEJC56RRYXDEO2TZX5LO743Q3E2M7NA7UP2W3OK2A====)
     updates: every 60s
```

To get JSON output, pass `--json`. You can include sensitive secret information by passing `--include-secret-information` flag. Someone who obtains this information can impersonate this device and participate as it in the magic folder (if they also gain access to the Tahoe-LAFS Grid being used).

1.1.6 Inviting Participant Devices

A new participant device is invited to collaborate on a magic folder using the `magic-folder invite` command. This produces an "invite code" which is a one-time code. This code should be communicated securely to the invitee. The code will allow the invitee's device to establish a connection to this device and exchange details. Thus, the code can only be used while this device is connected to the Internet. The code may only be used once, for a single invitee.

```
$ magic-folder --config ./foo invite --name example
```

An invitation code is created using an existing magic folder (`--name example` above). The magic-folder identified must have been created on this device.

Once the invitee runs `magic-folder join` (see below) the two devices will connect and exchange some information; this will complete the invitation. The "invite" command won't exit until the invitee has actually completed and will print out some details. If you pass `--no-wait` then the command will exit immediately (although the invite will still be valid).

XXX DECIDE: should the default be to wait, or to not? Developers are split on this; maybe some UX research or discussion can solve it? No matter what, the HTTP API will have to be two-part ("start invite -> X" and "status of invite X" or "wait for invite X")

Invites are valid until the magic-folder daemon stops running or until the default number of minutes pass (whichever is sooner). See the `--timeout` options for the default (or you can pass a different number of mintues if you prefer).

1.1.7 Joining a Magic Folder

A participant device accepts an invitation using the `magic-folder join` command:

```
$ magic-folder --config ./foo join $INVITECODE /home/bob/Documents/Shared
```

The first argument required is an invitation code, as described in *Inviting Participant Devices*. The second argument required is the path to a local directory. This is the directory to which content will be downloaded and from which it will be uploaded.

You must choose a name to identify content from this device with `--author`. The device which has invited you must also be connected to the internet for the invite to work: once a connection is established, the two devices exchange some information and the invite is complete.

Further options are documented in `magic-folder join --help`.

1.1.8 Leaving a Magic Folder

A participant device can reverse the action of joining a magic folder using the `magic-folder leave` command.

You must supply the name of the magic folder to leave with `--name`. Once a device has left a magic folder, further changes to files in the folder will not be synchronized. The local synchronized directory itself is not removed. **All configuration and state for the magic-folder is destroyed**.

Note that by default you cannot leave a folder that this device has created as it has the only copy of the write-capability which allows one to change the list of participants. If you really do want to `leave` such a folder you can indicate this desire and override the error with `--really-delete-write-capability`.

See `magic-folder leave --help` for details.

1.1.9 A quick test

If you want to test that things work as expected using a single machine, you can create two separate Tahoe-LAFS nodes, and assign corresponding magic folders with them, like so:

```
$ export ALICE_NODE=./grid/alice
$ export ALICE_FOLDER=./alice-sync-dir
$ export ALICE_MAGIC=./grid/alice-magic

$ export BOB_NODE=./grid/bob
$ export BOB_FOLDER=./bob-sync-dir
$ export BOB_MAGIC=./grid/bob-magic

# create magic-folder daemons and run them for alice+bob
$ mkdir -p $ALICE_FOLDER
$ mkdir -p $BOB_FOLDER
$ magic-folder init --node-directory $ALICE_NODE --listen-endpoint␣
↪tcp:4000:interface=localhost --config $ALICE_MAGIC
$ magic-folder init --node-directory $BOB_NODE --listen-endpoint␣
↪tcp:4001:interface=localhost --config $BOB_MAGIC
$ daemonize magic-folder --config $ALICE_MAGIC run
$ daemonize magic-folder --config $BOB_MAGIC run

# alice creates a magic-folder and invites bob
$ magic-folder --config $ALICE_MAGIC add --name example alice $ALICE_FOLDER
$ magic-folder --config $ALICE_MAGIC invite --name example bob >invitecode
$ export INVITECODE=$(cat invitecode)
$ magic-folder --config $BOB_MAGIC join --name example "$INVITECODE" $BOB_FOLDER
```

You can now experiment with creating files and directories in `./alice-magic` and `./bob-magic`. Any changes in one should be propagated to the other directory.

Note that when a file is deleted, the corresponding file in the other directory will be renamed to a filename ending in `.backup`. Deleting a directory will have no effect.

For other known issues and limitations, see *Known Issues and Limitations*.

It is also possible to run the nodes on different machines, to synchronize between three or more clients, to mix Windows and Linux clients, and to use multiple servers (as long as the Tahoe-LAFS encoding parameters are changed).

1.2 How invites work

Let us clarify the glossary first:

- A **folder** is an abstract directory that is synchronized between clients. A folder is not the same as the directory corresponding to it on any particular client, nor is it the same as a DMD.
- **DMD**, abbrevition for Distributed Mutable Directory, is a physical Tahoe-LAFS mutable directory. Each client has the write capability to their own DMD, and read capabilities to all other client's DMDs.
- A **collective** is the set of clients subscribed to a given Magic Folder.

1.2.1 Invitation Process

Suppose Alice wants to share files with Bob. These are the steps Alice would take.

0. Alice creates a magic-folder with a name, say `/home/alice/Documents/Shared`. She then proceeds to "invite" Bob to the folder. Only Alice as the creator of the folder can invite other people into the folder.
1. Alice would create an unlinked directory and get the write cap. This directory would form the "DMD" of the folder. Let us call it `dmd_write_cap`.
2. Alice would derive a read-only cap from the write cap derived in step #1. Let us call it `dmd_read_cap`.
3. Alice creates a "collective" directory (Alice has write caps to it) and stores an alias corresponds to that directory in her `private/aliases` file. Let us call the write cap to the collective a `collective_write_cap`. Note that there should only be one user who has writecaps to the collective folder, since concurrent writes to a mutable directory by multiple users is not guaranteed to be consistent.
4. Alice derives a read-only cap (`collective_read_cap`) to the collective.
5. Alice stores read-only cap from step #2 in `<collective>/nick` file.
6. To join the collective, Alice sends the string, `collective-read-cap + dmd_write_cap` to Bob.
7. Bob stores the collective_read_cap as "collective_dircap" and the `dmd_write_cap` as the `upload_dircap` in his in config file, `private/magic_folders.yaml`.

At this point Alice has stored Bob's nick in the collective. (Alice already stored her own nick during the "create" process when she invited herself into the collective).

1.3 Known Issues and Limitations

- The only way to determine whether uploads have failed is to look at the 'Operational Statistics' page linked from the Welcome page. This only shows a count of failures, not the names of files. Uploads are never retried.
- The Magic Folder frontend performs its uploads sequentially (i.e. it waits until each upload is finished before starting the next), even when there would be enough memory and bandwidth to efficiently perform them in parallel. A Magic Folder upload can occur in parallel with an upload by a different frontend, though. (#1459)
- On Linux, if there are a large number of near-simultaneous file creation or change events (greater than the number specified in the file `/proc/sys/fs/inotify/max_queued_events`), it is possible that some events could be missed. This is fairly unlikely under normal circumstances, because the default value of `max_queued_events` in most Linux distributions is 16384, and events are removed from this queue immediately without waiting for the corresponding upload to complete. (#1430)
- The Windows implementation might also occasionally miss file creation or change events, due to limitations of the underlying Windows API (ReadDirectoryChangesW). We do not know how likely or unlikely this is. (#1431)
- Some filesystems may not support the necessary change notifications. It is recommended for the local directory to be on a directly attached disk-based filesystem, not a network filesystem or one provided by a virtual machine.

- If a file in the upload directory is changed (actually relinked to a new file), then the old file is still present on the grid, and any other caps to it will remain valid. Eventually it will be possible to use garbage collection to reclaim the space used by these files; however currently they are retained indefinitely. (#2440)
- Unicode filenames are supported on both Linux and Windows, but on Linux the local name of a file must be encoded correctly in order for it to be uploaded. The expected encoding is that printed by `python -c "import sys; print sys.getfilesystemencoding()"`.
- On Windows, local directories with non-ASCII names are not currently working. (#2219)
- On Windows, `magic-folder run` may be unresponsive to Ctrl-C (it can only be killed using Task Manager or similar). (#2218)

1.4 Statement on Backdoors

October 5, 2010

The New York Times has recently reported that the current U.S. administration is proposing a bill that would apparently, if passed, require communication systems to facilitate government wiretapping and access to encrypted data.

(login required; username/password pairs available at bugmenot).

Commentary by the Electronic Frontier Foundation, Peter Suderman / Reason, Julian Sanchez / Cato Institute.

The core Magic-Folder developers promise never to change Magic-Folder to facilitate government access to data stored or transmitted by it. Even if it were desirable to facilitate such access – which it is not – we believe it would not be technically feasible to do so without severely compromising Magic-Folder's security against other attackers. There have been many examples in which backdoors intended for use by government have introduced vulnerabilities exploitable by other parties (a notable example being the Greek cellphone eavesdropping scandal in 2004/5). RFCs 1984 and 2804 elaborate on the security case against such backdoors.

Note that since Magic-Folder is open-source software, forks by people other than the current core developers are possible. In that event, we would try to persuade any such forks to adopt a similar policy.

The following Magic-Folder developers agree with this statement:

Jean-Paul Calderone

1.5 Magic-Folder Development

1.5.1 Process Overview

The high-level view of the Magic-Folder development process looks like this:

1. A shortcoming, defect, or new area of functionality is identified relating to the software. For example, perhaps files with names matching a certain pattern are never uploaded to the grid or a developer decides to add a history browsing user experience to the software.
2. *An issue is filed* describing the shortcoming, defect, or new area of functionality. The issue describes how the software will be different than it is now and why this is desirable.
3. Development work proceeds in a branch (typically created from a recent master branch revision). Prior to this development or in parallel with it, communication within the team helps identify and clear roadblocks early.
4. When development has progressed sufficiently, the branch is *proposed for review*.
5. One or more other developers *review the changes*.

6. The previous two steps may repeat a number of times until the changes pass review.
7. Upon passing review, the changes are *merged into the master branch*.

1.5.2 Filing Issues

The project uses GitHub issues for issue tracking.

1.5.3 Proposing Changes

To propose a change, create a GitHub Pull Request against master.

Changes should:

- have a corresponding ticket.
- be in a branch named `<ticket number>.<some short descriptive text>`.
- have appropriate developer-facing and user-facing documentation.
- have full line and branch coverage provided by the test suite.
- be implemented in a way that is not unnecessarily difficult to understand and maintain.
- satisfy all of the mechanical checks performed by the continuous integration system.

News Fragments

One of the mechanical checks performed by continuous integration against proposed changes is the existence of a "news fragment". News fragments are assembled at release time by towncrier to contribute to the release announcement. News fragments are meant to be user-facing and should have a consistent style. News fragments can use any inline formatting directives from reStructuredText.

magic-folder's news fragment style is adapted from the style guidelines from the Twisted project. The fragment types accepted are canonically defined by the towncrier configuration file in the project root.

Here are a few guidelines which should help you write good news fragments:

- The entry SHOULD contain a high-level description of the change suitable for end users.
- When the changes touch Python code, the grammatical subject of the sentence SHOULD be a Python class/method/function/interface/variable/etc, and the verb SHOULD be something that the object does. The verb MAY be prefixed with "now".
- When the changes touch user interface, the grammatical subject of the sentence SHOULD identify the part of the user interface affected and the verb SHOULD be something that the user interface does.
- For bugfix, it MAY contain a reference to the version in which the bug was introduced.

Fragment files MUST be placed in `newsfragments` and be named `<ticket number>.<fragment type>`. You can preview the generated news content using `tox -e draftnews`.

Backwards Incompatible Changes

The fragment type is `incompat`. Here are some examples.

```
``magic-folder create`` now requires the ``--path`` option.
```

```
Files in a Magic Folder with a name beginning with ``.`` are now uploaded instead of␣
↪ignored.
```

Features

The fragment type is `feature`. Here are some examples.

```
The downloader can now detect three-party conflicts.
```

```
On Windows, the uploader now notices file changes immediately instead of after a␣
↪several minute delay.
```

Bug Fixes

The fragment type is `bugfix`. Here are some examples.

```
The uploader no longer fails to upload all files with a space in their name.
```

```
The downloader now writes all files beneath the Magic Folder directory instead of the␣
↪user's home directory.
```

Dependency/Installation Changes

The fragment type is `installation`. Here are some examples.

```
magic-folder now requires Python 3.8 or newer.
```

```
magic-folder now requires pynacl.
```

Configuration Changes

The fragment type is `configuration`.

Removed Features

These are changes which deprecate or remove a feature. The fragment type is `removed`.

Here are some examples.

```
All HTTP API endpoints beneath ``/v1`` are now deprecated in favor of the ``/v2``␣
↪endpoints.
```

```
The HTTP API, deprecated since v1.2.3 in favor of the OTP API, has been removed.
```

Other Changes

These are changes that don't easily fit into another category. The fragment type is `other`.

Misc/Other

The fragment type is `minor`.

These fragments should be empty. Their contents will not be included in the assembled news file.

1.5.4 Reviewing Changes

First and foremost, the reviewer's job is to ensure the objective of the corresponding ticket has been satisfied.

Some specific areas to which a reviewer can pay attention:

- Is the implementation unnecessarily difficult for a human reader to understand (and maintain)?
- Does the test suite make correct assertions about the behavior of the code under test?
- Does the documentation (developer- and user-facing) accurately describe the new behavior?

Beyond these areas there are a number of mechanical checks applied by the continuous integration system. Changes should only be accepted if all of these mechanical checks pass *or* if there are failures which are certainly unrelated to the changes and for which tickets have been filed.

1.5.5 Merging Changes

Use the GitHub merge button to merge changes to master. Merge changes if they pass the mechanical continuous integration checks and the softer reviewer guidelines above.

1.6 Proposed Specifications

This directory is where we hold design notes about upcoming/proposed features. Usually this is kept in tickets on the bug tracker, but sometimes we put this directly into the source tree.

Most of these files are plain text, should be read from a source tree. This index only lists the files that are in .rst format.

1.6.1 Magic Folder local filesystem integration design

Scope

This document describes how to integrate the local filesystem with Magic Folder in an efficient and reliable manner. For now we ignore Remote to Local synchronization; the design and implementation of this is scheduled for a later time. We also ignore multiple writers for the same Magic Folder, which may or may not be supported in future. The design here will be updated to account for those features in later Objectives. Objective 3 may require modifying the database schema or operation, and Objective 5 may modify the User interface.

Tickets on the Tahoe-LAFS trac with the otf-magic-folder-objective2 keyword are within the scope of the local filesystem integration for Objective 2. *Local scanning and database*

When a Magic-Folder-enabled node starts up, it scans all directories under the local directory and adds every file to a first-in first-out "scan queue". When processing the scan queue, redundant uploads are avoided by using the same mechanism the Tahoe backup command uses: we keep track of previous uploads by recording each file's metadata such as size, `ctime` and `mtime`. This information is stored in a database, referred to from now on as the magic folder db. Using this recorded state, we ensure that when Magic Folder is subsequently started, the local directory tree can be scanned quickly by comparing current filesystem metadata with the previously recorded metadata. Each file referenced in the scan queue is uploaded only if its metadata differs at the time it is processed. If a change event is detected for a file that is already queued (and therefore will be processed later), the redundant event is ignored.

To implement the magic folder db, we will use an SQLite schema that initially is the existing Tahoe-LAFS backup schema. This schema may change in later objectives; this will cause no backward compatibility problems, because this new feature will be developed on a branch that makes no compatibility guarantees. However we will have a separate SQLite database file and separate mutex lock just for Magic Folder. This avoids usability problems related to mutual exclusion. (If a single file and lock were used, a backup would block Magic Folder updates for a long time, and a user would not be able to tell when backups are possible because Magic Folder would acquire a lock at arbitrary times.)

Eventual consistency property

During the process of reading a file in order to upload it, it is not possible to prevent further local writes. Such writes will result in temporary inconsistency (that is, the uploaded file will not reflect what the contents of the local file were at any specific time). Eventual consistency is reached when the queue of pending uploads is empty. That is, a consistent snapshot will be achieved eventually when local writes to the target folder cease for a sufficiently long period of time.

Detecting filesystem changes

For the Linux implementation, we will use the inotify Linux kernel subsystem to gather events on the local Magic Folder directory tree. This implementation was already present in Tahoe-LAFS 1.9.0, but needs to be changed to gather directory creation and move events, in addition to the events indicating that a file has been written that are gathered by the current code.

For the Windows implementation, we will use the `ReadDirectoryChangesW` Win32 API. The prototype implementation simulates a Python interface to the inotify API in terms of `ReadDirectoryChangesW`, allowing most of the code to be shared across platforms.

The alternative of using NTFS Change Journals for Windows was considered, but appears to be more complicated and does not provide any additional functionality over the scanning approach described above. The Change Journal mechanism is also only available for NTFS filesystems, but FAT32 filesystems are still common in user installations of Windows.

When we detect the creation of a new directory below the local Magic Folder directory, we create it in the Tahoe-LAFS filesystem, and also scan the new local directory for new files. This scan is necessary to avoid missing events for creation of files in a new directory before it can be watched, and to correctly handle cases where an existing directory is moved to be under the local Magic Folder directory.

User interface

The Magic Folder local filesystem integration will initially have a provisional configuration file-based interface that may not be ideal from a usability perspective. Creating our local filesystem integration in this manner will allow us to use and test it independently of the rest of the Magic Folder software components. We will focus greater attention on user interface design as a later milestone in our development roadmap.

The configuration file, `tahoe.cfg`, must define a target local directory to be synchronized. Provisionally, this configuration will replace the current `[drop_upload]` section:

```
[magic_folder]
enabled = true
local.directory = "/home/human"
```

When a filesystem directory is first configured for Magic Folder, the user needs to create the remote Tahoe-LAFS directory using `tahoe mkdir`, and configure the Magic-Folder-enabled node with its URI (e.g. by putting it in a file `private/magic_folder_dircap`). If there are existing files in the local directory, they will be uploaded as a result of the initial scan described earlier.

1.6.2 Magic Folder design for remote-to-local sync

Scope

In this Objective we will design remote-to-local synchronization:

- How to efficiently determine which objects (files and directories) have to be downloaded in order to bring the current local filesystem into sync with the newly-discovered version of the remote filesystem.
- How to distinguish overwrites, in which the remote side was aware of your most recent version and overwrote it with a new version, from conflicts, in which the remote side was unaware of your most recent version when it published its new version. The latter needs to be raised to the user as an issue the user will have to resolve and the former must not bother the user.
- How to overwrite the (stale) local versions of those objects with the newly acquired objects, while preserving backed-up versions of those overwritten objects in case the user didn't want this overwrite and wants to recover the old version.

Tickets on the Tahoe-LAFS trac with the otf-magic-folder-objective4 keyword are within the scope of the remote-to-local synchronization design.

Glossary

Object: a file or directory

DMD: distributed mutable directory

Folder: an abstract directory that is synchronized between clients. (A folder is not the same as the directory corresponding to it on any particular client, nor is it the same as a DMD.)

Collective: the set of clients subscribed to a given Magic Folder.

Descendant: a direct or indirect child in a directory or folder tree

Subfolder: a folder that is a descendant of a magic folder

Subpath: the path from a magic folder to one of its descendants

Write: a modification to a local filesystem object by a client

Read: a read from a local filesystem object by a client

Upload: an upload of a local object to the Tahoe-LAFS file store

Download: a download from the Tahoe-LAFS file store to a local object

Pending notification: a local filesystem change that has been detected but not yet processed.

Representing the Magic Folder in Tahoe-LAFS

Unlike the local case where we use inotify or ReadDirectoryChangesW to detect filesystem changes, we have no mechanism to register a monitor for changes to a Tahoe-LAFS directory. Therefore, we must periodically poll for changes.

An important constraint on the solution is Tahoe-LAFS' "write coordination directive", which prohibits concurrent writes by different storage clients to the same mutable object:

> Tahoe does not provide locking of mutable files and directories. If there is more than one simultaneous attempt to change a mutable file or directory, then an UncoordinatedWriteError may result. This might, in rare cases, cause the file or directory contents to be accidentally deleted. The user is expected to ensure

> that there is at most one outstanding write or update request for a given file or directory at a time. One convenient way to accomplish this is to make a different file or directory for each person or process that wants to write.

Since it is a goal to allow multiple users to write to a Magic Folder, if the write coordination directive remains the same as above, then we will not be able to implement the Magic Folder as a single Tahoe-LAFS DMD. In general therefore, we will have multiple DMDs —spread across clients— that together represent the Magic Folder. Each client in a Magic Folder collective polls the other clients' DMDs in order to detect remote changes.

Six possible designs were considered for the representation of subfolders of the Magic Folder:

1. All subfolders written by a given Magic Folder client are collapsed into a single client DMD, containing immutable files. The child name of each file encodes the full subpath of that file relative to the Magic Folder.

2. The DMD tree under a client DMD is a direct copy of the folder tree written by that client to the Magic Folder. Not all subfolders have corresponding DMDs; only those to which that client has written files or child subfolders.

3. The directory tree under a client DMD is a `tahoe backup` structure containing immutable snapshots of the folder tree written by that client to the Magic Folder. As in design 2, only objects written by that client are present.

4. *Each* client DMD contains an eventually consistent mirror of all files and folders written by *any* Magic Folder client. Thus each client must also copy changes made by other Magic Folder clients to its own client DMD.

5. *Each* client DMD contains a `tahoe backup` structure containing immutable snapshots of all files and folders written by *any* Magic Folder client. Thus each client must also create another snapshot in its own client DMD when changes are made by another client. (It can potentially batch changes, subject to latency requirements.)

6. The write coordination problem is solved by implementing two-phase commit. Then, the representation consists of a single DMD tree which is written by all clients.

Here is a summary of advantages and disadvantages of each design:

Key	
++	major advantage
+	minor advantage
–	minor disadvantage
– –	major disadvantage
– – –	showstopper

123456+: All designs have the property that a recursive add-lease operation starting from a *collective directory* containing all of the client DMDs, will find all of the files and directories used in the Magic Folder representation. Therefore the representation is compatible with garbage collection, even when a pre-Magic-Folder client does the lease marking.

123456+: All designs avoid "breaking" pre-Magic-Folder clients that read a directory or file that is part of the representation.

456++: Only these designs allow a readcap to one of the client directories —or one of their subdirectories— to be directly shared with other Tahoe-LAFS clients (not necessarily Magic Folder clients), so that such a client sees all of the contents of the Magic Folder. Note that this was not a requirement of the OTF proposal, although it is useful.

135+: A Magic Folder client has only one mutable Tahoe-LAFS object to monitor per other client. This minimizes communication bandwidth for polling, or alternatively the latency possible for a given polling bandwidth.

1236+: A client does not need to make changes to its own DMD that repeat changes that another Magic Folder client had previously made. This reduces write bandwidth and complexity.

1–: If the Magic Folder has many subfolders, their files will all be collapsed into the same DMD, which could get quite large. In practice a single DMD can easily handle the number of files expected to be written by a client, so this is unlikely to be a significant issue.

123– –: In these designs, the set of files in a Magic Folder is represented as the union of the files in all client DMDs. However, when a file is modified by more than one client, it will be linked from multiple client DMDs. We therefore need a mechanism, such as a version number or a monotonically increasing timestamp, to determine which copy takes priority.

35– –: When a Magic Folder client detects a remote change, it must traverse an immutable directory structure to see what has changed. Completely unchanged subtrees will have the same URI, allowing some of this traversal to be shortcutted.

24– – –: When a Magic Folder client detects a remote change, it must traverse a mutable directory structure to see what has changed. This is more complex and less efficient than traversing an immutable structure, because shortcutting is not possible (each DMD retains the same URI even if a descendant object has changed), and because the structure may change while it is being traversed. Also the traversal needs to be robust against cycles, which can only occur in mutable structures.

45– –: When a change occurs in one Magic Folder client, it will propagate to all the other clients. Each client will therefore see multiple representation changes for a single logical change to the Magic Folder contents, and must suppress the duplicates. This is particularly problematic for design 4 where it interacts with the preceding issue.

4– – –, 5– –: There is the potential for client DMDs to get "out of sync" with each other, potentially for long periods if errors occur. Thus each client must be able to "repair" its client directory (and its subdirectory structure) concurrently with performing its own writes. This is a significant complexity burden and may introduce failure modes that could not otherwise happen.

6– – –: While two-phase commit is a well-established protocol, its application to Tahoe-LAFS requires significant design work, and may still leave some corner cases of the write coordination problem unsolved.

Design Property	Designs Proposed					
advantages	*1*	*2*	*3*	*4*	*5*	*6*
Compatible with garbage collection	+	+	+	+	+	+
Does not break old clients	+	+	+	+	+	+
Allows direct sharing				++	++	++
Efficient use of bandwidth	+		+		+	
No repeated changes	+	+	+			+
disadvantages	*1*	*2*	*3*	*4*	*5*	*6*
Can result in large DMDs	–					
Need version number to determine priority	– –	– –	– –			
Must traverse immutable directory structure			– –		– –	
Must traverse mutable directory structure		– – –		– – –		
Must suppress duplicate representation changes				– –	– –	
"Out of sync" problem				– – –	– –	
Unsolved design problems						– – –

Evaluation of designs

Designs 2 and 3 have no significant advantages over design 1, while requiring higher polling bandwidth and greater complexity due to the need to create subdirectories. These designs were therefore rejected.

Design 4 was rejected due to the out-of-sync problem, which is severe and possibly unsolvable for mutable structures.

For design 5, the out-of-sync problem is still present but possibly solvable. However, design 5 is substantially more complex, less efficient in bandwidth/latency, and less scalable in number of clients and subfolders than design 1. It only gains over design 1 on the ability to share directory readcaps to the Magic Folder (or subfolders), which was not a requirement. It would be possible to implement this feature in future by switching to design 6.

For the time being, however, design 6 was considered out-of-scope for this project.

Therefore, design 1 was chosen. That is:

> All subfolders written by a given Magic Folder client are collapsed into a single client DMD, containing immutable files. The child name of each file encodes the full subpath of that file relative to the Magic Folder.

Each directory entry in a DMD also stores a version number, so that the latest version of a file is well-defined when it has been modified by multiple clients.

To enable representing empty directories, a client that creates a directory should link a corresponding zero-length file in its DMD, at a name that ends with the encoded directory separator character.

We want to enable dynamic configuration of the membership of a Magic Folder collective, without having to reconfigure or restart each client when another client joins. To support this, we have a single collective directory that links to all of the client DMDs, named by their client nicknames. If the collective directory is mutable, then it is possible to change its contents in order to add clients. Note that a client DMD should not be unlinked from the collective directory unless all of its files are first copied to some other client DMD.

A client needs to be able to write to its own DMD, and read from other DMDs. To be consistent with the Principle of Least Authority, each client's reference to its own DMD is a write capability, whereas its reference to the collective directory is a read capability. The latter transitively grants read access to all of the other client DMDs and the files linked from them, as required.

Design and implementation of the user interface for maintaining this DMD structure and configuration will be addressed in Objectives 5 and 6.

During operation, each client will poll for changes on other clients at a predetermined frequency. On each poll, it will reread the collective directory (to allow for added or removed clients), and then read each client DMD linked from it.

"Hidden" files, and files with names matching the patterns used for backup, temporary, and conflicted files, will be ignored, i.e. not synchronized in either direction. A file is hidden if it has a filename beginning with "." (on any platform), or has the hidden or system attribute on Windows.

Conflict Detection and Resolution

The combination of local filesystems and distributed objects is an example of shared state concurrency, which is highly error-prone and can result in race conditions that are complex to analyze. Unfortunately we have no option but to use shared state in this situation.

We call the resulting design issues "dragons" (as in "Here be dragons"), which as a convenient mnemonic we have named after the classical Greek elements Earth, Fire, Air, and Water.

Note: all filenames used in the following sections are examples, and the filename patterns we use in the actual implementation may differ. The actual patterns will probably include timestamps, and for conflicted files, the nickname of the client that last changed the file.

Earth Dragons: Collisions between local filesystem operations and downloads

Write/download collisions

Suppose that Alice's Magic Folder client is about to write a version of `foo` that it has downloaded in response to a remote change.

The criteria for distinguishing overwrites from conflicts are described later in the *Fire Dragons* section. Suppose that the remote change has been initially classified as an overwrite. (As we will see, it may be reclassified in some circumstances.)

Note that writing a file that does not already have an entry in the *magic folder db* is initially classed as an overwrite.

A *write/download collision* occurs when another program writes to `foo` in the local filesystem, concurrently with the new version being written by the Magic Folder client. We need to ensure that this does not cause data loss, as far as possible.

An important constraint on the design is that on Windows, it is not possible to rename a file to the same name as an existing file in that directory. Also, on Windows it may not be possible to delete or rename a file that has been opened by another process (depending on the sharing flags specified by that process). Therefore we need to consider carefully how to handle failure conditions.

In our proposed design, Alice's Magic Folder client follows this procedure for an overwrite in response to a remote change:

1. Write a temporary file, say `.foo.tmp`.
2. Use the procedure described in the *Fire Dragons_* section to obtain an initial classification as an overwrite or a conflict. (This takes as input the `last_downloaded_uri` field from the directory entry of the changed `foo`.)
3. Set the `mtime` of the replacement file to be at least *T* seconds before the current local time. Stat the replacement file to obtain its `mtime` and `ctime` as stored in the local filesystem, and update the file's last-seen statinfo in the magic folder db with this information. (Note that the retrieved `mtime` may differ from the one that was set due to rounding.)
4. Perform a ''file replacement" operation (explained below) with backup filename `foo.backup`, replaced file `foo`, and replacement file `.foo.tmp`. If any step of this operation fails, reclassify as a conflict and stop.

To reclassify as a conflict, attempt to rename `.foo.tmp` to `foo.conflicted`, suppressing errors.

The implementation of file replacement differs between Unix and Windows. On Unix, it can be implemented as follows:

- 4a. Stat the replaced path, and set the permissions of the replacement file to be the same as the replaced file, bitwise-or'd with octal 600 (`rw-------`). If the replaced file does not exist, set the permissions according to the user's umask. If there is a directory at the replaced path, fail.
- 4b. Attempt to move the replaced file (`foo`) to the backup filename (`foo.backup`). If an `ENOENT` error occurs because the replaced file does not exist, ignore this error and continue with steps 4c and 4d.
- 4c. Attempt to create a hard link at the replaced filename (`foo`) pointing to the replacement file (`.foo.tmp`).
- 4d. Attempt to unlink the replacement file (`.foo.tmp`), suppressing errors.

Note that, if there is no conflict, the entry for `foo` recorded in the *magic folder db* will reflect the `mtime` set in step 3. The move operation in step 4b will cause a `MOVED_FROM` event for `foo`, and the link operation in step 4c will cause an `IN_CREATE` event for `foo`. However, these events will not trigger an upload, because they are guaranteed to be processed only after the file replacement has finished, at which point the last-seen statinfo recorded in the database entry will exactly match the metadata for the file's inode on disk. (The two hard links — `foo` and, while it still exists, `.foo.tmp` — share the same inode and therefore the same metadata.)

On Windows, file replacement can be implemented by a call to the ReplaceFileW API (with the `REPLACEFILE_IGNORE_MERGE_ERRORS` flag). If an error occurs because the replaced file does not exist, then we ignore this error and attempt to move the replacement file to the replaced file.

Similar to the Unix case, the ReplaceFileW operation will cause one or more change notifications for `foo`. The replaced `foo` has the same `mtime` as the replacement file, and so any such notification(s) will not trigger an unwanted upload.

To determine whether this procedure adequately protects against data loss, we need to consider what happens if another process attempts to update `foo`, for example by renaming `foo.other` to `foo`. This requires us to analyze all possible interleavings between the operations performed by the Magic Folder client and the other process. (Note that atomic operations on a directory are totally ordered.) The set of possible interleavings differs between Windows and Unix.

On Unix, for the case where the replaced file already exists, we have:

- Interleaving A: the other process' rename precedes our rename in step 4b, and we get an `IN_MOVED_TO` event for its rename by step 2. Then we reclassify as a conflict; its changes end up at `foo` and ours end up at `foo.conflicted`. This avoids data loss.
- Interleaving B: its rename precedes ours in step 4b, and we do not get an event for its rename by step 2. Its changes end up at `foo.backup`, and ours end up at `foo` after being linked there in step 4c. This avoids data loss.
- Interleaving C: its rename happens between our rename in step 4b, and our link operation in step 4c of the file replacement. The latter fails with an `EEXIST` error because `foo` already exists. We reclassify as a conflict; the old version ends up at `foo.backup`, the other process' changes end up at `foo`, and ours at `foo.conflicted`. This avoids data loss.
- Interleaving D: its rename happens after our link in step 4c, and causes an `IN_MOVED_TO` event for `foo`. Its rename also changes the `mtime` for `foo` so that it is different from the `mtime` calculated in step 3, and therefore different from the metadata recorded for `foo` in the magic folder db. (Assuming no system clock changes, its rename will set an `mtime` timestamp corresponding to a time after step 4c, which is after the timestamp *T* seconds before step 4a, provided that *T* seconds is sufficiently greater than the timestamp granularity.) Therefore, an upload will be triggered for `foo` after its change, which is correct and avoids data loss.

If the replaced file did not already exist, an `ENOENT` error occurs at step 4b, and we continue with steps 4c and 4d. The other process' rename races with our link operation in step 4c. If the other process wins the race then the effect is similar to Interleaving C, and if we win the race this it is similar to Interleaving D. Either case avoids data loss.

On Windows, the internal implementation of ReplaceFileW is similar to what we have described above for Unix; it works like this:

- 4a. Copy metadata (which does not include `mtime`) from the replaced file (`foo`) to the replacement file (`.foo.tmp`).
- 4b. Attempt to move the replaced file (`foo`) onto the backup filename (`foo.backup`), deleting the latter if it already exists.
- 4c. Attempt to move the replacement file (`.foo.tmp`) to the replaced filename (`foo`); fail if the destination already exists.

Notice that this is essentially the same as the algorithm we use for Unix, but steps 4c and 4d on Unix are combined into a single step 4c. (If there is a failure at steps 4c after step 4b has completed, the ReplaceFileW call will fail with return code `ERROR_UNABLE_TO_MOVE_REPLACEMENT_2`. However, it is still preferable to use this API over two MoveFileExW calls, because it retains the attributes and ACLs of `foo` where possible. Also note that if the ReplaceFileW call fails with `ERROR_FILE_NOT_FOUND` because the replaced file does not exist, then the replacment operation ignores this error and continues with the equivalent of step 4c, as on Unix.)

However, on Windows the other application will not be able to directly rename `foo.other` onto `foo` (which would fail because the destination already exists); it will have to rename or delete `foo` first. Without loss of generality, let's say `foo` is deleted. This complicates the interleaving analysis, because we have two operations done by the other process interleaving with three done by the magic folder process (rather than one operation interleaving with four as on Unix).

So on Windows, for the case where the replaced file already exists, we have:

- Interleaving A: the other process' deletion of `foo` and its rename of `foo.other` to `foo` both precede our rename in step 4b. We get an event corresponding to its rename by step 2. Then we reclassify as a conflict; its changes end up at `foo` and ours end up at `foo.conflicted`. This avoids data loss.
- Interleaving B: the other process' deletion of `foo` and its rename of `foo.other` to `foo` both precede our rename in step 4b. We do not get an event for its rename by step 2. Its changes end up at `foo.backup`, and ours end up at `foo` after being moved there in step 4c. This avoids data loss.

- Interleaving C: the other process' deletion of `foo` precedes our rename of `foo` to `foo.backup` done by ReplaceFileW, but its rename of `foo.other` to `foo` does not, so we get an `ERROR_FILE_NOT_FOUND` error from ReplaceFileW indicating that the replaced file does not exist. We ignore this error and attempt to move `foo.tmp` to `foo`, racing with the other process which is attempting to move `foo.other` to `foo`. If we win the race, then our changes end up at `foo`, and the other process' move fails. If the other process wins the race, then its changes end up at `foo`, our move fails, and we reclassify as a conflict, so that our changes end up at `foo.conflicted`. Either possibility avoids data loss.
- Interleaving D: the other process' deletion and/or rename happen during the call to ReplaceFileW, causing the latter to fail. There are two subcases:
  - if the error is `ERROR_UNABLE_TO_MOVE_REPLACEMENT_2`, then `foo` is renamed to `foo.backup` and `.foo.tmp` remains at its original name after the call.
  - for all other errors, `foo` and `.foo.tmp` both remain at their original names after the call.

  In both subcases, we reclassify as a conflict and rename `.foo.tmp` to `foo.conflicted`. This avoids data loss.
- Interleaving E: the other process' deletion of `foo` and attempt to rename `foo.other` to `foo` both happen after all internal operations of ReplaceFileW have completed. This causes deletion and rename events for `foo` (which will in practice be merged due to the pending delay, although we don't rely on that for correctness). The rename also changes the `mtime` for `foo` so that it is different from the `mtime` calculated in step 3, and therefore different from the metadata recorded for `foo` in the magic folder db. (Assuming no system clock changes, its rename will set an `mtime` timestamp corresponding to a time after the internal operations of ReplaceFileW have completed, which is after the timestamp *T* seconds before ReplaceFileW is called, provided that *T* seconds is sufficiently greater than the timestamp granularity.) Therefore, an upload will be triggered for `foo` after its change, which is correct and avoids data loss.

If the replaced file did not already exist, we get an `ERROR_FILE_NOT_FOUND` error from ReplaceFileW, and attempt to move `foo.tmp` to `foo`. This is similar to Interleaving C, and either possibility for the resulting race avoids data loss.

We also need to consider what happens if another process opens `foo` and writes to it directly, rather than renaming another file onto it:

- On Unix, open file handles refer to inodes, not paths. If the other process opens `foo` before it has been renamed to `foo.backup`, and then closes the file, changes will have been written to the file at the same inode, even if that inode is now linked at `foo.backup`. This avoids data loss.
- On Windows, we have two subcases, depending on whether the sharing flags specified by the other process when it opened its file handle included `FILE_SHARE_DELETE`. (This flag covers both deletion and rename operations.)
  1. If the sharing flags *do not* allow deletion/renaming, the ReplaceFileW operation will fail without renaming `foo`. In this case we will end up with `foo` changed by the other process, and the downloaded file still in `foo.tmp`. This avoids data loss.
  2. If the sharing flags *do* allow deletion/renaming, then data loss or corruption may occur. This is unavoidable and can be attributed to other process making a poor choice of sharing flags (either explicitly if it used CreateFile, or via whichever higher-level API it used).

Note that it is possible that another process tries to open the file between steps 4b and 4c (or 4b and 4c on Windows). In this case the open will fail because `foo` does not exist. Nevertheless, no data will be lost, and in many cases the user will be able to retry the operation.

Above we only described the case where the download was initially classified as an overwrite. If it was classed as a conflict, the procedure is the same except that we choose a unique filename for the conflicted file (say, `foo.conflicted_unique`). We write the new contents to `.foo.tmp` and then rename it to `foo.conflicted_unique` in such a way that the rename will fail if the destination already exists. (On Windows this is a simple rename; on Unix it can be implemented as a link operation followed by an unlink, similar to steps 4c and 4d above.) If this fails because another process wrote `foo.conflicted_unique` after we chose the filename, then we retry with a different filename.

Read/download collisions

A *read/download collision* occurs when another program reads from `foo` in the local filesystem, concurrently with the new version being written by the Magic Folder client. We want to ensure that any successful attempt to read the file by the other program obtains a consistent view of its contents.

On Unix, the above procedure for writing downloads is sufficient to achieve this. There are three cases:

- A. The other process opens `foo` for reading before it is renamed to `foo.backup`. Then the file handle will continue to refer to the old file across the rename, and the other process will read the old contents.
- B. The other process attempts to open `foo` after it has been renamed to `foo.backup`, and before it is linked in step c. The open call fails, which is acceptable.
- C. The other process opens `foo` after it has been linked to the new file. Then it will read the new contents.

On Windows, the analysis is very similar, but case A needs to be split into two subcases, depending on the sharing mode the other process uses when opening the file for reading:

- A. The other process opens `foo` before the Magic Folder client's attempt to rename `foo` to `foo.backup` (as part of the implementation of ReplaceFileW). The subcases are:
  1. The other process uses sharing flags that deny deletion and renames. The ReplaceFileW call fails, and the download is reclassified as a conflict. The downloaded file ends up at `foo.conflicted`, which is correct.
  2. The other process uses sharing flags that allow deletion and renames. The ReplaceFileW call succeeds, and the other process reads inconsistent data. This can be attributed to a poor choice of sharing flags by the other process.
- B. The other process attempts to open `foo` at the point during the ReplaceFileW call where it does not exist. The open call fails, which is acceptable.
- C. The other process opens `foo` after it has been linked to the new file. Then it will read the new contents.

For both write/download and read/download collisions, we have considered only interleavings with a single other process, and only the most common possibilities for the other process' interaction with the file. If multiple other processes are involved, or if a process performs operations other than those considered, then we cannot say much about the outcome in general; however, we believe that such cases will be much less common.

Fire Dragons: Distinguishing conflicts from overwrites

When synchronizing a file that has changed remotely, the Magic Folder client needs to distinguish between overwrites, in which the remote side was aware of your most recent version (if any) and overwrote it with a new version, and conflicts, in which the remote side was unaware of your most recent version when it published its new version. Those two cases have to be handled differently — the latter needs to be raised to the user as an issue the user will have to resolve and the former must not bother the user.

For example, suppose that Alice's Magic Folder client sees a change to `foo` in Bob's DMD. If the version it downloads from Bob's DMD is "based on" the version currently in Alice's local filesystem at the time Alice's client attempts to write the downloaded file –or if there is no existing version in Alice's local filesystem at that time– then it is an overwrite. Otherwise it is initially classified as a conflict.

This initial classification is used by the procedure for writing a file described in the *Earth Dragons* section above. As explained in that section, we may reclassify an overwrite as a conflict if an error occurs during the write procedure.

In order to implement this policy, we need to specify how the "based on" relation between file versions is recorded and updated.

We propose to record this information:

- in the *magic folder db*, for local files;
- in the Tahoe-LAFS directory metadata, for files stored in the Magic Folder.

In the magic folder db we will add a *last-downloaded record*, consisting of `last_downloaded_uri` and `last_downloaded_timestamp` fields, for each path stored in the database. Whenever a Magic Folder client downloads a file, it stores the downloaded version's URI and the current local timestamp in this record. Since only immutable files are used, the URI will be an immutable file URI, which is deterministically and uniquely derived from the file contents and the Tahoe-LAFS node's convergence secret.

(Note that the last-downloaded record is updated regardless of whether the download is an overwrite or a conflict. The rationale for this to avoid "conflict loops" between clients, where every new version after the first conflict would be considered as another conflict.)

Later, in response to a local filesystem change at a given path, the Magic Folder client reads the last-downloaded record associated with that path (if any) from the database and then uploads the current file. When it links the uploaded file into its client DMD, it includes the `last_downloaded_uri` field in the metadata of the directory entry, overwriting any existing field of that name. If there was no last-downloaded record associated with the path, this field is omitted.

Note that `last_downloaded_uri` field does *not* record the URI of the uploaded file (which would be redundant); it records the URI of the last download before the local change that caused the upload. The field will be absent if the file has never been downloaded by this client (i.e. if it was created on this client and no change by any other client has been detected).

A possible refinement also takes into account the `last_downloaded_timestamp` field from the magic folder db, and compares it to the timestamp of the change that caused the upload (which should be later, assuming no system clock changes). If the duration between these timestamps is very short, then we are uncertain about whether the process on Bob's system that wrote the local file could have taken into account the last download. We can use this information to be conservative about treating changes as conflicts. So, if the duration is less than a configured threshold, we omit the `last_downloaded_uri` field from the metadata. This will have the effect of making other clients treat this change as a conflict whenever they already have a copy of the file.

Conflict/overwrite decision algorithm

Now we are ready to describe the algorithm for determining whether a download for the file `foo` is an overwrite or a conflict (refining step 2 of the procedure from the *Earth Dragons* section).

Let `last_downloaded_uri` be the field of that name obtained from the directory entry metadata for `foo` in Bob's DMD (this field may be absent). Then the algorithm is:

- 2a. Attempt to "stat" `foo` to get its *current statinfo* (size in bytes, `mtime`, and `ctime`). If Alice has no local copy of `foo`, classify as an overwrite.
- 2b. Read the following information for the path `foo` from the local magic folder db:
  - the *last-seen statinfo*, if any (this is the size in bytes, `mtime`, and `ctime` stored in the `local_files` table when the file was last uploaded);
  - the `last_uploaded_uri` field of the `local_files` table for this file, which is the URI under which the file was last uploaded.
- 2c. If any of the following are true, then classify as a conflict:

- i. there are pending notifications of changes to `foo`;
- ii. the last-seen statinfo is either absent (i.e. there is no entry in the database for this path), or different from the current statinfo;
- iii. either `last_downloaded_uri` or `last_uploaded_uri` (or both) are absent, or they are different.

Otherwise, classify as an overwrite.

Air Dragons: Collisions between local writes and uploads

Short of filesystem-specific features on Unix or the shadow copy service on Windows (which is per-volume and therefore difficult to use in this context), there is no way to *read* the whole contents of a file atomically. Therefore, when we read a file in order to upload it, we may read an inconsistent version if it was also being written locally.

A well-behaved application can avoid this problem for its writes:

- On Unix, if another process modifies a file by renaming a temporary file onto it, then we will consistently read either the old contents or the new contents.
- On Windows, if the other process uses sharing flags to deny reads while it is writing a file, then we will consistently read either the old contents or the new contents, unless a sharing error occurs. In the case of a sharing error we should retry later, up to a maximum number of retries.

In the case of a not-so-well-behaved application writing to a file at the same time we read from it, the magic folder will still be eventually consistent, but inconsistent versions may be visible to other users' clients.

In Objective 2 we implemented a delay, called the *pending delay*, after the notification of a filesystem change and before the file is read in order to upload it (Tahoe-LAFS ticket #1440). If another change notification occurs within the pending delay time, the delay is restarted. This helps to some extent because it means that if files are written more quickly than the pending delay and less frequently than the pending delay, we shouldn't encounter this inconsistency.

The likelihood of inconsistency could be further reduced, even for writes by not-so-well-behaved applications, by delaying the actual upload for a further period —called the *stability delay*— after the file has finished being read. If a notification occurs between the end of the pending delay and the end of the stability delay, then the read would be aborted and the notification requeued.

This would have the effect of ensuring that no write notifications have been received for the file during a time window that brackets the period when it was being read, with margin before and after this period defined by the pending and stability delays. The delays are intended to account for asynchronous notification of events, and caching in the filesystem.

Note however that we cannot guarantee that the delays will be long enough to prevent inconsistency in any particular case. Also, the stability delay would potentially affect performance significantly because (unlike the pending delay) it is not overlapped when there are multiple files on the upload queue. This performance impact could be mitigated by uploading files in parallel where possible (Tahoe-LAFS ticket #1459).

We have not yet decided whether to implement the stability delay, and it is not planned to be implemented for the OTF objective 4 milestone. Ticket #2431 has been opened to track this idea.

Note that the situation of both a local process and the Magic Folder client reading a file at the same time cannot cause any inconsistency.

Water Dragons: Handling deletion and renames

Deletion of a file

When a file is deleted from the filesystem of a Magic Folder client, the most intuitive behavior is for it also to be deleted under that name from other clients. To avoid data loss, the other clients should actually rename their copies to a backup filename.

It would not be sufficient for a Magic Folder client that deletes a file to implement this simply by removing the directory entry from its DMD. Indeed, the entry may not exist in the client's DMD if it has never previously changed the file.

Instead, the client links a zero-length file into its DMD and sets `deleted: true` in the directory entry metadata. Other clients take this as a signal to rename their copies to the backup filename.

Note that the entry for this zero-length file has a version number as usual, and later versions may restore the file.

When the downloader deletes a file (or renames it to a filename ending in `.backup`) in response to a remote change, a local filesystem notification will occur, and we must make sure that this is not treated as a local change. To do this we have the downloader set the `size` field in the magic folder db to `None` (SQL NULL) just before deleting the file, and suppress notifications for which the local file does not exist, and the recorded `size` field is `None`.

When a Magic Folder client restarts, we can detect files that had been downloaded but were deleted while it was not running, because their paths will have last-downloaded records in the magic folder db with a `size` other than `None`, and without any corresponding local file.

Deletion of a directory

Local filesystems (unlike a Tahoe-LAFS filesystem) normally cannot unlink a directory that has any remaining children. Therefore a Magic Folder client cannot delete local copies of directories in general, because they will typically contain backup files. This must be done manually on each client if desired.

Nevertheless, a Magic Folder client that deletes a directory should set `deleted: true` on the metadata entry for the corresponding zero-length file. This avoids the directory being recreated after it has been manually deleted from a client.

Renaming

It is sufficient to handle renaming of a file by treating it as a deletion and an addition under the new name.

This also applies to directories, although users may find the resulting behavior unintuitive: all of the files under the old name will be renamed to backup filenames, and a new directory structure created under the new name. We believe this is the best that can be done without imposing unreasonable implementation complexity.

Summary

This completes the design of remote-to-local synchronization. We realize that it may seem very complicated. Anecdotally, proprietary filesystem synchronization designs we are aware of, such as Dropbox, are said to incur similar or greater design complexity.

1.6.3 Magic Folder user interface design

Scope

In this Objective we will design a user interface to allow users to conveniently and securely indicate which folders on some devices should be "magically" linked to which folders on other devices.

This is a critical usability and security issue for which there is no known perfect solution, but which we believe is amenable to a "good enough" trade-off solution. This document explains the design and justifies its trade-offs in terms of security, usability, and time-to-market.

Tickets on the Tahoe-LAFS trac with the otf-magic-folder-objective6 keyword are within the scope of the user interface design.

Glossary

Object: a file or directory

DMD: distributed mutable directory

Folder: an abstract directory that is synchronized between clients. (A folder is not the same as the directory corresponding to it on any particular client, nor is it the same as a DMD.)

Collective: the set of clients subscribed to a given Magic Folder.

Diminishing: the process of deriving, from an existing capability, another capability that gives less authority (for example, deriving a read cap from a read/write cap).

Design Constraints

The design of the Tahoe-side representation of a Magic Folder, and the polling mechanism that the Magic Folder clients will use to detect remote changes was discussed in *remote-to-local-sync*, and we will not revisit that here. The assumption made by that design was that each client would be configured with the following information:

- a write cap to its own *client DMD*.
- a read cap to a *collective directory*.

The collective directory contains links to each client DMD named by the corresponding client's nickname.

This design was chosen to allow straightforward addition of clients without requiring each existing client to change its configuration.

Note that each client in a Magic Folder collective has the authority to add, modify or delete any object within the Magic Folder. It is also able to control to some extent whether its writes will be treated by another client as overwrites or as conflicts. However, there is still a reliability benefit to preventing a client from accidentally modifying another client's DMD, or from accidentally modifying the collective directory in a way that would lose data. This motivates ensuring that each client only has access to the caps above, rather than, say, every client having a write cap to the collective directory.

Another important design constraint is that we cannot violate the write coordination directive; that is, we cannot write to the same mutable directory from multiple clients, even during the setup phase when adding a client.

Within these constraints, for usability we want to minimize the number of steps required to configure a Magic Folder collective.

Proposed Design

Three `tahoe` subcommands are added:

```
tahoe magic-folder create MAGIC: [MY_NICKNAME LOCAL_DIR]

  Create an empty Magic Folder. The MAGIC: local alias is set
  to a write cap which can be used to refer to this Magic Folder
```

(continues on next page)

(continued from previous page)

```
  in future ``tahoe magic-folder invite`` commands.

  If MY_NICKNAME and LOCAL_DIR are given, the current client
  immediately joins the newly created Magic Folder with that
  nickname and local directory.


tahoe magic-folder invite MAGIC: THEIR_NICKNAME

  Print an "invitation" that can be used to invite another
  client to join a Magic Folder, with the given nickname.

  The invitation must be sent to the user of the other client
  over a secure channel (e.g. PGP email, OTR, or ssh).

  This command will normally be run by the same client that
  created the Magic Folder. However, it may be run by a
  different client if the ``MAGIC:`` alias is copied to
  the ``private/aliases`` file of that other client, or if
  ``MAGIC:`` is replaced by the write cap to which it points.


tahoe magic-folder join INVITATION LOCAL_DIR

  Accept an invitation created by ``tahoe magic-folder invite``.
  The current client joins the specified Magic Folder, which will
  appear in the local filesystem at the given directory.
```

There are no commands to remove a client or to revoke an invitation, although those are possible features that could be added in future. (When removing a client, it is necessary to copy each file it added to some other client's DMD, if it is the most recent version of that file.)

Implementation

For "`tahoe magic-folder create MAGIC: [MY_NICKNAME LOCAL_DIR]`":

1. Run "`tahoe create-alias MAGIC:`".
2. If `MY_NICKNAME` and `LOCAL_DIR` are given, do the equivalent of:

```
INVITATION=`tahoe invite-magic-folder MAGIC: MY_NICKNAME`
tahoe join-magic-folder INVITATION LOCAL_DIR
```

For "`tahoe magic-folder invite COLLECTIVE_WRITECAP NICKNAME`":

(`COLLECTIVE_WRITECAP` can, as a special case, be an alias such as `MAGIC:`.)

1. Create an empty client DMD. Let its write URI be `CLIENT_WRITECAP`.
2. Diminish `CLIENT_WRITECAP` to `CLIENT_READCAP`, and diminish `COLLECTIVE_WRITECAP` to `COLLECTIVE_READCAP`.
3. Run "`tahoe ln CLIENT_READCAP COLLECTIVE_WRITECAP/NICKNAME`".
4. Print "`COLLECTIVE_READCAP+CLIENT_WRITECAP`" as the invitation, accompanied by instructions on how to accept the invitation and the need to send it over a secure channel.

For "`tahoe magic-folder join INVITATION LOCAL_DIR`":

1. Parse `INVITATION` as `COLLECTIVE_READCAP+CLIENT_WRITECAP`.
2. Write `CLIENT_WRITECAP` to the file `magic_folder_dircap` under the client's `private` directory.
3. Write `COLLECTIVE_READCAP` to the file `collective_dircap` under the client's `private` directory.
4. Edit the client's `tahoe.cfg` to set `[magic_folder] enabled = True` and `[magic_folder] local.directory = LOCAL_DIR`.

Discussion

The proposed design has a minor violation of the Principle of Least Authority in order to reduce the number of steps needed. The invoker of "`tahoe magic-folder invite`" creates the client DMD on behalf of the invited client, and could retain its write cap (which is part of the invitation).

A possible alternative design would be for the invited client to create its own client DMD, and send it back to the inviter to be linked into the collective directory. However this would require another secure communication and another command invocation per client. Given that, as mentioned earlier, each client in a Magic Folder collective already has the authority to add, modify or delete any object within the Magic Folder, we considered the potential security/reliability improvement here not to be worth the loss of usability.

We also considered a design where each client had write access to the collective directory. This would arguably be a more serious violation of the Principle of Least Authority than the one above (because all clients would have excess authority rather than just the inviter). In any case, it was not clear how to make such a design satisfy the write coordination directive, because the collective directory would have needed to be written to by multiple clients.

The reliance on a secure channel to send the invitation to its intended recipient is not ideal, since it may involve additional software such as clients for PGP, OTR, ssh etc. However, we believe that this complexity is necessary rather than incidental, because there must be some way to distinguish the intended recipient from potential attackers who would try to become members of the Magic Folder collective without authorization. By making use of existing channels that have likely already been set up by security-conscious users, we avoid reinventing the wheel or imposing substantial extra implementation costs.

The length of an invitation will be approximately the combined length of a Tahoe-LAFS read cap and write cap. This is several lines long, but still short enough to be cut-and-pasted successfully if care is taken. Errors in copying the invitation can be detected since Tahoe-LAFS cap URIs are self-authenticating.

The implementation of the `tahoe` subcommands is straightforward and raises no further difficult design issues.

1.6.4 Multi-party Conflict Detection

The current Magic-Folder remote conflict detection design does not properly detect remote conflicts for groups of three or more parties. This design is specified in the "Fire Dragon" section of this document: https://github.com/tahoe-lafs/tahoe-lafs/blob/2551.wip.2/docs/proposed/magic-folder/remote-to-local-sync.rst#fire-dragons-distinguishing-conflicts-from-overwrites

This Tahoe-LAFS trac ticket comment outlines a scenario with three parties in which a remote conflict is falsely detected:

1.6.5 Summary and definitions

Abstract file: a file being shared by a Magic Folder.

Local file: a file in a client's local filesystem corresponding to an abstract file.

Relative path: the path of an abstract or local file relative to the Magic Folder root.

Version: a snapshot of an abstract file, with associated metadata, that is uploaded by a Magic Folder client.

A version is associated with the file's relative path, its contents, and mtime and ctime timestamps. Versions also have a unique identity.

Follows relation: * If and only if a change to a client's local file at relative path F that results in an upload of version V', was made when the client already had version V of that file, then we say that V' directly follows V. * The follows relation is the irreflexive transitive closure of the "directly follows" relation.

The follows relation is transitive and acyclic, and therefore defines a DAG called the Version DAG. Different abstract files correspond to disconnected sets of nodes in the Version DAG (in other words there are no "follows" relations between different files).

The DAG is only ever extended, not mutated.

The desired behaviour for initially classifying overwrites and conflicts is as follows:

- if a client Bob currently has version V of a file at relative path F, and it sees a new version V' of that file in another client Alice's DMD, such that V' follows V, then the write of the new version is initially an overwrite and should be to the same filename.
- if, in the same situation, V' does not follow V, then the write of the new version should be classified as a conflict.

The existing *Magic Folder design for remote-to-local sync* document defines when an initial overwrite should be reclassified as a conflict.

The above definitions completely specify the desired solution of the false conflict behaviour described in the ticket comment. However, they do not give a concrete algorithm to compute the follows relation, or a representation in the Tahoe-LAFS file store of the metadata needed to compute it.

We will consider two alternative designs, proposed by Leif Ryge and Zooko Wilcox-O'Hearn, that aim to fill this gap.

1.6.6 Leif's Proposal: Magic-Folder "single-file" snapshot design

Abstract

We propose a relatively simple modification to the initial Magic Folder design which adds merkle DAGs of immutable historical snapshots for each file. The full history does not necessarily need to be retained, and the choice of how much history to retain can potentially be made on a per-file basis.

Motivation:

no SPOFs, no admins

Additionally, the initial design had two cases of excess authority:

1. The magic folder administrator (inviter) has everyone's write-caps and is thus essentially "root"
2. Each client shares ambient authority and can delete anything or everything and (assuming there is not a conflict) the data will be deleted from all clients. So, each client is effectively "root" too.

Thus, while it is useful for file synchronization, the initial design is a much less safe place to store data than in a single mutable tahoe directory (because more client computers have the possibility to delete it).

Glossary

- merkle DAG: like a merkle tree but with multiple roots, and with each node potentially having multiple parents
- magic folder: a logical directory that can be synchronized between many clients (devices, users, ...) using a Tahoe-LAFS storage grid

- client: a Magic-Folder-enabled Tahoe-LAFS client instance that has access to a magic folder
- DMD: "distributed mutable directory", a physical Tahoe-LAFS mutable directory. Each client has the write cap to their own DMD, and read caps to all other client's DMDs (as in the original Magic Folder design).
- snapshot: a reference to a version of a file; represented as an immutable directory containing an entry called "content" (pointing to the immutable file containing the file's contents), and an entry called "parent0" (pointing to a parent snapshot), and optionally parent1 through parentN pointing at other parents. The Magic Folder snapshot object is conceptually very similar to a git commit object, except for that it is created automatically and it records the history of an individual file rather than an entire repository. Also, commits do not need to have authors (although an author field could be easily added later).
- deletion snapshot: immutable directory containing no content entry (only one or more parents)
- capability: a Tahoe-LAFS diminishable cryptographic capability
- cap: short for capability
- conflict: the situation when another client's current snapshot for a file is different than our current snapshot, and is not a descendant of ours.
- overwrite: the situation when another client's current snapshot for a file is a (not necessarily direct) descendant of our current snapshot.

Overview

This new design will track the history of each file using "snapshots" which are created at each upload. Each snapshot will specify one or more parent snapshots, forming a directed acyclic graph. A Magic-Folder user's DMD uses a flattened directory hierarchy naming scheme, as in the original design. But, instead of pointing directly at file contents, each file name will link to that user's latest snapshot for that file.

Inside the dmd there will also be an immutable directory containing the client's subscriptions (read-caps to other clients' dmds).

Clients periodically poll each other's DMDs. When they see the current snapshot for a file is different than their own current snapshot for that file, they immediately begin downloading its contents and then walk backwards through the DAG from the new snapshot until they find their own snapshot or a common ancestor.

For the common ancestor search to be efficient, the client will need to keep a local store (in the magic folder db) of all of the snapshots (but not their contents) between the oldest current snapshot of any of their subscriptions and their own current snapshot. See "local cache purging policy" below for more details.

If the new snapshot is a descendant of the client's existing snapshot, then this update is an "overwrite" - like a git fast-forward. So, when the download of the new file completes it can overwrite the existing local file with the new contents and update its dmd to point at the new snapshot.

If the new snapshot is not a descendant of the client's current snapshot, then the update is a conflict. The new file is downloaded and named $filename.conflict-$user1,$user2 (including a list of other subscriptions who have that version as their current version).

Changes to the local .conflict- file are not tracked. When that file disappears (either by deletion, or being renamed) a new snapshot for the conflicting file is created which has two parents - the client's snapshot prior to the conflict, and the new conflicting snapshot. If multiple .conflict files are deleted or renamed in a short period of time, a single conflict-resolving snapshot with more than two parents can be created.

! I think this behavior will confuse users.

Tahoe-LAFS snapshot objects

These Tahoe-LAFS snapshot objects only track the history of a single file, not a directory hierarchy. Snapshot objects contain only two field types: - `Content`: an immutable capability of the file contents (omitted if deletion snapshot) - `Parent0..N`: immutable capabilities representing parent snapshots

Therefore in this system an interesting side effect of this Tahoe snapshot object is that there is no snapshot author. The only notion of an identity in the Magic-Folder system is the write capability of the user's DMD.

The snapshot object is an immutable directory which looks like this: content -> immutable cap to file content parent0 -> immutable cap to a parent snapshot object parent1..N -> more parent snapshots

Snapshot Author Identity

Snapshot identity might become an important feature so that bad actors can be recognized and other clients can stop "subscribing" to (polling for) updates from them.

Perhaps snapshots could be signed by the user's Magic-Folder write key for this purpose? Probably a bad idea to reuse the write-cap key for this. Better to introduce ed25519 identity keys which can (optionally) sign snapshot contents and store the signature as another member of the immutable directory.

Conflict Resolution

detection of conflicts

A Magic-Folder client updates a given file's current snapshot link to a snapshot which is a descendent of the previous snapshot. For a given file, let's say "file1", Alice can detect that Bob's DMD has a "file1" that links to a snapshot which conflicts. Two snapshots conflict if one is not an ancestor of the other.

a possible UI for resolving conflicts

If Alice links a conflicting snapshot object for a file named "file1", Bob and Carole will see a file in their Magic-Folder called "file1.conflicted.Alice". Alice conversely will see an additional file called "file1.conflicted.previous". If Alice wishes to resolve the conflict with her new version of the file then she simply deletes the file called "file1.conflicted.previous". If she wants to choose the other version then she moves it into place:

> mv file1.conflicted.previous file1

This scheme works for N number of conflicts. Bob for instance could choose the same resolution for the conflict, like this:

> mv file1.Alice file1

Deletion propagation and eventual Garbage Collection

When a user deletes a file, this is represented by a link from their DMD file object to a deletion snapshot. Eventually all users will link this deletion snapshot into their DMD. When all users have the link then they locally cache the deletion snapshot and remove the link to that file in their DMD. Deletions can of course be undeleted; this means creating a new snapshot object that specifies itself a descent of the deletion snapshot.

Clients periodically renew leases to all capabilities recursively linked to in their DMD. Files which are unlinked by ALL the users of a given Magic-Folder will eventually be garbage collected.

Lease expirey duration must be tuned properly by storage servers such that Garbage Collection does not occur too frequently.

Performance Considerations

local changes

Our old scheme requires two remote Tahoe-LAFS operations per local file modification: 1. upload new file contents (as an immutable file) 2. modify mutable directory (DMD) to link to the immutable file cap

Our new scheme requires three remote operations: 1. upload new file contents (as in immutable file) 2. upload immutable directory representing Tahoe-LAFS snapshot object 3. modify mutable directory (DMD) to link to the immutable snapshot object

remote changes

Our old scheme requires one remote Tahoe-LAFS operation per remote file modification (not counting the polling of the dmd): 1. Download new file content

Our new scheme requires a minimum of two remote operations (not counting the polling of the dmd) for conflicting downloads, or three remote operations for overwrite downloads: 1. Download new snapshot object 2. Download the content it points to 3. If the download is an overwrite, modify the DMD to indicate that the downloaded version is their current version.

If the new snapshot is not a direct descendant of our current snapshot or the other party's previous snapshot we saw, we will also need to download more snapshots to determine if it is a conflict or an overwrite. However, those can be done in parallel with the content download since we will need to download the content in either case.

While the old scheme is obviously more efficient, we think that the properties provided by the new scheme make it worth the additional cost.

Physical updates to the DMD overiouslly need to be serialized, so multiple logical updates should be combined when an update is already in progress.

conflict detection and local caching

Local caching of snapshots is important for performance. We refer to the client's local snapshot cache as the `magic-folder db`.

Conflict detection can be expensive because it may require the client to download many snapshots from the other user's DMD in order to try and find it's own current snapshot or a descendent. The cost of scanning the remote DMDs should not be very high unless the client conducting the scan has lots of history to download because of being offline for a long time while many new snapshots were distributed.

local cache purging policy

The client's current snapshot for each file should be cached at all times. When all clients' views of a file are synchronized (they all have the same snapshot for that file), no ancestry for that file needs to be cached. When clients' views of a file are *not* synchronized, the most recent common ancestor of all clients' snapshots must be kept cached, as must all intermediate snapshots.

Local Merge Property

Bob can in fact, set a pre-existing directory (with files) as his new Magic-Folder directory, resulting in a merge of the Magic-Folder with Bob's local directory. Filename collisions will result in conflicts because Bob's new snapshots are not descendent's of the existing Magic-Folder file snapshots.

Example: simultaneous update with four parties:

1. A, B, C, D are in sync for file “foo” at snapshot X
2. A and B simultaneously change the file, creating snapshots XA and XB (both descendants of X).
3. C hears about XA first, and D hears about XB first. Both accept an overwrite.
4. All four parties hear about the other update they hadn’t heard about yet.
5. **Result:**
    - everyone’s local file “foo” has the content pointed to by the snapshot in their DMD’s “foo” entry
    - A and C’s DMDs each have the “foo” entry pointing at snapshot XA
    - B and D’s DMDs each have the “foo” entry pointing at snapshot XB
    - A and C have a local file called foo.conflict-B,D with XB’s content
    - B and D have a local file called foo.conflict-A,C with XA’s content

Later:

- Everyone ignores the conflict, and continue updating their local “foo”. but slowly enough that there are no further conflicts, so that A and C remain in sync with eachother, and B and D remain in sync with eachother.
- A and C’s foo.conflict-B,D file continues to be updated with the latest version of the file B and D are working on, and vice-versa.
- A and C edit the file at the same time again, causing a new conflict.
- Local files are now:

A: “foo”, “foo.conflict-B,D”, “foo.conflict-C”

C: “foo”, “foo.conflict-B,D”, “foo.conflict-A”

B and D: “foo”, “foo.conflict-A”, “foo.conflict-C”

- Finally, D decides to look at “foo.conflict-A” and “foo.conflict-C”, and they manually integrate (or decide to ignore) the differences into their own local file “foo”.
- D deletes their conflict files.
- D’s DMD now points to a snapshot that is a descendant of everyone else’s current snapshot, resolving all conflicts.
- The conflict files on A, B, and C disappear, and everyone’s local file “foo” contains D’s manually-merged content.

Daira: I think it is too complicated to include multiple nicknames in the .conflict files (e.g. “foo.conflict-B,D”). It should be sufficient to have one file for each other client, reflecting that client’s latest version, regardless of who else it conflicts with.

1.6.7 Zooko’s Design (as interpreted by Daira)

A version map is a mapping from client nickname to version number.

Definition: a version map M’ strictly-follows a mapping M iff for every entry c->v in M, there is an entry c->v’ in M’ such that v’ > v.

Each client maintains a ‘local version map’ and a ‘conflict version map’ for each file in its magic folder db. If it has never written the file, then the entry for its own nickname in the local version map is zero. The conflict version map only contains entries for nicknames B where “$FILENAME.conflict-$B” exists.

When a client A uploads a file, it increments the version for its own nickname in its local version map for the file, and includes that map as metadata with its upload.

A download by client A from client B is an overwrite iff the downloaded version map strictly-follows A's local version map for that file; in this case A replaces its local version map with the downloaded version map. Otherwise it is a conflict, and the download is put into "$FILENAME.conflict-$B"; in this case A's local version map remains unchanged, and the entry B->v taken from the downloaded version map is added to its conflict version map.

If client A deletes or renames a conflict file "$FILENAME.conflict-$B", then A copies the entry for B from its conflict version map to its local version map, deletes the entry for B in its conflict version map, and performs another upload (with incremented version number) of $FILENAME.

Example: A, B, C = (10, 20, 30) everyone agrees. A updates: (11, 20, 30) B updates: (10, 21, 30)

C will see either A or B first. Both would be an overwrite, if considered alone.