Tahoe-LAFS Documentation

Release 1.x

The Tahoe-LAFS Developers

Aug 16, 2017

Contents

Contents:

CHAPTER 1

Welcome to Tahoe-LAFS!

What is Tahoe-LAFS?

Welcome to Tahoe-LAFS, the first decentralized storage system with *provider-independent security*.

Tahoe-LAFS is a system that helps you to store files. You run a client program on your computer, which talks to one or more storage servers on other computers. When you tell your client to store a file, it will encrypt that file, encode it into multiple pieces, then spread those pieces out among multiple servers. The pieces are all encrypted and protected against modifications. Later, when you ask your client to retrieve the file, it will find the necessary pieces, make sure they haven't been corrupted, reassemble them, and decrypt the result.

The client creates more pieces (or "shares") than it will eventually need, so even if some of the servers fail, you can still get your data back. Corrupt shares are detected and ignored, so the system can tolerate server-side hard-drive errors. All files are encrypted (with a unique key) before uploading, so even a malicious server operator cannot read your data. The only thing you ask of the servers is that they can (usually) provide the shares when you ask for them: you aren't relying upon them for confidentiality, integrity, or absolute availability.

What is "provider-independent security"?

Every seller of cloud storage services will tell you that their service is "secure". But what they mean by that is something fundamentally different from what we mean. What they mean by "secure" is that after you've given them the power to read and modify your data, they try really hard not to let this power be abused. This turns out to be difficult! Bugs, misconfigurations, or operator error can accidentally expose your data to another customer or to the public, or can corrupt your data. Criminals routinely gain illicit access to corporate servers. Even more insidious is the fact that the employees themselves sometimes violate customer privacy out of carelessness, avarice, or mere curiosity. The most conscientious of these service providers spend considerable effort and expense trying to mitigate these risks.

What we mean by "security" is something different. *The service provider never has the ability to read or modify your data in the first place: never.* If you use Tahoe-LAFS, then all of the threats described above are non-issues to you. Not only is it easy and inexpensive for the service provider to maintain the security of your data, but in fact they couldn't violate its security if they tried. This is what we call *provider-independent security*.

This guarantee is integrated naturally into the Tahoe-LAFS storage system and doesn't require you to perform a manual pre-encryption step or cumbersome key management. (After all, having to do cumbersome manual operations when storing or accessing your data would nullify one of the primary benefits of using cloud storage in the first place: convenience.)

Here's how it works:

A "storage grid" is made up of a number of storage servers. A storage server has direct attached storage (typically one or more hard disks). A "gateway" communicates with storage nodes, and uses them to provide access to the grid over protocols such as HTTP(S), SFTP or FTP.

Note that you can find "client" used to refer to gateway nodes (which act as a client to storage servers), and also to processes or programs connecting to a gateway node and performing operations on the grid – for example, a CLI command, Web browser, SFTP client, or FTP client.

Users do not rely on storage servers to provide *confidentiality* nor *integrity* for their data – instead all of the data is encrypted and integrity-checked by the gateway, so that the servers can neither read nor modify the contents of the files.

Users do rely on storage servers for *availability*. The ciphertext is erasure-coded into `N` shares distributed across at least `H` distinct storage servers (the default value for `N` is 10 and for `H` is 7) so that it can be recovered from any `K` of these servers (the default value of `K` is 3). Therefore only the failure of `H-K+1` (with the defaults, 5) servers can make the data unavailable.

In the typical deployment mode each user runs her own gateway on her own machine. This way she relies on her own machine for the confidentiality and integrity of the data.

An alternate deployment mode is that the gateway runs on a remote machine and the user connects to it over HTTPS or SFTP. This means that the operator of the gateway can view and modify the user's data (the user *relies on* the gateway for confidentiality and integrity), but the advantage is that the user can access the Tahoe-LAFS grid with a client that doesn't have the gateway software installed, such as an Internet kiosk or cell phone.

Access Control

There are two kinds of files: immutable and mutable. When you upload a file to the storage grid you can choose which kind of file it will be in the grid. Immutable files can't be modified once they have been uploaded. A mutable file can be modified by someone with read-write access to it. A user can have read-write access to a mutable file or read-only access to it, or no access to it at all.

A user who has read-write access to a mutable file or directory can give another user read-write access to that file or directory, or they can give read-only access to that file or directory. A user who has read-only access to a file or directory can give another user read-only access to it.

When linking a file or directory into a parent directory, you can use a read-write link or a read-only link. If you use a read-write link, then anyone who has read-write access to the parent directory can gain read-write access to the child, and anyone who has read-only access to the parent directory can gain read-only access to the child. If you use a read-only link, then anyone who has either read-write or read-only access to the parent directory can gain read-only access to the child.

For more technical detail, please see the the doc page on the Wiki.

Get Started

To use Tahoe-LAFS, please see *Installing Tahoe-LAFS*.

License

Tahoe-LAFS is an open-source project; please see the top-level README for details.

CHAPTER 2

Installing Tahoe-LAFS

Welcome to the Tahoe-LAFS project, a secure, decentralized, fault-tolerant storage system. See *Welcome to Tahoe-LAFS!* for an overview of the architecture and security properties of the system.

This procedure should work on Windows, Mac, illumos (previously OpenSolaris), and too many flavors of Linux and of BSD to list.

First: In Case Of Trouble

In some cases these instructions may fail due to peculiarities of your platform.

If the following instructions don't Just Work without any further effort on your part, then please write to the tahoe-dev mailing list where friendly hackers will help you out.

Pre-Packaged Versions

You may not need to build Tahoe at all.

If you are on Windows, please see *Building Tahoe-LAFS on Windows* for platform-specific instructions.

If you are on a Mac, you can either follow these instructions, or use the pre-packaged bundle described in *OS-X Packaging*. The Tahoe project hosts pre-compiled "wheels" for all dependencies, so use the `--find-links=` option described below to avoid needing a compiler.

Many Linux distributions include Tahoe-LAFS packages. Debian and Ubuntu users can `apt-get install tahoe-lafs`. See OSPackages for other platforms.

Preliminaries

If you don't use a pre-packaged copy of Tahoe, you can build it yourself. You'll need Python2.7, pip, and virtualenv. On unix-like platforms, you will need a C compiler, the Python development headers, and some libraries (libffi-dev

and libssl-dev).

On a modern Debian/Ubuntu-derived distribution, this command will get you everything you need:

```
apt-get install build-essential python-dev libffi-dev libssl-dev libyaml-dev python-
↪virtualenv
```

On OS-X, install pip and virtualenv as described below. If you want to compile the dependencies yourself (instead of using `--find-links` to take advantage of the pre-compiled ones we host), you'll also need to install Xcode and its command-line tools.

Python 2.7

Check if you already have an adequate version of Python installed by running `python -V`. The latest version of Python v2.7 is recommended, which is 2.7.11 as of this writing. Python v2.6.x and v3 do not work. On Windows, we recommend the use of native Python v2.7, not Cygwin Python. If you don't have one of these versions of Python installed, download and install the latest version of Python v2.7. Make sure that the path to the installation directory has no spaces in it (e.g. on Windows, do not install Python in the "Program Files" directory):

```
% python --version
Python 2.7.11
```

pip

Many Python installations already include `pip`, but in case yours does not, get it with the pip install instructions:

```
% pip --version
pip 8.1.1 from ... (python 2.7)
```

virtualenv

If you do not have an OS-provided copy of `virtualenv`, install it with the instructions from the virtualenv documentation:

```
% virtualenv --version
15.0.1
```

C compiler and libraries

Except on OS-X, where the Tahoe project hosts pre-compiled wheels for all dependencies, you will need several C libraries installed before you can build. You will also need the Python development headers, and a C compiler (your python installation should know how to find these).

On Debian/Ubuntu-derived systems, the necessary packages are `python-dev`, `libffi-dev`, and `libssl-dev`, and can be installed with `apt-get`. On RPM-based system (like Fedora) these may be named `python-devel`, etc, instead, and cam be installed with `yum` or `rpm`.

Install the Latest Tahoe-LAFS Release

We recommend creating a fresh virtualenv for your Tahoe-LAFS install, to isolate it from any python packages that are already installed (and to isolate the rest of your system from Tahoe's dependencies).

This example uses a virtualenv named `venv`, but you can call it anything you like. Many people prefer to keep all their virtualenvs in one place, like `~/.local/venvs/` or `~/venvs/`.

It's usually a good idea to upgrade the virtualenv's `pip` and `setuptools` to their latest versions, with `venv/bin/pip install -U pip setuptools`. Many operating systems have an older version of `virtualenv`, which then includes older versions of pip and setuptools. Upgrading is easy, and only affects the virtualenv: not the rest of your computer.

Then use the virtualenv's `pip` to install the latest Tahoe-LAFS release from PyPI with `venv/bin/pip install tahoe-lafs`. After installation, run `venv/bin/tahoe --version` to confirm the install was successful:

```
% virtualenv venv
New python executable in ~/venv/bin/python2.7
Installing setuptools, pip, wheel...done.

% venv/bin/pip install -U pip setuptools
Downloading/unpacking pip from https://pypi.python.org/...
...
Successfully installed pip setuptools

% venv/bin/pip install tahoe-lafs
Collecting tahoe-lafs
...
Installing collected packages: ...
Successfully installed ...

% venv/bin/tahoe --version
tahoe-lafs: 1.12.1
foolscap: ...

%
```

On OS-X, instead of `pip install tahoe-lafs`, use this command to take advantage of the hosted pre-compiled wheels:

```
venv/bin/pip install --find-links=https://tahoe-lafs.org/deps tahoe-lafs
```

Install From a Source Tarball

You can also install directly from the source tarball URL:

```
% virtualenv venv
New python executable in ~/venv/bin/python2.7
Installing setuptools, pip, wheel...done.

% venv/bin/pip install https://tahoe-lafs.org/downloads/tahoe-lafs-1.12.1.tar.bz2
Collecting https://tahoe-lafs.org/downloads/tahoe-lafs-1.12.1.tar.bz2
...
Installing collected packages: ...
Successfully installed ...
```

```
% venv/bin/tahoe --version
tahoe-lafs: 1.12.1
...
```

Hacking On Tahoe-LAFS

To modify the Tahoe source code, you should get a git checkout, and install with the `--editable` flag. You should also use the `[test]` extra to get the additional libraries needed to run the unit tests:

```
% git clone https://github.com/tahoe-lafs/tahoe-lafs.git

% cd tahoe-lafs

% virtualenv venv

% venv/bin/pip install --editable .[test]
Obtaining file::~/tahoe-lafs
...
Successfully installed ...

% venv/bin/tahoe --version
tahoe-lafs: 1.12.1.post34.dev0
...
```

This way, you won't have to re-run the `pip install` step each time you modify the source code.

Running the `tahoe` executable

The rest of the Tahoe-LAFS documentation assumes that you can run the `tahoe` executable that you just created. You have four basic options:

- Use the full path each time (e.g. `~/venv/bin/tahoe`).
- "Activate" the virtualenv with `. venv/bin/activate`, to get a subshell with a `$PATH` that includes the `venv/bin/` directory, then you can just run `tahoe`.
- Change your `$PATH` to include the `venv/bin/` directory, so you can just run `tahoe`.
- Symlink from `~/bin/tahoe` to the `tahoe` executable. Since `~/bin` is typically in your `$PATH` (at least if it exists when you log in), this will let you just run `tahoe`.

You might also find the pipsi tool convenient: `pipsi install tahoe-lafs` will create a new virtualenv, install tahoe into it, then symlink just the executable (into `~/.local/bin/tahoe`). Then either add `~/.local/bin/` to your `$PATH`, or make one last symlink into `~/bin/tahoe`.

Running the Self-Tests

To run the self-tests from a source tree, you'll need `tox` installed. On a Debian/Ubuntu system, use `apt-get install tox`. You can also install it into your tahoe-specific virtualenv with `pip install tox`.

Then just run `tox`. This will create a new fresh virtualenv, install Tahoe (from the source tree, including any changes you have made) and all its dependencies (including testing-only dependencies) into the virtualenv, then run the unit tests. This ensures that the tests are repeatable and match the results of other users, unaffected by any other Python

packages installed on your machine. On a modern computer this will take 5-10 minutes, and should result in a "all tests passed" mesage:

```
% tox
GLOB sdist-make: ~/tahoe-lafs/setup.py
py27 recreate: ~/tahoe-lafs/.tox/py27
py27 inst: ~/tahoe-lafs/.tox/dist/tahoe-lafs-1.12.1.post8.dev0.zip
py27 runtests: commands[0] | tahoe --version
py27 runtests: commands[1] | trial --rterrors allmydata
allmydata.test.test_auth
  AccountFileCheckerKeyTests
    test_authenticated ...                                          [OK]
    test_missing_signature ...                                      [OK]
 ...
Ran 1186 tests in 423.179s

PASSED (skips=7, expectedFailures=3, successes=1176)
_____________________________ summary _______________________________
  py27: commands succeeded
  congratulations :)
```

Common Problems

If you see an error like `fatal error: Python.h: No such file or directory` while compiling the dependencies, you need the Python development headers. If you are on a Debian or Ubuntu system, you can install them with `sudo apt-get install python-dev`. On RedHat/Fedora, install `python-devel`.

Similar errors about `openssl/crypto.h` indicate that you are missing the OpenSSL development headers (`libssl-dev`). Likewise `ffi.h` means you need `libffi-dev`.

Using Tahoe-LAFS

Now you are ready to deploy a decentralized filesystem. You will use the `tahoe` executable to create, configure, and launch your Tahoe-LAFS nodes. See *How To Run Tahoe-LAFS* for instructions on how to do that.

CHAPTER 3

How To Run Tahoe-LAFS

Introduction

This is how to run a Tahoe-LAFS client or a complete Tahoe-LAFS grid. First you have to install the Tahoe-LAFS software, as documented in *Installing Tahoe-LAFS*.

The `tahoe` program in your virtualenv's `bin` directory is used to create, start, and stop nodes. Each node lives in a separate base directory, in which there is a configuration file named `tahoe.cfg`. Nodes read and write files within this base directory.

A grid consists of a set of *storage nodes* and *client nodes* running the Tahoe-LAFS code. There is also an *introducer node* that is responsible for getting the other nodes talking to each other.

If you're getting started we recommend you try connecting to the public test grid as you only need to create a client node. When you want to create your own grid you'll need to create the introducer and several initial storage nodes (see the note about small grids below).

Being Introduced to a Grid

A collection of Tahoe servers is called a Grid and usually has 1 Introducer (but sometimes more, and it's possible to run with zero). The Introducer announces which storage servers constitute the Grid and how to contact them. There is a secret "fURL" you need to know to talk to the Introducer.

One way to get this secret is using traditional tools such as encrypted email, encrypted instant-messaging, etcetera. It is important to transmit this fURL secretly as knowing it gives you access to the Grid.

An additional way to share the fURL securely is via magic wormhole. This uses a weak one-time password and a server on the internet (at *wormhole.tahoe-lafs.org*) to open a secure channel between two computers. In Tahoe-LAFS this functions via the commands *tahoe invite* and *tahoe create-client –join*. A person who already has access to a Grid can use *tahoe invite* to create one end of the magic wormhole and then transmits some JSON (including the Introducer's secret fURL) to the other end. *tahoe invite* will print a one-time secret code; you must then communicate this code to the person who will join the Grid.

The other end of the magic wormhole in this case is *tahoe create-client –join <one-time code>*, where the person being invited types in the code they were given. Ideally, this code would be transmitted securely. It is, however, only useful exactly once. Also, it is much easier to transcribe by a human. Codes look like *7-surrender-tunnel* (a short number and two words).

Running a Client

To construct a client node, run "`tahoe create-client`", which will create `~/.tahoe` to be the node's base directory. Acquire the `introducer.furl` (see below if you are running your own introducer, or use the one from the TestGrid page), and paste it after `introducer.furl =` in the `[client]` section of `~/.tahoe/tahoe.cfg`. Then use "`tahoe run ~/.tahoe`". After that, the node should be off and running. The first thing it will do is connect to the introducer and get itself connected to all other nodes on the grid.

Some Grids use "magic wormhole" one-time codes to configure the basic options. In such a case you use `tahoe create-client --join <one-time-code>` and do not have to do any of the `tahoe.cfg` editing mentioned above.

By default, "`tahoe create-client`" creates a client-only node, that does not offer its disk space to other nodes. To configure other behavior, use "`tahoe create-node`" or see *Configuring a Tahoe-LAFS node*.

The "`tahoe run`" command above will run the node in the foreground. On Unix, you can run it in the background instead by using the "`tahoe start`" command. To stop a node started in this way, use "`tahoe stop`". `tahoe --help` gives a summary of all commands.

Running a Server or Introducer

To build either a storage server node, or an introducer node, you'll need a way for clients to connect to it. The simplest case is when the computer is on the public internet (e.g. a "VPS" virtual private server, with a public IP address and a DNS hostname like `example.net`). See *How To Configure A Server* for help with more complex scenarios, using the `--port` and `--location` arguments.

To construct an introducer, create a new base directory for it (the name of the directory is up to you), `cd` into it, and run "`tahoe create-introducer --hostname=example.net .`" (but using the hostname of your VPS). Now run the introducer using "`tahoe start .`". After it starts, it will write a file named `introducer.furl` into the `private/` subdirectory of that base directory. This file contains the URL the other nodes must use in order to connect to this introducer. (Note that "`tahoe run .`" doesn't work for introducers, this is a known issue: #937.)

You can distribute your Introducer fURL securely to new clients by using the `tahoe invite` command. This will prepare some JSON to send to the other side, request a magic wormhole code from `wormhole.tahoe-lafs.org` and print it out to the terminal. This one-time code should be transmitted to the user of the client, who can then run `tahoe create-client --join <one-time-code>`.

Storage servers are created the same way: `tahoe create-node --hostname=HOSTNAME .` from a new directory. You'll need to provide the introducer FURL (either as a `--introducer=` argument, or by editing the `tahoe.cfg` configuration file afterwards) to connect to the introducer of your choice.

See *Configuring a Tahoe-LAFS node* for more details about how to configure Tahoe-LAFS.

A note about small grids

By default, Tahoe-LAFS ships with the configuration parameter `shares.happy` set to 7. If you are using Tahoe-LAFS on a grid with fewer than 7 storage nodes, this won't work well for you — none of your uploads will succeed. To fix this, see *Configuring a Tahoe-LAFS node* to learn how to set `shares.happy` to a more suitable value for your grid.

Do Stuff With It

This is how to use your Tahoe-LAFS node.

The WUI

Point your web browser to http://127.0.0.1:3456 — which is the URL of the gateway running on your own local computer — to use your newly created node.

Create a new directory (with the button labelled "create a directory"). Your web browser will load the new directory. Now if you want to be able to come back to this directory later, you have to bookmark it, or otherwise save a copy of the URL. If you lose the URL to this directory, then you can never again come back to this directory.

The CLI

Prefer the command-line? Run "`tahoe --help`" (the same command-line tool that is used to start and stop nodes serves to navigate and use the decentralized file store). To get started, create a new directory and mark it as the 'tahoe:' alias by running "`tahoe create-alias tahoe`". Once you've done that, you can do "`tahoe ls tahoe:`" and "`tahoe cp LOCALFILE tahoe:foo.txt`" to work with your file store. The Tahoe-LAFS CLI uses similar syntax to the well-known scp and rsync tools. See *The Tahoe-LAFS CLI commands* for more details.

To backup a directory full of files and subdirectories, run "`tahoe backup LOCALDIRECTORY tahoe:`". This will create a new LAFS subdirectory inside the "tahoe" LAFS directory named "Archive", and inside "Archive", it will create a new subdirectory whose name is the current date and time. That newly created subdirectory will be populated with a snapshot copy of all files and directories currently reachable from LOCALDIRECTORY. Then `tahoe backup` will make a link to that snapshot directory from the "tahoe" LAFS directory, and name the link "Latest".

`tahoe backup` cleverly avoids uploading any files or directories that haven't changed, and it also cleverly deduplicates any files or directories that have identical contents to other files or directories that it has previously backed-up. This means that running `tahoe backup` is a nice incremental operation that backs up your files and directories efficiently, and if it gets interrupted (for example by a network outage, or by you rebooting your computer during the backup, or so on), it will resume right where it left off the next time you run `tahoe backup`.

See *The Tahoe-LAFS CLI commands* for more information about the `tahoe backup` command, as well as other commands.

As with the WUI (and with all current interfaces to Tahoe-LAFS), you are responsible for remembering directory capabilities yourself. If you create a new directory and lose the capability to it, then you cannot access that directory ever again.

The SFTP and FTP frontends

You can access your Tahoe-LAFS grid via any SFTP or FTP client. See *Tahoe-LAFS SFTP and FTP Frontends* for how to set this up. On most Unix platforms, you can also use SFTP to plug Tahoe-LAFS into your computer's local filesystem via `sshfs`, but see the FAQ about performance problems.

The SftpFrontend page on the wiki has more information about using SFTP with Tahoe-LAFS.

The WAPI

Want to program your Tahoe-LAFS node to do your bidding? Easy! See *The Tahoe REST-ful Web API*.

Socialize

You can chat with other users of and hackers of this software on the #tahoe-lafs IRC channel at `irc.freenode.net`, or on the tahoe-dev mailing list.

Complain

Bugs can be filed on the Tahoe-LAFS "Trac" instance, at https://tahoe-lafs.org/trac/ .

You can also "fork" the repo and submit Pull Requests on Github: https://github.com/tahoe-lafs/tahoe-lafs .

CHAPTER 4

Magic Wormhole Invites

Magic Wormhole

magic wormhole is a server and a client which together use Password Authenticated Key Exchange (PAKE) to use a short code to establish a secure channel between two computers. These codes are one-time use and an attacker gets at most one "guess", thus allowing low-entropy codes to be used.

Invites and Joins

Inside Tahoe-LAFS we are using a channel created using magic wormhole to exchange configuration and the secret fURL of the Introducer with new clients. In the future, we would like to make the Magic Folder (*Magic Folder HOWTO*) invites and joins work this way as well.

This is a two-part process. Alice runs a grid and wishes to have her friend Bob use it as a client. She runs `tahoe invite bob` which will print out a short "wormhole code" like `2-unicorn-quiver`. You may also include some options for total, happy and needed shares if you like.

Alice then transmits this one-time secret code to Bob. Alice must keep her command running until Bob has done his step as it is waiting until a secure channel is established before sending the data.

Bob then runs `tahoe create-client --join <secret code>` with any other options he likes. This will "use up" the code establishing a secure session with Alice's computer. If an attacker tries to guess the code, they get only once chance to do so (and then Bob's side will fail). Once Bob's computer has connected to Alice's computer, the two computers performs the protocol described below, resulting in some JSON with the Introducer fURL, nickname and any other options being sent to Bob's computer. The `tahoe create-client` command then uses these options to set up Bob's client.

Tahoe-LAFS Secret Exchange

The protocol that the Alice (the one doing the invite) and Bob (the one being invited) sides perform once a magic wormhole secure channel has been established goes as follows:

Alice and Bob both immediately send an "abilities" message as JSON. For Alice this is `{"abilities": {"server-v1": {}}}`. For Bob, this is `{"abilities": {"client-v1": {}}}`.

After receiving the message from the other side and confirming the expected protocol, Alice transmits the configuration JSON:

```
{
    "needed": 3,
    "total": 10,
    "happy": 7,
    "nickname": "bob",
    "introducer": "pb://xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx@example.com:41505/
↪yyyyyyyyyyyyyyyyyyyyyy"
}
```

Both sides then disconnect.

As you can see, there is room for future revisions of the protocol but as of yet none have been sketched out.

CHAPTER 5

Configuring a Tahoe-LAFS node

A Tahoe-LAFS node is configured by writing to files in its base directory. These files are read by the node when it starts, so each time you change them, you need to restart the node.

The node also writes state to its base directory, so it will create files on its own.

This document contains a complete list of the config files that are examined by the client node, as well as the state files that you'll observe in its base directory.

The main file is named "`tahoe.cfg`", and is an ".INI"-style configuration file (parsed by the Python stdlib 'ConfigParser' module: "`[name]`" section markers, lines with "`key.subkey: value`", rfc822-style continuations). There are also other files containing information that does not easily fit into this format. The "`tahoe create-node`" or "`tahoe create-client`" command will create an initial `tahoe.cfg` file for you. After creation, the node will never modify the `tahoe.cfg` file: all persistent state is put in other files.

The item descriptions below use the following types:

`boolean`

one of (True, yes, on, 1, False, off, no, 0), case-insensitive

`strports string`

a Twisted listening-port specification string, like "`tcp:80`" or "`tcp:3456:interface=127.0.0.1`". For a full description of the format, see the Twisted strports documentation. Please note, if interface= is not specified, Tahoe-LAFS will attempt to bind the port specified on all interfaces.

`endpoint specification string`

a Twisted Endpoint specification string, like "`tcp:80`" or "`tcp:3456:interface=127.0.0.1`". These are replacing strports strings. For a full description of the format, see the Twisted Endpoints documentation. Please note, if interface= is not specified, Tahoe-LAFS will attempt to bind the port specified on all interfaces. Also note that `tub.port` only works with TCP endpoints right now.

`FURL string`

a Foolscap endpoint identifier, like `pb://soklj4y7eok5c3xkmjeqpw@192.168.69.247:44801/eqpwqtzm`

Node Types

A node can be a client/server, an introducer, or a statistics gatherer.

Client/server nodes provide one or more of the following services:

- web-API service
- SFTP service
- FTP service
- Magic Folder service
- helper service
- storage service.

A client/server that provides storage service (i.e. storing shares for clients) is called a "storage server". If it provides any of the other services, it is a "storage client" (a node can be both a storage server and a storage client). A client/server node that provides web-API service is called a "gateway".

Overall Node Configuration

This section controls the network behavior of the node overall: which ports and IP addresses are used, when connections are timed out, etc. This configuration applies to all node types and is independent of the services that the node is offering.

If your node is behind a firewall or NAT device and you want other clients to connect to it, you'll need to open a port in the firewall or NAT, and specify that port number in the tub.port option. If behind a NAT, you *may* need to set the `tub.location` option described below.

`[node]`

`nickname = (UTF-8 string, optional)`

This value will be displayed in management tools as this node's "nickname". If not provided, the nickname will be set to "<unspecified>". This string shall be a UTF-8 encoded Unicode string.

`web.port = (strports string, optional)`

This controls where the node's web server should listen, providing node status and, if the node is a client/server, providing web-API service as defined in *The Tahoe REST-ful Web API*.

This file contains a Twisted "strports" specification such as "3456" or "tcp:3456:interface=127.0.0.1". The "tahoe create-node" or "tahoe create-client" commands set the web.port to "tcp:3456:interface=127.0.0.1" by default; this is overridable by the --webport option. You can make it use SSL by writing "ssl:3456:privateKey=mykey.pem:certKey=cert.pem" instead.

If this is not provided, the node will not run a web server.

web.static = (string, optional)

This controls where the /static portion of the URL space is served. The value is a directory name (~username is allowed, and non-absolute names are interpreted relative to the node's basedir), which can contain HTML and other files. This can be used to serve a Javascript-based frontend to the Tahoe-LAFS node, or other services.

The default value is "public_html", which will serve BASEDIR/public_html . With the default settings, http://127.0.0.1:3456/static/foo.html will serve the contents of BASEDIR/public_html/foo.html .

tub.port = (endpoint specification strings or "disabled", optional)

This controls which port the node uses to accept Foolscap connections from other nodes. It is parsed as a comma-separated list of Twisted "server endpoint descriptor" strings, each of which is a value like tcp:12345 and tcp:23456:interface=127.0.0.1.

To listen on multiple ports at once (e.g. both TCP-on-IPv4 and TCP-on-IPv6), use something like tcp6:interface=2600\:3c01\:f03c\:91ff\:fe93\:d272:3456, tcp:interface=8.8.8.8:3456. Lists of endpoint descriptor strings like the following tcp:12345,tcp6:12345 are known to not work because an Address already in use. error.

If tub.port is the string disabled, the node will not listen at all, and thus cannot accept connections from other nodes. If [storage] enabled = true, or [helper] enabled = true, or the node is an Introducer, then it is an error to have tub.port be empty. If tub.port is disabled, then tub.location must also be disabled, and vice versa.

For backwards compatibility, if this contains a simple integer, it will be used as a TCP port number, like tcp:%d (which will accept connections on all interfaces). However tub.port cannot be 0 or tcp:0 (older versions accepted this, but the node is no longer willing to ask Twisted to allocate port numbers in this way). If tub.port is present, it may not be empty.

If the tub.port config key is not provided (e.g. tub.port appears nowhere in the [node] section, or is commented out), the node will look in BASEDIR/client.port (or BASEDIR/introducer.port, for introducers) for the descriptor that was used last time.

If neither tub.port nor the port file is available, the node will ask the kernel to allocate any available port (the moral equivalent of tcp:0). The allocated port number will be written into a descriptor string in BASEDIR/client.port (or introducer.port), so that subsequent runs will re-use the same port.

tub.location = (hint string or "disabled", optional)

In addition to running as a client, each Tahoe-LAFS node can also run as a server, listening for connections from other Tahoe-LAFS clients. The node announces its location by publishing a "FURL" (a string with some connection hints) to the Introducer. The string it publishes can be found in BASEDIR/private/storage.furl . The tub.location configuration controls what location is published in this announcement.

If your node is meant to run as a server, you should fill this in, using a hostname or IP address that is reachable from your intended clients.

If `tub.port` is set to `disabled`, then `tub.location` must also be `disabled`.

If you don't provide `tub.location`, the node will try to figure out a useful one by itself, by using tools like "`ifconfig`" to determine the set of IP addresses on which it can be reached from nodes both near and far. It will also include the TCP port number on which it is listening (either the one specified by `tub.port`, or whichever port was assigned by the kernel when `tub.port` is left unspecified). However this automatic address-detection is discouraged, and will probably be removed from a future release. It will include the `127.0.0.1` "localhost" address (which is only useful to clients running on the same computer), and RFC1918 private-network addresses like `10.*.*.*` and `192.168.*.*` (which are only useful to clients on the local LAN). In general, the automatically-detected IP addresses will only be useful if the node has a public IP address, such as a VPS or colo-hosted server.

You will certainly need to set `tub.location` if your node lives behind a firewall that is doing inbound port forwarding, or if you are using other proxies such that the local IP address or port number is not the same one that remote clients should use to connect. You might also want to control this when using a Tor proxy to avoid revealing your actual IP address through the Introducer announcement.

If `tub.location` is specified, by default it entirely replaces the automatically determined set of IP addresses. To include the automatically determined addresses as well as the specified ones, include the uppercase string "`AUTO`" in the list.

The value is a comma-separated string of method:host:port location hints, like this:

```
tcp:123.45.67.89:8098,tcp:tahoe.example.com:8098,tcp:127.0.0.1:8098
```

A few examples:

- Don't listen at all (client-only mode):

```
tub.port = disabled
tub.location = disabled
```

- Use a DNS name so you can change the IP address more easily:

```
tub.port = tcp:8098
tub.location = tcp:tahoe.example.com:8098
```

- Run a node behind a firewall (which has an external IP address) that has been configured to forward external port 7912 to our internal node's port 8098:

```
tub.port = tcp:8098
tub.location = tcp:external-firewall.example.com:7912
```

- Emulate default behavior, assuming your host has public IP address of 123.45.67.89, and the kernel-allocated port number was 8098:

```
tub.port = tcp:8098
tub.location = tcp:123.45.67.89:8098,tcp:127.0.0.1:8098
```

- Use a DNS name but also include the default set of addresses:

```
tub.port = tcp:8098
tub.location = tcp:tahoe.example.com:8098,AUTO
```

- Run a node behind a Tor proxy (perhaps via `torsocks`), in client-only mode (i.e. we can make outbound connections, but other nodes will not be able to connect to us). The literal '`unreachable.`

`example.org`‘ will not resolve, but will serve as a reminder to human observers that this node cannot be reached. “Don’t call us.. we’ll call you”:

```
tub.port = tcp:8098
tub.location = tcp:unreachable.example.org:0
```

- Run a node behind a Tor proxy, and make the server available as a Tor “hidden service”. (This assumes that other clients are running their node with `torsocks`, such that they are prepared to connect to a `.onion` address.) The hidden service must first be configured in Tor, by giving it a local port number and then obtaining a `.onion` name, using something in the `torrc` file like:

```
HiddenServiceDir /var/lib/tor/hidden_services/tahoe
HiddenServicePort 29212 127.0.0.1:8098
```

once Tor is restarted, the `.onion` hostname will be in `/var/lib/tor/hidden_services/tahoe/hostname`. Then set up your `tahoe.cfg` like:

```
tub.port = tcp:8098
tub.location = tor:ualhejtq2p7ohfbb.onion:29212
```

`log_gatherer.furl = (FURL, optional)`

If provided, this contains a single FURL string that is used to contact a “log gatherer”, which will be granted access to the logport. This can be used to gather operational logs in a single place. Note that in previous releases of Tahoe-LAFS, if an old-style `BASEDIR/log_gatherer.furl` file existed it would also be used in addition to this value, allowing multiple log gatherers to be used at once. As of Tahoe-LAFS v1.9.0, an old-style file is ignored and a warning will be emitted if one is detected. This means that as of Tahoe-LAFS v1.9.0 you can have at most one log gatherer per node. See ticket #1423 about lifting this restriction and letting you have multiple log gatherers.

`timeout.keepalive = (integer in seconds, optional)`

`timeout.disconnect = (integer in seconds, optional)`

If `timeout.keepalive` is provided, it is treated as an integral number of seconds, and sets the Foolscap “keepalive timer” to that value. For each connection to another node, if nothing has been heard for a while, we will attempt to provoke the other end into saying something. The duration of silence that passes before sending the PING will be between KT and 2*KT. This is mainly intended to keep NAT boxes from expiring idle TCP sessions, but also gives TCP’s long-duration keepalive/disconnect timers some traffic to work with. The default value is 240 (i.e. 4 minutes).

If timeout.disconnect is provided, this is treated as an integral number of seconds, and sets the Foolscap “disconnect timer” to that value. For each connection to another node, if nothing has been heard for a while, we will drop the connection. The duration of silence that passes before dropping the connection will be between DT-2*KT and 2*DT+2*KT (please see ticket #521 for more details). If we are sending a large amount of data to the other end (which takes more than DT-2*KT to deliver), we might incorrectly drop the connection. The default behavior (when this value is not provided) is to disable the disconnect timer.

See ticket #521 for a discussion of how to pick these timeout values. Using 30 minutes means we’ll disconnect after 22 to 68 minutes of inactivity. Receiving data will reset this timeout, however if we have more than 22min of data in the outbound queue (such as 800kB in two pipelined segments of 10 shares each) and the far end has no need to contact us, our ping might be delayed, so we may disconnect them by accident.

`tempdir = (string, optional)`

This specifies a temporary directory for the web-API server to use, for holding large files while they are being uploaded. If a web-API client attempts to upload a 10GB file, this tempdir will need to have at least

10GB available for the upload to complete.

The default value is the `tmp` directory in the node's base directory (i.e. `BASEDIR/tmp`), but it can be placed elsewhere. This directory is used for files that usually (on a Unix system) go into `/tmp`. The string will be interpreted relative to the node's base directory.

`reveal-IP-address = (boolean, optional, defaults to True)`

This is a safety flag. When set to False (aka "private mode"), the node will refuse to start if any of the other configuration options would reveal the node's IP address to servers or the external network. This flag does not directly affect the node's behavior: its only power is to veto node startup when something looks unsafe.

The default is True (non-private mode), because setting it to False requires the installation of additional libraries (use `pip install tahoe-lafs[tor]` and/or `pip install tahoe-lafs[i2p]` to get them) as well as additional non-python software (Tor/I2P daemons). Performance is also generally reduced when operating in private mode.

When False, any of the following configuration problems will cause `tahoe start` to throw a PrivacyError instead of starting the node:

- `[node] tub.location` contains any `tcp:` hints
- `[node] tub.location` uses `AUTO`, or is missing/empty (because that defaults to AUTO)
- `[connections] tcp =` is set to `tcp` (or left as the default), rather than being set to `tor` or `disabled`

Connection Management

Three sections (`[tor]`, `[i2p]`, and `[connections]`) control how the Tahoe node makes outbound connections. Tor and I2P are configured here. This also controls when Tor and I2P are used: for all TCP connections (to hide your IP address), or only when necessary (just for servers which declare that they need Tor, because they use `.onion` addresses).

Note that if you want to protect your node's IP address, you should set `[node] reveal-IP-address = False`, which will refuse to launch the node if any of the other configuration settings might violate this privacy property.

`[connections]`

This section controls *when* Tor and I2P are used. The `[tor]` and `[i2p]` sections (described later) control *how* Tor/I2P connections are managed.

All Tahoe nodes need to make a connection to the Introducer; the `[client] introducer.furl` setting (described below) indicates where the Introducer lives. Tahoe client nodes must also make connections to storage servers: these targets are specified in announcements that come from the Introducer. Both are expressed as FURLs (a Foolscap URL), which include a list of "connection hints". Each connection hint describes one (of perhaps many) network endpoints where the service might live.

Connection hints include a type, and look like:

- `tcp:tahoe.example.org:12345`
- `tor:u33m4y7klhz3b.onion:1000`
- `i2p:c2ng2pbrmxmlwpijn`

`tor` hints are always handled by the `tor` handler (configured in the `[tor]` section, described below). Likewise, `i2p` hints are always routed to the `i2p` handler. But either will be ignored if Tahoe was not installed with the necessary Tor/I2P support libraries, or if the Tor/I2P daemon is unreachable.

The `[connections]` section lets you control how `tcp` hints are handled. By default, they use the normal TCP handler, which just makes direct connections (revealing your node's IP address to both the target server and the intermediate network). The node behaves this way if the `[connections]` section is missing entirely, or if it looks like this:

```
[connections]
 tcp = tcp
```

To hide the Tahoe node's IP address from the servers that it uses, set the `[connections]` section to use Tor for TCP hints:

```
[connections]
 tcp = tor
```

You can also disable TCP hints entirely, which would be appropriate when running an I2P-only node:

```
[connections]
 tcp = disabled
```

(Note that I2P does not support connections to normal TCP ports, so `[connections] tcp = i2p` is invalid)

In the future, Tahoe services may be changed to live on HTTP/HTTPS URLs instead of Foolscap. In that case, connections will be made using whatever handler is configured for `tcp` hints. So the same `tcp = tor` configuration will work.

[tor]

This controls how Tor connections are made. The defaults (all empty) mean that, when Tor is needed, the node will try to connect to a Tor daemon's SOCKS proxy on localhost port 9050 or 9150. Port 9050 is the default Tor SOCKS port, so it should be available under any system Tor instance (e.g. the one launched at boot time when the standard Debian `tor` package is installed). Port 9150 is the SOCKS port for the Tor Browser Bundle, so it will be available any time the TBB is running.

You can set `launch = True` to cause the Tahoe node to launch a new Tor daemon when it starts up (and kill it at shutdown), if you don't have a system-wide instance available. Note that it takes 30-60 seconds for Tor to get running, so using a long-running Tor process may enable a faster startup. If your Tor executable doesn't live on `$PATH`, use `tor.executable=` to specify it.

`[tor]`

`enabled = (boolean, optional, defaults to True)`

> If False, this will disable the use of Tor entirely. The default of True means the node will use Tor, if necessary, and if possible.

`socks.port = (string, optional, endpoint specification string, defaults to empty)`

> This tells the node that Tor connections should be routed to a SOCKS proxy listening on the given endpoint. The default (of an empty value) will cause the node to first try localhost port 9050, then if that fails, try localhost port 9150. These are the default listening ports of the standard Tor daemon, and the Tor Browser Bundle, respectively.
>
> While this nominally accepts an arbitrary endpoint string, internal limitations prevent it from accepting anything but `tcp:HOST:PORT` (unfortunately, unix-domain sockets are not yet supported). See ticket

#2813 for details. Also note that using a HOST of anything other than localhost is discouraged, because you would be revealing your IP address to external (and possibly hostile) machines.

`control.port = (string, optional, endpoint specification string)`

This tells the node to connect to a pre-existing Tor daemon on the given control port (which is typically `unix:/var/run/tor/control` or `tcp:localhost:9051`). The node will then ask Tor what SOCKS port it is using, and route Tor connections to that.

`launch = (bool, optional, defaults to False)`

If True, the node will spawn a new (private) copy of Tor at startup, and will kill it at shutdown. The new Tor will be given a persistent state directory under `NODEDIR/private/`, where Tor's microdescriptors will be cached, to speed up subsequent startup.

`tor.executable = (string, optional, defaults to empty)`

This controls which Tor executable is used when `launch = True`. If empty, the first executable program named `tor` found on `$PATH` will be used.

There are 5 valid combinations of these configuration settings:

- 1: `(empty)`: use SOCKS on port 9050/9150
- 2: `launch = true`: launch a new Tor
- 3: `socks.port = tcp:HOST:PORT`: use an existing Tor on the given SOCKS port
- 4: `control.port = ENDPOINT`: use an existing Tor at the given control port
- 5: `enabled = false`: no Tor at all

1 is the default, and should work for any Linux host with the system Tor package installed. 2 should work on any box with Tor installed into $PATH, but will take an extra 30-60 seconds at startup. 3 and 4 can be used for specialized installations, where Tor is already running, but not listening on the default port. 5 should be used in environments where Tor is installed, but should not be used (perhaps due to a site-wide policy).

Note that Tor support depends upon some additional Python libraries. To install Tahoe with Tor support, use `pip install tahoe-lafs[tor]`.

[i2p]

This controls how I2P connections are made. Like with Tor, the all-empty defaults will cause I2P connections to be routed to a pre-existing I2P daemon on port 7656. This is the default SAM port for the `i2p` daemon.

`[i2p]`

`enabled = (boolean, optional, defaults to True)`

If False, this will disable the use of I2P entirely. The default of True means the node will use I2P, if necessary, and if possible.

`sam.port = (string, optional, endpoint descriptor, defaults to empty)`

This tells the node that I2P connections should be made via the SAM protocol on the given port. The default (of an empty value) will cause the node to try localhost port 7656. This is the default listening port of the standard I2P daemon.

`launch = (bool, optional, defaults to False)`

If True, the node will spawn a new (private) copy of I2P at startup, and will kill it at shutdown. The new I2P will be given a persistent state directory under `NODEDIR/private/`, where I2P's microdescriptors will be cached, to speed up subsequent startup. The daemon will allocate its own SAM port, which will be queried from the config directory.

`i2p.configdir = (string, optional, directory)`

This tells the node to parse an I2P config file in the given directory, and use the SAM port it finds there. If `launch = True`, the new I2P daemon will be told to use the given directory (which can be pre-populated with a suitable config file). If `launch = False`, we assume there is a pre-running I2P daemon running from this directory, and can again parse the config file for the SAM port.

`i2p.executable = (string, optional, defaults to empty)`

This controls which I2P executable is used when `launch = True`. If empty, the first executable program named `i2p` found on `$PATH` will be used.

Client Configuration

`[client]`

`introducer.furl = (FURL string, mandatory)`

This FURL tells the client how to connect to the introducer. Each Tahoe-LAFS grid is defined by an introducer. The introducer's FURL is created by the introducer node and written into its private base directory when it starts, whereupon it should be published to everyone who wishes to attach a client to that grid

`helper.furl = (FURL string, optional)`

If provided, the node will attempt to connect to and use the given helper for uploads. See *The Tahoe Upload Helper* for details.

`stats_gatherer.furl = (FURL string, optional)`

If provided, the node will connect to the given stats gatherer and provide it with operational statistics.

`shares.needed = (int, optional) aka "k", default 3`

`shares.total = (int, optional) aka "N", N >= k, default 10`

`shares.happy = (int, optional) 1 <= happy <= N, default 7`

These three values set the default encoding parameters. Each time a new file is uploaded, erasure-coding is used to break the ciphertext into separate shares. There will be `N` (i.e. `shares.total`) shares created, and the file will be recoverable if any `k` (i.e. `shares.needed`) shares are retrieved. The default values are 3-of-10 (i.e. `shares.needed = 3`, `shares.total = 10`). Setting `k` to 1 is equivalent to simple replication (uploading `N` copies of the file).

These values control the tradeoff between storage overhead and reliability. To a first approximation, a 1MB file will use (1MB * `N`/`k`) of backend storage space (the actual value will be a bit more, because of other forms of overhead). Up to `N`-`k` shares can be lost before the file becomes unrecoverable. So large `N`/`k` ratios are more reliable, and small `N`/`k` ratios use less disk space. `N` cannot be larger than 256, because of the 8-bit erasure-coding algorithm that Tahoe-LAFS uses. `k` can not be greater than `N`. See *Performance costs for some common operations* for more details.

`shares.happy` allows you control over how well to "spread out" the shares of an immutable file. For a successful upload, shares are guaranteed to be initially placed on at least `shares.happy` distinct servers, the correct functioning of any `k` of which is sufficient to guarantee the availability of the uploaded file. This value should not be larger than the number of servers on your grid.

A value of `shares.happy` <= `k` is allowed, but this is not guaranteed to provide any redundancy if some servers fail or lose shares. It may still provide redundancy in practice if `N` is greater than the number of connected servers, because in that case there will typically be more than one share on at least some

storage nodes. However, since a successful upload only guarantees that at least `shares.happy` shares have been stored, the worst case is still that there is no redundancy.

(Mutable files use a different share placement algorithm that does not currently consider this parameter.)

`mutable.format = sdmf or mdmf`

This value tells Tahoe-LAFS what the default mutable file format should be. If `mutable.format=sdmf`, then newly created mutable files will be in the old SDMF format. This is desirable for clients that operate on grids where some peers run older versions of Tahoe-LAFS, as these older versions cannot read the new MDMF mutable file format. If `mutable.format` is `mdmf`, then newly created mutable files will use the new MDMF format, which supports efficient in-place modification and streaming downloads. You can overwrite this value using a special mutable-type parameter in the webapi. If you do not specify a value here, Tahoe-LAFS will use SDMF for all newly-created mutable files.

Note that this parameter applies only to files, not to directories. Mutable directories, which are stored in mutable files, are not controlled by this parameter and will always use SDMF. We may revisit this decision in future versions of Tahoe-LAFS.

See *Mutable Files* for details about mutable file formats.

`peers.preferred = (string, optional)`

This is an optional comma-separated list of Node IDs of servers that will be tried first when selecting storage servers for reading or writing.

Servers should be identified here by their Node ID as it appears in the web ui, underneath the server's nickname. For storage servers running tahoe versions >=1.10 (if the introducer is also running tahoe >=1.10) this will be a "Node Key" (which is prefixed with 'v0-'). For older nodes, it will be a TubID instead. When a preferred server (and/or the introducer) is upgraded to 1.10 or later, clients must adjust their configs accordingly.

Every node selected for upload, whether preferred or not, will still receive the same number of shares (one, if there are `N` or more servers accepting uploads). Preferred nodes are simply moved to the front of the server selection lists computed for each file.

This is useful if a subset of your nodes have different availability or connectivity characteristics than the rest of the grid. For instance, if there are more than `N` servers on the grid, and `K` or more of them are at a single physical location, it would make sense for clients at that location to prefer their local servers so that they can maintain access to all of their uploads without using the internet.

Frontend Configuration

The Tahoe-LAFS client process can run a variety of frontend file access protocols. You will use these to create and retrieve files from the Tahoe-LAFS file store. Configuration details for each are documented in the following protocol-specific guides:

HTTP

Tahoe runs a webserver by default on port 3456. This interface provides a human-oriented "WUI", with pages to create, modify, and browse directories and files, as well as a number of pages to check on the status of your Tahoe node. It also provides a machine-oriented "WAPI", with a REST-ful HTTP interface that can be used by other programs (including the CLI tools). Please see *The Tahoe REST-ful Web API* for full details, and the `web.port` and `web.static` config variables above. *Download status* also describes a few WUI status pages.

CLI

The main `tahoe` executable includes subcommands for manipulating the file store, uploading/downloading files, and creating/running Tahoe nodes. See *The Tahoe-LAFS CLI commands* for details.

SFTP, FTP

Tahoe can also run both SFTP and FTP servers, and map a username/password pair to a top-level Tahoe directory. See *Tahoe-LAFS SFTP and FTP Frontends* for instructions on configuring these services, and the `[sftpd]` and `[ftpd]` sections of `tahoe.cfg`.

Magic Folder

A node running on Linux or Windows can be configured to automatically upload files that are created or changed in a specified local directory. See *Tahoe-LAFS Magic Folder Frontend* for details.

Storage Server Configuration

`[storage]`

`enabled = (boolean, optional)`

If this is `True`, the node will run a storage server, offering space to other clients. If it is `False`, the node will not run a storage server, meaning that no shares will be stored on this node. Use `False` for clients who do not wish to provide storage service. The default value is `True`.

`readonly = (boolean, optional)`

If `True`, the node will run a storage server but will not accept any shares, making it effectively read-only. Use this for storage servers that are being decommissioned: the `storage/` directory could be mounted read-only, while shares are moved to other servers. Note that this currently only affects immutable shares. Mutable shares (used for directories) will be written and modified anyway. See ticket #390 for the current status of this bug. The default value is `False`.

`reserved_space = (str, optional)`

If provided, this value defines how much disk space is reserved: the storage server will not accept any share that causes the amount of free disk space to drop below this value. (The free space is measured by a call to `statvfs(2)` on Unix, or `GetDiskFreeSpaceEx` on Windows, and is the space available to the user account under which the storage server runs.)

This string contains a number, with an optional case-insensitive scale suffix, optionally followed by "B" or "iB". The supported scale suffixes are "K", "M", "G", "T", "P" and "E", and a following "i" indicates to use powers of 1024 rather than 1000. So "100MB", "100 M", "100000000B", "100000000", and "100000kb" all mean the same thing. Likewise, "1MiB", "1024KiB", "1024 Ki", and "1048576 B" all mean the same thing.

"`tahoe create-node`" generates a tahoe.cfg with "`reserved_space=1G`", but you may wish to raise, lower, or remove the reservation to suit your needs.

`expire.enabled =`

`expire.mode =`

`expire.override_lease_duration =`

`expire.cutoff_date =`

`expire.immutable =`

`expire.mutable =`

These settings control garbage collection, in which the server will delete shares that no longer have an up-to-date lease on them. Please see *Garbage Collection in Tahoe* for full details.

Running A Helper

A "helper" is a regular client node that also offers the "upload helper" service.

`[helper]`

`enabled = (boolean, optional)`

If `True`, the node will run a helper (see *The Tahoe Upload Helper* for details). The helper's contact FURL will be placed in `private/helper.furl`, from which it can be copied to any clients that wish to use it. Clearly nodes should not both run a helper and attempt to use one: do not create `helper.furl` and also define `[helper]enabled` in the same node. The default is `False`.

Running An Introducer

The introducer node uses a different `.tac` file (named "`introducer.tac`"), and pays attention to the `[node]` section, but not the others.

The Introducer node maintains some different state than regular client nodes.

`BASEDIR/private/introducer.furl`

This is generated the first time the introducer node is started, and used again on subsequent runs, to give the introduction service a persistent long-term identity. This file should be published and copied into new client nodes before they are started for the first time.

Other Files in BASEDIR

Some configuration is not kept in `tahoe.cfg`, for the following reasons:

- it doesn't fit into the INI format of `tahoe.cfg` (e.g. `private/servers.yaml`)
- it is generated by the node at startup, e.g. encryption keys. The node never writes to `tahoe.cfg`.
- it is generated by user action, e.g. the "`tahoe create-alias`" command.

In addition, non-configuration persistent state is kept in the node's base directory, next to the configuration knobs.

This section describes these other files.

`private/node.pem`

This contains an SSL private-key certificate. The node generates this the first time it is started, and re-uses it on subsequent runs. This certificate allows the node to have a cryptographically-strong identifier (the Foolscap "TubID"), and to establish secure connections to other nodes.

`storage/`

Nodes that host StorageServers will create this directory to hold shares of files on behalf of other clients. There will be a directory underneath it for each StorageIndex for which this node is holding shares. There is also an "incoming" directory where partially-completed shares are held while they are being received.

`tahoe-client.tac`

This file defines the client, by constructing the actual Client instance each time the node is started. It is used by the "`twistd`" daemonization program (in the `-y` mode), which is run internally by the "`tahoe start`" command. This file is created by the "`tahoe create-node`" or "`tahoe create-client`" commands.

`tahoe-introducer.tac`

This file is used to construct an introducer, and is created by the "`tahoe create-introducer`" command.

`tahoe-stats-gatherer.tac`

This file is used to construct a statistics gatherer, and is created by the "`tahoe create-stats-gatherer`" command.

`private/control.furl`

This file contains a FURL that provides access to a control port on the client node, from which files can be uploaded and downloaded. This file is created with permissions that prevent anyone else from reading it (on operating systems that support such a concept), to insure that only the owner of the client node can use this feature. This port is intended for debugging and testing use.

`private/logport.furl`

This file contains a FURL that provides access to a 'log port' on the client node, from which operational logs can be retrieved. Do not grant logport access to strangers, because occasionally secret information may be placed in the logs.

`private/helper.furl`

If the node is running a helper (for use by other clients), its contact FURL will be placed here. See *The Tahoe Upload Helper* for more details.

`private/root_dir.cap` (optional)

The command-line tools will read a directory cap out of this file and use it, if you don't specify a '–dir-cap' option or if you specify '–dir-cap=root'.

`private/convergence` (automatically generated)

An added secret for encrypting immutable files. Everyone who has this same string in their `private/convergence` file encrypts their immutable files in the same way when uploading them. This causes identical files to "converge" – to share the same storage space since they have identical ciphertext – which conserves space and optimizes upload time, but it also exposes file contents to the possibility of a brute-force attack by people who know that string. In this attack, if the attacker can guess most of the contents of a file, then they can use brute-force to learn the remaining contents.

So the set of people who know your `private/convergence` string is the set of people who converge their storage space with you when you and they upload identical immutable files, and it is also the set of people who could mount such an attack.

The content of the `private/convergence` file is a base-32 encoded string. If the file doesn't exist, then when the Tahoe-LAFS client starts up it will generate a random 256-bit string and write the base-32 encoding of this string into the file. If you want to converge your immutable files with as many people as possible, put the empty string (so that `private/convergence` is a zero-length file).

Additional Introducer Definitions

The `private/introducers.yaml` file defines additional Introducers. The first introducer is defined in `tahoe.cfg`, in `[client] introducer.furl`. To use two or more Introducers, choose a locally-unique "petname" for each one, then define their FURLs in `private/introducers.yaml` like this:

```
introducers:
  petname2:
    furl: FURL2
```

```
petname3:
  furl: FURL3
```

Servers will announce themselves to all configured introducers. Clients will merge the announcements they receive from all introducers. Nothing will re-broadcast an announcement (i.e. telling introducer 2 about something you heard from introducer 1).

If you omit the introducer definitions from both `tahoe.cfg` and `introducers.yaml`, the node will not use an Introducer at all. Such "introducerless" clients must be configured with static servers (described below), or they will not be able to upload and download files.

Static Server Definitions

The `private/servers.yaml` file defines "static servers": those which are not announced through the Introducer. This can also control how we connect to those servers.

Most clients do not need this file. It is only necessary if you want to use servers which are (for some specialized reason) not announced through the Introducer, or to connect to those servers in different ways. You might do this to "freeze" the server list: use the Introducer for a while, then copy all announcements into `servers.yaml`, then stop using the Introducer entirely. Or you might have a private server that you don't want other users to learn about (via the Introducer). Or you might run a local server which is announced to everyone else as a Tor onion address, but which you can connect to directly (via TCP).

The file syntax is YAML, with a top-level dictionary named `storage`. Other items may be added in the future.

The `storage` dictionary takes keys which are server-ids, and values which are dictionaries with two keys: `ann` and `connections`. The `ann` value is a dictionary which will be used in lieu of the introducer announcement, so it can be populated by copying the `ann` dictionary from `NODEDIR/introducer_cache.yaml`.

The server-id can be any string, but ideally you should use the public key as published by the server. Each server displays this as "Node ID:" in the top-right corner of its "WUI" web welcome page. It can also be obtained from other client nodes, which record it as `key_s:` in their `introducer_cache.yaml` file. The format is "v0-" followed by 52 base32 characters like so:

```
v0-c2ng2pbrmxmlwpijn3mr72ckk5fmzk6uxf6nhowyosaubrt6y5mq
```

The `ann` dictionary really only needs one key:

- `anonymous-storage-FURL`: how we connect to the server

(note that other important keys may be added in the future, as Accounting and HTTP-based servers are implemented)

Optional keys include:

- `nickname`: the name of this server, as displayed on the Welcome page server list
- `permutation-seed-base32`: this controls how shares are mapped to servers. This is normally computed from the server-ID, but can be overridden to maintain the mapping for older servers which used to use Foolscap TubIDs as server-IDs. If your selected server-ID cannot be parsed as a public key, it will be hashed to compute the permutation seed. This is fine as long as all clients use the same thing, but if they don't, then your client will disagree with the other clients about which servers should hold each share. This will slow downloads for everybody, and may cause additional work or consume extra storage when repair operations don't converge.
- anything else from the `introducer_cache.yaml` announcement, like `my-version`, which is displayed on the Welcome page server list

For example, a private static server could be defined with a `private/servers.yaml` file like this:

```
storage:
  v0-4uazse3xb6uu5qpkb7tel2bm6bpea4jhuigdhqcuvvse7hugtsia:
    ann:
      nickname: my-server-1
      anonymous-storage-FURL: pb://u33m4y7klhz3bypswqkozwetvabelhxt@tcp:8.8.8.8:51298/
↪eiu2i7p6d6mm4ihmss7ieou5hac3wn6b
```

Or, if you're feeling really lazy:

```
storage:
  my-serverid-1:
    ann:
      anonymous-storage-FURL: pb://u33m4y7klhz3bypswqkozwetvabelhxt@tcp:8.8.8.8:51298/
↪eiu2i7p6d6mm4ihmss7ieou5hac3wn6b
```

Overriding Connection-Handlers for Static Servers

A `connections` entry will override the default connection-handler mapping (as established by `tahoe.cfg [connections]`). This can be used to build a "Tor-mostly client": one which is restricted to use Tor for all connections, except for a few private servers to which normal TCP connections will be made. To override the published announcement (and thus avoid connecting twice to the same server), the server ID must exactly match.

`tahoe.cfg`:

```
[connections]
 # this forces the use of Tor for all "tcp" hints
 tcp = tor
```

`private/servers.yaml`:

```
storage:
  v0-c2ng2pbrmxmlwpijn3mr72ckk5fmzk6uxf6nhowyosaubrt6y5mq:
    ann:
      nickname: my-server-1
      anonymous-storage-FURL: pb://u33m4y7klhz3bypswqkozwetvabelhxt@tcp:10.1.2.
↪3:51298/eiu2i7p6d6mm4ihmss7ieou5hac3wn6b
    connections:
      # this overrides the tcp=tor from tahoe.cfg, for just this server
      tcp: tcp
```

The `connections` table is needed to override the `tcp = tor` mapping that comes from `tahoe.cfg`. Without it, the client would attempt to use Tor to connect to `10.1.2.3`, which would fail because it is a local/non-routeable (RFC1918) address.

Other files

`logs/`

> Each Tahoe-LAFS node creates a directory to hold the log messages produced as the node runs. These logfiles are created and rotated by the "`twistd`" daemonization program, so `logs/twistd.log` will contain the most recent messages, `logs/twistd.log.1` will contain the previous ones, `logs/twistd.log.2` will be older still, and so on. `twistd` rotates logfiles after they grow beyond 1MB in size. If the space consumed by logfiles becomes troublesome, they should be pruned: a cron job to delete all files that were created more than a month ago in this `logs/` directory should be sufficient.

`my_nodeid`

this is written by all nodes after startup, and contains a base32-encoded (i.e. human-readable) NodeID that identifies this specific node. This NodeID is the same string that gets displayed on the web page (in the "which peers am I connected to" list), and the shortened form (the first few characters) is recorded in various log messages.

`access.blacklist`

Gateway nodes may find it necessary to prohibit access to certain files. The web-API has a facility to block access to filecaps by their storage index, returning a 403 "Forbidden" error instead of the original file. For more details, see the "Access Blacklist" section of *The Tahoe REST-ful Web API*.

Example

The following is a sample `tahoe.cfg` file, containing values for some of the keys described in the previous section. Note that this is not a recommended configuration (most of these are not the default values), merely a legal one.

```
[node]
nickname = Bob's Tahoe-LAFS Node
tub.port = tcp:34912
tub.location = tcp:123.45.67.89:8098,tcp:44.55.66.77:8098
web.port = tcp:3456
log_gatherer.furl = pb://soklj4y7eok5c3xkmjeqpw@192.168.69.247:44801/eqpwqtzm
timeout.keepalive = 240
timeout.disconnect = 1800

[client]
introducer.furl = pb://ok45ssoklj4y7eok5c3xkmj@tcp:tahoe.example:44801/ii3uumo
helper.furl = pb://ggti5ssoklj4y7eok5c3xkmj@tcp:helper.tahoe.example:7054/kk8lhr

[storage]
enabled = True
readonly = True
reserved_space = 10000000000

[helper]
enabled = True
```

Old Configuration Files

Tahoe-LAFS releases before v1.3.0 had no `tahoe.cfg` file, and used distinct files for each item. This is no longer supported and if you have configuration in the old format you must manually convert it to the new format for Tahoe-LAFS to detect it. See *Old Configuration Files*.

CHAPTER 6

Tahoe-LAFS Architecture

Overview

(See the docs/specifications directory for more details.)

There are three layers: the key-value store, the file store, and the application.

The lowest layer is the key-value store. The keys are "capabilities" – short ASCII strings – and the values are sequences of data bytes. This data is encrypted and distributed across a number of nodes, such that it will survive the loss of most of the nodes. There are no hard limits on the size of the values, but there may be performance issues with extremely large values (just due to the limitation of network bandwidth). In practice, values as small as a few bytes and as large as tens of gigabytes are in common use.

The middle layer is the decentralized file store: a directed graph in which the intermediate nodes are directories and the leaf nodes are files. The leaf nodes contain only the data – they contain no metadata other than the length in bytes.

The edges leading to leaf nodes have metadata attached to them about the file they point to. Therefore, the same file may be associated with different metadata if it is referred to through different edges.

The top layer consists of the applications using the file store. Allmydata.com used it for a backup service: the application periodically copies files from the local disk onto the decentralized file store. We later provide read-only access to those files, allowing users to recover them. There are several other applications built on top of the Tahoe-LAFS file store (see the RelatedProjects page of the wiki for a list).

The Key-Value Store

The key-value store is implemented by a grid of Tahoe-LAFS storage servers – user-space processes. Tahoe-LAFS storage clients communicate with the storage servers over TCP.

Storage servers hold data in the form of "shares". Shares are encoded pieces of files. There are a configurable number of shares for each file, 10 by default. Normally, each share is stored on a separate server, but in some cases a single server can hold multiple shares of a file.

Nodes learn about each other through an "introducer". Each server connects to the introducer at startup and announces its presence. Each client connects to the introducer at startup, and receives a list of all servers from it. Each client then connects to every server, creating a "bi-clique" topology. In the current release, nodes behind NAT boxes will connect to all nodes that they can open connections to, but they cannot open connections to other nodes behind NAT boxes. Therefore, the more nodes behind NAT boxes, the less the topology resembles the intended bi-clique topology.

The introducer is a Single Point of Failure ("SPoF"), in that clients who never connect to the introducer will be unable to connect to any storage servers, but once a client has been introduced to everybody, it does not need the introducer again until it is restarted. The danger of a SPoF is further reduced in two ways. First, the introducer is defined by a hostname and a private key, which are easy to move to a new host in case the original one suffers an unrecoverable hardware problem. Second, even if the private key is lost, clients can be reconfigured to use a new introducer.

For future releases, we have plans to decentralize introduction, allowing any server to tell a new client about all the others.

File Encoding

When a client stores a file on the grid, it first encrypts the file. It then breaks the encrypted file into small segments, in order to reduce the memory footprint, and to decrease the lag between initiating a download and receiving the first part of the file; for example the lag between hitting "play" and a movie actually starting.

The client then erasure-codes each segment, producing blocks of which only a subset are needed to reconstruct the segment (3 out of 10, with the default settings).

It sends one block from each segment to a given server. The set of blocks on a given server constitutes a "share". Therefore a subset f the shares (3 out of 10, by default) are needed to reconstruct the file.

A hash of the encryption key is used to form the "storage index", which is used for both server selection (described below) and to index shares within the Storage Servers on the selected nodes.

The client computes secure hashes of the ciphertext and of the shares. It uses Merkle Trees so that it is possible to verify the correctness of a subset of the data without requiring all of the data. For example, this allows you to verify the correctness of the first segment of a movie file and then begin playing the movie file in your movie viewer before the entire movie file has been downloaded.

These hashes are stored in a small datastructure named the Capability Extension Block which is stored on the storage servers alongside each share.

The capability contains the encryption key, the hash of the Capability Extension Block, and any encoding parameters necessary to perform the eventual decoding process. For convenience, it also contains the size of the file being stored.

To download, the client that wishes to turn a capability into a sequence of bytes will obtain the blocks from storage servers, use erasure-decoding to turn them into segments of ciphertext, use the decryption key to convert that into plaintext, then emit the plaintext bytes to the output target.

Capabilities

Capabilities to immutable files represent a specific set of bytes. Think of it like a hash function: you feed in a bunch of bytes, and you get out a capability, which is deterministically derived from the input data: changing even one bit of the input data will result in a completely different capability.

Read-only capabilities to mutable files represent the ability to get a set of bytes representing some version of the file, most likely the latest version. Each read-only capability is unique. In fact, each mutable file has a unique public/private key pair created when the mutable file is created, and the read-only capability to that file includes a secure hash of the public key.

Read-write capabilities to mutable files represent the ability to read the file (just like a read-only capability) and also to write a new version of the file, overwriting any extant version. Read-write capabilities are unique – each one includes the secure hash of the private key associated with that mutable file.

The capability provides both "location" and "identification": you can use it to retrieve a set of bytes, and then you can use it to validate ("identify") that these potential bytes are indeed the ones that you were looking for.

The "key-value store" layer doesn't include human-meaningful names. Capabilities sit on the "global+secure" edge of Zooko's Triangle. They are self-authenticating, meaning that nobody can trick you into accepting a file that doesn't match the capability you used to refer to that file. The file store layer (described below) adds human-meaningful names atop the key-value layer.

Server Selection

When a file is uploaded, the encoded shares are sent to some servers. But to which ones? The "server selection" algorithm is used to make this choice.

The storage index is used to consistently-permute the set of all servers nodes (by sorting them by `HASH(storage_index+nodeid)`). Each file gets a different permutation, which (on average) will evenly distribute shares among the grid and avoid hotspots. Each server has announced its available space when it connected to the introducer, and we use that available space information to remove any servers that cannot hold an encoded share for our file. Then we ask some of the servers thus removed if they are already holding any encoded shares for our file; we use this information later. (We ask any servers which are in the first 2*‘‘N‘‘ elements of the permuted list.)

We then use the permuted list of servers to ask each server, in turn, if it will hold a share for us (a share that was not reported as being already present when we talked to the full servers earlier, and that we have not already planned to upload to a different server). We plan to send a share to a server by sending an 'allocate_buckets() query' to the server with the number of that share. Some will say yes they can hold that share, others (those who have become full since they announced their available space) will say no; when a server refuses our request, we take that share to the next server on the list. In the response to allocate_buckets() the server will also inform us of any shares of that file that it already has. We keep going until we run out of shares that need to be stored. At the end of the process, we'll have a table that maps each share number to a server, and then we can begin the encode and push phase, using the table to decide where each share should be sent.

Most of the time, this will result in one share per server, which gives us maximum reliability. If there are fewer writable servers than there are unstored shares, we'll be forced to loop around, eventually giving multiple shares to a single server.

If we have to loop through the node list a second time, we accelerate the query process, by asking each node to hold multiple shares on the second pass. In most cases, this means we'll never send more than two queries to any given node.

If a server is unreachable, or has an error, or refuses to accept any of our shares, we remove it from the permuted list, so we won't query it again for this file. If a server already has shares for the file we're uploading, we add that information to the share-to-server table. This lets us do less work for files which have been uploaded once before, while making sure we still wind up with as many shares as we desire.

Before a file upload is called successful, it has to pass an upload health check. For immutable files, we check to see that a condition called 'servers-of-happiness' is satisfied. When satisfied, 'servers-of-happiness' assures us that enough pieces of the file are distributed across enough servers on the grid to ensure that the availability of the file will not be affected if a few of those servers later fail. For mutable files and directories, we check to see that all of the encoded shares generated during the upload process were successfully placed on the grid. This is a weaker check than 'servers-of-happiness'; it does not consider any information about how the encoded shares are placed on the grid, and cannot detect situations in which all or a majority of the encoded shares generated during the upload process reside on only one storage server. We hope to extend 'servers-of-happiness' to mutable files in a future release of Tahoe-LAFS. If, at the end of the upload process, the appropriate upload health check fails, the upload is considered a failure.

The current defaults use `k` = 3, `servers_of_happiness` = 7, and `N` = 10. `N` = 10 means that we'll try to place 10 shares. `k` = 3 means that we need any three shares to recover the file. `servers_of_happiness` = 7 means that we'll consider an immutable file upload to be successful if we can place shares on enough servers that there are 7 different servers, the correct functioning of any `k` of which guarantee the availability of the immutable file.

`N` = 10 and `k` = 3 means there is a 3.3x expansion factor. On a small grid, you should set `N` about equal to the number of storage servers in your grid; on a large grid, you might set it to something smaller to avoid the overhead of contacting every server to place a file. In either case, you should then set `k` such that `N`/`k` reflects your desired availability goals. The best value for `servers_of_happiness` will depend on how you use Tahoe-LAFS. In a friendnet with a variable number of servers, it might make sense to set it to the smallest number of servers that you expect to have online and accepting shares at any given time. In a stable environment without much server churn, it may make sense to set `servers_of_happiness` = `N`.

When downloading a file, the current version just asks all known servers for any shares they might have. Once it has received enough responses that it knows where to find the needed k shares, it downloads at least the first segment from those servers. This means that it tends to download shares from the fastest servers. If some servers had more than one share, it will continue sending "Do You Have Block" requests to other servers, so that it can download subsequent segments from distinct servers (sorted by their DYHB round-trip times), if possible.

future work

A future release will use the server selection algorithm to reduce the number of queries that must be sent out.

Other peer-node selection algorithms are possible. One earlier version (known as "Tahoe 3") used the permutation to place the nodes around a large ring, distributed the shares evenly around the same ring, then walked clockwise from 0 with a basket. Each time it encountered a share, it put it in the basket, each time it encountered a server, give it as many shares from the basket as they'd accept. This reduced the number of queries (usually to 1) for small grids (where `N` is larger than the number of nodes), but resulted in extremely non-uniform share distribution, which significantly hurt reliability (sometimes the permutation resulted in most of the shares being dumped on a single node).

Another algorithm (known as "denver airport"[1]) uses the permuted hash to decide on an approximate target for each share, then sends lease requests via Chord routing. The request includes the contact information of the uploading node, and asks that the node which eventually accepts the lease should contact the uploader directly. The shares are then transferred over direct connections rather than through

[1] all of these names are derived from the location where they were concocted, in this case in a car ride from Boulder to DEN. To be precise, "Tahoe 1" was an unworkable scheme in which everyone who holds shares for a given file would form a sort of cabal which kept track of all the others, "Tahoe 2" is the first-100-nodes in the permuted hash described in this document, and "Tahoe 3" (or perhaps "Potrero hill 1") was the abandoned ring-with-many-hands approach.

multiple Chord hops. Download uses the same approach. This allows nodes to avoid maintaining a large number of long-term connections, at the expense of complexity and latency.

Swarming Download, Trickling Upload

Because the shares being downloaded are distributed across a large number of nodes, the download process will pull from many of them at the same time. The current encoding parameters require 3 shares to be retrieved for each segment, which means that up to 3 nodes will be used simultaneously. For larger networks, 8-of-22 encoding could be used, meaning 8 nodes can be used simultaneously. This allows the download process to use the sum of the available nodes' upload bandwidths, resulting in downloads that take full advantage of the common 8x disparity between download and upload bandwith on modern ADSL lines.

On the other hand, uploads are hampered by the need to upload encoded shares that are larger than the original data (3.3x larger with the current default encoding parameters), through the slow end of the asymmetric connection. This means that on a typical 8x ADSL line, uploading a file will take about 32 times longer than downloading it again later.

Smaller expansion ratios can reduce this upload penalty, at the expense of reliability (see *Reliability*, below). By using an "upload helper", this penalty is eliminated: the client does a 1x upload of encrypted data to the helper, then the helper performs encoding and pushes the shares to the storage servers. This is an improvement if the helper has significantly higher upload bandwidth than the client, so it makes the most sense for a commercially-run grid for which all of the storage servers are in a colo facility with high interconnect bandwidth. In this case, the helper is placed in the same facility, so the helper-to-storage-server bandwidth is huge.

See *The Tahoe Upload Helper* for details about the upload helper.

The File Store Layer

The "file store" layer is responsible for mapping human-meaningful pathnames (directories and filenames) to pieces of data. The actual bytes inside these files are referenced by capability, but the file store layer is where the directory names, file names, and metadata are kept.

The file store layer is a graph of directories. Each directory contains a table of named children. These children are either other directories or files. All children are referenced by their capability.

A directory has two forms of capability: read-write caps and read-only caps. The table of children inside the directory has a read-write and read-only capability for each child. If you have a read-only capability for a given directory, you will not be able to access the read-write capability of its children. This results in "transitively read-only" directory access.

By having two different capabilities, you can choose which you want to share with someone else. If you create a new directory and share the read-write capability for it with a friend, then you will both be able to modify its contents. If instead you give them the read-only capability, then they will *not* be able to modify the contents. Any capability that you receive can be linked in to any directory that you can modify, so very powerful shared+published directory structures can be built from these components.

This structure enable individual users to have their own personal space, with links to spaces that are shared with specific other users, and other spaces that are globally visible.

Leases, Refreshing, Garbage Collection

When a file or directory in the file store is no longer referenced, the space that its shares occupied on each storage server can be freed, making room for other shares. Tahoe-LAFS uses a garbage collection ("GC") mechanism to

implement this space-reclamation process. Each share has one or more "leases", which are managed by clients who want the file/directory to be retained. The storage server accepts each share for a pre-defined period of time, and is allowed to delete the share if all of the leases are cancelled or allowed to expire.

Garbage collection is not enabled by default: storage servers will not delete shares without being explicitly configured to do so. When GC is enabled, clients are responsible for renewing their leases on a periodic basis at least frequently enough to prevent any of the leases from expiring before the next renewal pass.

See *Garbage Collection in Tahoe* for further information, and for how to configure garbage collection.

File Repairer

Shares may go away because the storage server hosting them has suffered a failure: either temporary downtime (affecting availability of the file), or a permanent data loss (affecting the preservation of the file). Hard drives crash, power supplies explode, coffee spills, and asteroids strike. The goal of a robust distributed file store is to survive these setbacks.

To work against this slow, continual loss of shares, a File Checker is used to periodically count the number of shares still available for any given file. A more extensive form of checking known as the File Verifier can download the ciphertext of the target file and perform integrity checks (using strong hashes) to make sure the data is still intact. When the file is found to have decayed below some threshold, the File Repairer can be used to regenerate and re-upload the missing shares. These processes are conceptually distinct (the repairer is only run if the checker/verifier decides it is necessary), but in practice they will be closely related, and may run in the same process.

The repairer process does not get the full capability of the file to be maintained: it merely gets the "repairer capability" subset, which does not include the decryption key. The File Verifier uses that data to find out which nodes ought to hold shares for this file, and to see if those nodes are still around and willing to provide the data. If the file is not healthy enough, the File Repairer is invoked to download the ciphertext, regenerate any missing shares, and upload them to new nodes. The goal of the File Repairer is to finish up with a full set of `N` shares.

There are a number of engineering issues to be resolved here. The bandwidth, disk IO, and CPU time consumed by the verification/repair process must be balanced against the robustness that it provides to the grid. The nodes involved in repair will have very different access patterns than normal nodes, such that these processes may need to be run on hosts with more memory or network connectivity than usual. The frequency of repair will directly affect the resources consumed. In some cases, verification of multiple files can be performed at the same time, and repair of files can be delegated off to other nodes.

future work

Currently there are two modes of checking on the health of your file: "Checker" simply asks storage servers which shares they have and does nothing to try to verify that they aren't lying. "Verifier" downloads and cryptographically verifies every bit of every share of the file from every server, which costs a lot of network and CPU. A future improvement would be to make a random-sampling verifier which downloads and cryptographically verifies only a few randomly-chosen blocks from each server. This would require much less network and CPU but it could make it extremely unlikely that any sort of corruption – even malicious corruption intended to evade detection – would evade detection. This would be an instance of a cryptographic notion called "Proof of Retrievability". Note that to implement this requires no change to the server or to the cryptographic data structure – with the current data structure and the current protocol it is up to the client which blocks they choose to download, so this would be solely a change in client behavior.

Security

The design goal for this project is that an attacker may be able to deny service (i.e. prevent you from recovering a file that was uploaded earlier) but can accomplish none of the following three attacks:

1. violate confidentiality: the attacker gets to view data to which you have not granted them access
2. violate integrity: the attacker convinces you that the wrong data is actually the data you were intending to retrieve
3. violate unforgeability: the attacker gets to modify a mutable file or directory (either the pathnames or the file contents) to which you have not given them write permission

Integrity (the promise that the downloaded data will match the uploaded data) is provided by the hashes embedded in the capability (for immutable files) or the digital signature (for mutable files). Confidentiality (the promise that the data is only readable by people with the capability) is provided by the encryption key embedded in the capability (for both immutable and mutable files). Data availability (the hope that data which has been uploaded in the past will be downloadable in the future) is provided by the grid, which distributes failures in a way that reduces the correlation between individual node failure and overall file recovery failure, and by the erasure-coding technique used to generate shares.

Many of these security properties depend upon the usual cryptographic assumptions: the resistance of AES and RSA to attack, the resistance of SHA-256 to collision attacks and pre-image attacks, and upon the proximity of 2^-128 and 2^-256 to zero. A break in AES would allow a confidentiality violation, a collision break in SHA-256 would allow a consistency violation, and a break in RSA would allow a mutability violation.

There is no attempt made to provide anonymity, neither of the origin of a piece of data nor the identity of the subsequent downloaders. In general, anyone who already knows the contents of a file will be in a strong position to determine who else is uploading or downloading it. Also, it is quite easy for a sufficiently large coalition of nodes to correlate the set of nodes who are all uploading or downloading the same file, even if the attacker does not know the contents of the file in question.

Also note that the file size and (when convergence is being used) a keyed hash of the plaintext are not protected. Many people can determine the size of the file you are accessing, and if they already know the contents of a given file, they will be able to determine that you are uploading or downloading the same one.

The capability-based security model is used throughout this project. Directory operations are expressed in terms of distinct read- and write- capabilities. Knowing the read-capability of a file is equivalent to the ability to read the corresponding data. The capability to validate the correctness of a file is strictly weaker than the read-capability (possession of read-capability automatically grants you possession of validate-capability, but not vice versa). These capabilities may be expressly delegated (irrevocably) by simply transferring the relevant secrets.

The application layer can provide whatever access model is desired, built on top of this capability access model.

Reliability

File encoding and peer-node selection parameters can be adjusted to achieve different goals. Each choice results in a number of properties; there are many tradeoffs.

First, some terms: the erasure-coding algorithm is described as `k`-out-of-`N` (for this release, the default values are `k` = 3 and `N` = 10). Each grid will have some number of nodes; this number will rise and fall over time as nodes join, drop out, come back, and leave forever. Files are of various sizes, some are popular, others are unpopular. Nodes have various capacities, variable upload/download bandwidths, and network latency. Most of the mathematical models that look at node failure assume some average (and independent) probability 'P' of a given node being available: this can be high (servers tend to be online and available >90% of the time) or low (laptops tend to be turned on for an hour then disappear for several days). Files are encoded in segments of a given maximum size, which affects memory usage.

The ratio of N/k is the "expansion factor". Higher expansion factors improve reliability very quickly (the binomial distribution curve is very sharp), but consumes much more grid capacity. When P=50%, the absolute value of k affects the granularity of the binomial curve (1-out-of-2 is much worse than 50-out-of-100), but high values asymptotically approach a constant (i.e. 500-of-1000 is not much better than 50-of-100). When P is high and the expansion factor is held at a constant, higher values of k and N give much better reliability (for P=99%, 50-out-of-100 is much much better than 5-of-10, roughly 10^50 times better), because there are more shares that can be lost without losing the file.

Likewise, the total number of nodes in the network affects the same granularity: having only one node means a single point of failure, no matter how many copies of the file you make. Independent nodes (with uncorrelated failures) are necessary to hit the mathematical ideals: if you have 100 nodes but they are all in the same office building, then a single power failure will take out all of them at once. Pseudospoofing, also called a "Sybil Attack", is where a single attacker convinces you that they are actually multiple servers, so that you think you are using a large number of independent nodes, but in fact you have a single point of failure (where the attacker turns off all their machines at once). Large grids, with lots of truly independent nodes, will enable the use of lower expansion factors to achieve the same reliability, but will increase overhead because each node needs to know something about every other, and the rate at which nodes come and go will be higher (requiring network maintenance traffic). Also, the File Repairer work will increase with larger grids, although then the job can be distributed out to more nodes.

Higher values of N increase overhead: more shares means more Merkle hashes that must be included with the data, and more nodes to contact to retrieve the shares. Smaller segment sizes reduce memory usage (since each segment must be held in memory while erasure coding runs) and improves "alacrity" (since downloading can validate a smaller piece of data faster, delivering it to the target sooner), but also increase overhead (because more blocks means more Merkle hashes to validate them).

In general, small private grids should work well, but the participants will have to decide between storage overhead and reliability. Large stable grids will be able to reduce the expansion factor down to a bare minimum while still retaining high reliability, but large unstable grids (where nodes are coming and going very quickly) may require more repair/verification bandwidth than actual upload/download traffic.

CHAPTER 7

The Tahoe-LAFS CLI commands

Overview

Tahoe-LAFS provides a single executable named "`tahoe`", which can be used to create and manage client/server nodes, manipulate the file store, and perform several debugging/maintenance tasks. This executable is installed into your virtualenv when you run `pip install tahoe-lafs`.

CLI Command Overview

The "`tahoe`" tool provides access to three categories of commands.

- node management: create a client/server node, start/stop/restart it
- file store manipulation: list files, upload, download, unlink, rename
- debugging: unpack cap-strings, examine share files

To get a list of all commands, just run "`tahoe`" with no additional arguments. "`tahoe --help`" might also provide something useful.

Running "`tahoe --version`" will display a list of version strings, starting with the "allmydata" module (which contains the majority of the Tahoe-LAFS functionality) and including versions for a number of dependent libraries, like Twisted, Foolscap, pycryptopp, and zfec. "`tahoe --version-and-path`" will also show the path from which each library was imported.

On Unix systems, the shell expands filename wildcards (`'*'` and `'?'`) before the program is able to read them, which may produce unexpected results for many `tahoe` comands. We recommend, if you use wildcards, to start the path with "`./`", for example "`tahoe cp -r ./* somewhere:`". This prevents the expanded filename from being interpreted as an option or as an alias, allowing filenames that start with a dash or contain colons to be handled correctly.

On Windows, a single letter followed by a colon is treated as a drive specification rather than an alias (and is invalid unless a local path is allowed in that context). Wildcards cannot be used to specify multiple filenames to `tahoe` on Windows.

Unicode Support

As of Tahoe-LAFS v1.7.0 (v1.8.0 on Windows), the `tahoe` tool supports non-ASCII characters in command lines and output. On Unix, the command-line arguments are assumed to use the character encoding specified by the current locale (usually given by the `LANG` environment variable).

If a name to be output contains control characters or characters that cannot be represented in the encoding used on your terminal, it will be quoted. The quoting scheme used is similar to POSIX shell quoting: in a "double-quoted" string, backslashes introduce escape sequences (like those in Python strings), but in a 'single-quoted' string all characters stand for themselves. This quoting is only used for output, on all operating systems. Your shell interprets any quoting or escapes used on the command line.

Node Management

"`tahoe create-node [NODEDIR]`" is the basic make-a-new-node command. It creates a new directory and populates it with files that will allow the "`tahoe start`" command to use it later on. This command creates nodes that have client functionality (upload/download files), web API services (controlled by the '[node]web.port' configuration), and storage services (unless `--no-storage` is specified).

NODEDIR defaults to `~/.tahoe/`, and newly-created nodes default to publishing a web server on port 3456 (limited to the loopback interface, at 127.0.0.1, to restrict access to other programs on the same host). All of the other "`tahoe`" subcommands use corresponding defaults (with the exception that "`tahoe run`" defaults to running a node in the current directory).

"`tahoe create-client [NODEDIR]`" creates a node with no storage service. That is, it behaves like "`tahoe create-node --no-storage [NODEDIR]`". (This is a change from versions prior to v1.6.0.)

"`tahoe create-introducer [NODEDIR]`" is used to create the Introducer node. This node provides introduction services and nothing else. When started, this node will produce a `private/introducer.furl` file, which should be published to all clients.

"`tahoe run [NODEDIR]`" will start a previously-created node in the foreground.

"`tahoe start [NODEDIR]`" will launch a previously-created node. It will launch the node into the background, using the standard Twisted "`twistd`" daemon-launching tool. On some platforms (including Windows) this command is unable to run a daemon in the background; in that case it behaves in the same way as "`tahoe run`".

"`tahoe stop [NODEDIR]`" will shut down a running node.

"tahoe restart [NODEDIR]" will stop and then restart a running node. This is most often used by developers who have just modified the code and want to start using their changes.

File Store Manipulation

These commands let you exmaine a Tahoe-LAFS file store, providing basic list/upload/download/unlink/rename/mkdir functionality. They can be used as primitives by other scripts. Most of these commands are fairly thin wrappers around web-API calls, which are described in *The Tahoe REST-ful Web API*.

By default, all file store manipulation commands look in ~/.tahoe/ to figure out which Tahoe-LAFS node they should use. When the CLI command makes web-API calls, it will use ~/.tahoe/node.url for this purpose: a running Tahoe-LAFS node that provides a web-API port will write its URL into this file. If you want to use a node on some other host, just create ~/.tahoe/ and copy that node's web-API URL into this file, and the CLI commands will contact that node instead of a local one.

These commands also use a table of "aliases" to figure out which directory they ought to use a starting point. This is explained in more detail below.

Starting Directories

As described in *Tahoe-LAFS Architecture*, the Tahoe-LAFS distributed file store consists of a collection of directories and files, each of which has a "read-cap" or a "write-cap" (also known as a URI). Each directory is simply a table that maps a name to a child file or directory, and this table is turned into a string and stored in a mutable file. The whole set of directory and file "nodes" are connected together into a directed graph.

To use this collection of files and directories, you need to choose a starting point: some specific directory that we will refer to as a "starting directory". For a given starting directory, the "ls [STARTING_DIR]" command would list the contents of this directory, the "ls [STARTING_DIR]/dir1" command would look inside this directory for a child named "dir1" and list its contents, "ls [STARTING_DIR]/dir1/subdir2" would look two levels deep, etc.

Note that there is no real global "root" directory, but instead each starting directory provides a different, possibly overlapping perspective on the graph of files and directories.

Each Tahoe-LAFS node remembers a list of starting points, called "aliases", which are short Unicode strings that stand in for a directory read- or write- cap. They are stored (encoded as UTF-8) in the file NODEDIR/private/aliases. If you use the command line "tahoe ls" without any "[STARTING_DIR]" argument, then it will use the default alias, which is tahoe:, therefore "tahoe ls" has the same effect as "tahoe ls tahoe:". The same goes for the other commands that can reasonably use a default alias: get, put, mkdir, mv, and rm.

For backwards compatibility with Tahoe-LAFS v1.0, if the tahoe: alias is not found in ~/.tahoe/private/aliases, the CLI will use the contents of ~/.tahoe/private/root_dir.cap instead. Tahoe-LAFS v1.0 had only a single starting point, and stored it in this root_dir.cap file, so v1.1 and later will use it if necessary. However, once you've set a tahoe: alias with "tahoe set-alias", that will override anything in the old root_dir.cap file.

The Tahoe-LAFS CLI commands use a similar path syntax to scp and rsync – an optional ALIAS: prefix, followed by the pathname or filename. Some commands (like "tahoe cp") use the lack of an alias to mean that you want to refer to a local file, instead of something from the Tahoe-LAFS file store. Another way to indicate this is to start the pathname with "./", "~/", "~username/", or "/". On Windows, aliases cannot be a single character, so that it is possible to distinguish a path relative to an alias from a path starting with a local drive specifier.

When you're dealing a single starting directory, the tahoe: alias is all you need. But when you want to refer to something that isn't yet attached to the graph rooted at that starting directory, you need to refer to it by its capability. The way to do that is either to use its capability directory as an argument on the command line, or to add an alias to

it, with the "`tahoe add-alias`" command. Once you've added an alias, you can use that alias as an argument to commands.

The best way to get started with Tahoe-LAFS is to create a node, start it, then use the following command to create a new directory and set it as your `tahoe:` alias:

```
tahoe create-alias tahoe
```

After that you can use "`tahoe ls tahoe:`" and "`tahoe cp local.txt tahoe:`", and both will refer to the directory that you've just created.

SECURITY NOTE: For users of shared systems

Another way to achieve the same effect as the above "`tahoe create-alias`" command is:

```
tahoe add-alias tahoe `tahoe mkdir`
```

However, command-line arguments are visible to other users (through the `ps` command or `/proc` filesystem, or the Windows Process Explorer tool), so if you are using a Tahoe-LAFS node on a shared host, your login neighbors will be able to see (and capture) any directory caps that you set up with the "`tahoe add-alias`" command.

The "`tahoe create-alias`" command avoids this problem by creating a new directory and putting the cap into your aliases file for you. Alternatively, you can edit the `NODEDIR/private/aliases` file directly, by adding a line like this:

```
fun:␣
↪URI:DIR2:ovjy4yhylqlfoqg2vcze36dhde:4d4f47qko2xm5g7osgo2yyidi5m4muyo2vjjy53q4vjju2u55mfa
```

By entering the dircap through the editor, the command-line arguments are bypassed, and other users will not be able to see them. Once you've added the alias, no other secrets are passed through the command line, so this vulnerability becomes less significant: they can still see your filenames and other arguments you type there, but not the caps that Tahoe-LAFS uses to permit access to your files and directories.

Command Syntax Summary

```
tahoe add-alias ALIAS[:]  DIRCAP

tahoe create-alias ALIAS[:]

tahoe list-aliases

tahoe mkdir

tahoe mkdir PATH

tahoe ls [PATH]

tahoe webopen [PATH]

tahoe put [--mutable] [FROMLOCAL|-]

tahoe put [--mutable] FROMLOCAL|- TOPATH

tahoe put [FROMLOCAL|-] mutable-file-writecap

tahoe get FROMPATH [TOLOCAL|-]

tahoe cp [-r] FROMPATH TOPATH

tahoe rm PATH
```

`tahoe mv FROMPATH TOPATH`

`tahoe ln FROMPATH TOPATH`

`tahoe backup FROMLOCAL TOPATH`

In these summaries, `PATH`, `TOPATH` or `FROMPATH` can be one of:

- `[SUBDIRS/]FILENAME` for a path relative to the default `tahoe:` alias;
- `ALIAS:[SUBDIRS/]FILENAME` for a path relative to another alias;
- `DIRCAP/[SUBDIRS/]FILENAME` or `DIRCAP:./[SUBDIRS/]FILENAME` for a path relative to a directory cap.

See *CLI Command Overview* above for information on using wildcards with local paths, and different treatment of colons between Unix and Windows.

`FROMLOCAL` or `TOLOCAL` is a path in the local filesystem.

Command Examples

`tahoe add-alias ALIAS[:] DIRCAP`

An example would be:

```
tahoe add-alias fun␣
↪URI:DIR2:ovjy4yhylqlfoqg2vcze36dhde:4d4f47qko2xm5g7osgo2yyidi5m4muyo2vjjy53q4vjju2u55mfa
```

This creates an alias `fun:` and configures it to use the given directory cap. Once this is done, "`tahoe ls fun:`" will list the contents of this directory. Use "`tahoe add-alias tahoe DIRCAP`" to set the contents of the default `tahoe:` alias.

Since Tahoe-LAFS v1.8.2, the alias name can be given with or without the trailing colon.

On Windows, the alias should not be a single character, because it would be confused with the drive letter of a local path.

`tahoe create-alias fun`

This combines "`tahoe mkdir`" and "`tahoe add-alias`" into a single step.

`tahoe list-aliases`

This displays a table of all configured aliases.

`tahoe mkdir`

This creates a new empty unlinked directory, and prints its write-cap to stdout. The new directory is not attached to anything else.

`tahoe mkdir subdir`

`tahoe mkdir /subdir`

This creates a new empty directory and attaches it below the root directory of the default `tahoe:` alias with the name "`subdir`".

`tahoe ls`

`tahoe ls /`

`tahoe ls tahoe:`

`tahoe ls tahoe:/`

All four list the root directory of the default `tahoe:` alias.

```
tahoe ls subdir
```

This lists a subdirectory of your file store.

```
tahoe webopen
tahoe webopen tahoe:
tahoe webopen tahoe:subdir/
tahoe webopen subdir/
```

This uses the python 'webbrowser' module to cause a local web browser to open to the web page for the given directory. This page offers interfaces to add, download, rename, and unlink files and subdirectories in that directory. If no alias or path is given, this command opens the root directory of the default `tahoe:` alias.

```
tahoe put file.txt
tahoe put ./file.txt
tahoe put /tmp/file.txt
tahoe put ~/file.txt
```

These upload the local file into the grid, and prints the new read-cap to stdout. The uploaded file is not attached to any directory. All one-argument forms of "`tahoe put`" perform an unlinked upload.

```
tahoe put -
tahoe put
```

These also perform an unlinked upload, but the data to be uploaded is taken from stdin.

```
tahoe put file.txt uploaded.txt
tahoe put file.txt tahoe:uploaded.txt
```

These upload the local file and add it to your `tahoe:` root with the name "`uploaded.txt`".

```
tahoe put file.txt subdir/foo.txt
tahoe put - subdir/foo.txt
tahoe put file.txt tahoe:subdir/foo.txt
tahoe put file.txt DIRCAP/foo.txt
tahoe put file.txt DIRCAP/subdir/foo.txt
```

These upload the named file and attach them to a subdirectory of the given root directory, under the name "`foo.txt`". When a directory write-cap is given, you can use either `/` (as shown above) or `:./` to separate it from the following path. When the source file is named "`-`", the contents are taken from stdin.

```
tahoe put file.txt --mutable
```

Create a new (SDMF) mutable file, fill it with the contents of `file.txt`, and print the new write-cap to stdout.

```
tahoe put file.txt MUTABLE-FILE-WRITECAP
```

Replace the contents of the given mutable file with the contents of `file.txt` and print the same write-cap to stdout.

```
tahoe cp file.txt tahoe:uploaded.txt

tahoe cp file.txt tahoe:

tahoe cp file.txt tahoe:/

tahoe cp ./file.txt tahoe:
```

These upload the local file and add it to your `tahoe:` root with the name "`uploaded.txt`".

```
tahoe cp tahoe:uploaded.txt downloaded.txt

tahoe cp tahoe:uploaded.txt ./downloaded.txt

tahoe cp tahoe:uploaded.txt /tmp/downloaded.txt

tahoe cp tahoe:uploaded.txt ~/downloaded.txt
```

This downloads the named file from your `tahoe:` root, and puts the result on your local filesystem.

```
tahoe cp tahoe:uploaded.txt fun:stuff.txt
```

This copies a file from your `tahoe:` root to a different directory, set up earlier with "`tahoe add-alias fun DIRCAP`" or "`tahoe create-alias fun`".

```
tahoe cp -r ~/my_dir/ tahoe:
```

This copies the folder `~/my_dir/` and all its children to the grid, creating the new folder `tahoe:my_dir`. Note that the trailing slash is not required: all source arguments which are directories will be copied into new subdirectories of the target.

The behavior of `tahoe cp`, like the regular UNIX `/bin/cp`, is subtly different depending upon the exact form of the arguments. In particular:

- Trailing slashes indicate directories, but are not required.
- If the target object does not already exist: * and if the source is a single file, it will be copied into the target; * otherwise, the target will be created as a directory.
- If there are multiple sources, the target must be a directory.
- If the target is a pre-existing file, the source must be a single file.
- If the target is a directory, each source must be a named file, a named directory, or an unnamed directory. It is not possible to copy an unnamed file (e.g. a raw filecap) into a directory, as there is no way to know what the new file should be named.

```
tahoe unlink uploaded.txt

tahoe unlink tahoe:uploaded.txt
```

This unlinks a file from your `tahoe:` root (that is, causes there to no longer be an entry `uploaded.txt` in the root directory that points to it). Note that this does not delete the file from the grid. For backward compatibility, `tahoe rm` is accepted as a synonym for `tahoe unlink`.

```
tahoe mv uploaded.txt renamed.txt

tahoe mv tahoe:uploaded.txt tahoe:renamed.txt
```

These rename a file within your `tahoe:` root directory.

```
tahoe mv uploaded.txt fun:

tahoe mv tahoe:uploaded.txt fun:

tahoe mv tahoe:uploaded.txt fun:uploaded.txt
```

These move a file from your `tahoe:` root directory to the directory set up earlier with "`tahoe add-alias fun DIRCAP`" or "`tahoe create-alias fun`".

`tahoe backup ~ work:backups`

This command performs a versioned backup of every file and directory underneath your "~" home directory, placing an immutable timestamped snapshot in e.g. `work:backups/Archives/2009-02-06_04:00:05Z/` (note that the timestamp is in UTC, hence the "Z" suffix), and a link to the latest snapshot in work:backups/Latest/ . This command uses a small SQLite database known as the "backupdb", stored in `~/.tahoe/private/backupdb.sqlite`, to remember which local files have been backed up already, and will avoid uploading files that have already been backed up (except occasionally that will randomly upload them again if it has been awhile since had last been uploaded, just to make sure that the copy of it on the server is still good). It compares timestamps and filesizes when making this comparison. It also re-uses existing directories which have identical contents. This lets it run faster and reduces the number of directories created.

If you reconfigure your client node to switch to a different grid, you should delete the stale backupdb.sqlite file, to force "`tahoe backup`" to upload all files to the new grid.

The fact that "tahoe backup" checks timestamps on your local files and skips ones that don't appear to have been changed is one of the major differences between "tahoe backup" and "tahoe cp -r". The other major difference is that "tahoe backup" keeps links to all of the versions that have been uploaded to the grid, so you can navigate among old versions stored in the grid. In contrast, "tahoe cp -r" unlinks the previous version from the grid directory and links the new version into place, so unless you have a link to the older version stored somewhere else, you'll never be able to get back to it.

`tahoe backup --exclude=*~ ~ work:backups`

Same as above, but this time the backup process will ignore any filename that will end with '~'. `--exclude` will accept any standard Unix shell-style wildcards, as implemented by the Python fnmatch module. You may give multiple `--exclude` options. Please pay attention that the pattern will be matched against any level of the directory tree; it's still impossible to specify absolute path exclusions.

`tahoe backup --exclude-from=/path/to/filename ~ work:backups`

`--exclude-from` is similar to `--exclude`, but reads exclusion patterns from `/path/to/filename`, one per line.

`tahoe backup --exclude-vcs ~ work:backups`

This command will ignore any file or directory name known to be used by version control systems to store metadata. The excluded names are:

- CVS
- RCS
- SCCS
- .git
- .gitignore
- .cvsignore
- .svn
- .arch-ids
- {arch}
- =RELEASE-ID
- =meta-update

- =update
- .bzr
- .bzrignore
- .bzrtags
- .hg
- .hgignore
- _darcs

Storage Grid Maintenance

`tahoe manifest tahoe:`

`tahoe manifest --storage-index tahoe:`

`tahoe manifest --verify-cap tahoe:`

`tahoe manifest --repair-cap tahoe:`

`tahoe manifest --raw tahoe:`

This performs a recursive walk of the given directory, visiting every file and directory that can be reached from that point. It then emits one line to stdout for each object it encounters.

The default behavior is to print the access cap string (like `URI:CHK:..` or `URI:DIR2:..`), followed by a space, followed by the full path name.

If `--storage-index` is added, each line will instead contain the object's storage index. This (string) value is useful to determine which share files (on the server) are associated with this directory tree. The `--verify-cap` and `--repair-cap` options are similar, but emit a verify-cap and repair-cap, respectively. If `--raw` is provided instead, the output will be a JSON-encoded dictionary that includes keys for pathnames, storage index strings, and cap strings. The last line of the `--raw` output will be a JSON encoded deep-stats dictionary.

`tahoe stats tahoe:`

This performs a recursive walk of the given directory, visiting every file and directory that can be reached from that point. It gathers statistics on the sizes of the objects it encounters, and prints a summary to stdout.

Debugging

For a list of all debugging commands, use "`tahoe debug`". For more detailed help on any of these commands, use "`tahoe debug COMMAND --help`".

"`tahoe debug find-shares STORAGEINDEX NODEDIRS..`" will look through one or more storage nodes for the share files that are providing storage for the given storage index.

"`tahoe debug catalog-shares NODEDIRS..`" will look through one or more storage nodes and locate every single share they contain. It produces a report on stdout with one line per share, describing what kind of share it is, the storage index, the size of the file is used for, etc. It may be useful to concatenate these reports from all storage hosts and use it to look for anomalies.

"`tahoe debug dump-share SHAREFILE`" will take the name of a single share file (as found by "`tahoe find-shares`") and print a summary of its contents to stdout. This includes a list of leases, summaries of the

hash tree, and information from the UEB (URI Extension Block). For mutable file shares, it will describe which version (seqnum and root-hash) is being stored in this share.

"`tahoe debug dump-cap CAP`" will take any Tahoe-LAFS URI and unpack it into separate pieces. The most useful aspect of this command is to reveal the storage index for any given URI. This can be used to locate the share files that are holding the encoded+encrypted data for this file.

"`tahoe debug corrupt-share SHAREFILE`" will flip a bit in the given sharefile. This can be used to test the client-side verification/repair code. Obviously, this command should not be used during normal operation.

CHAPTER 8

The Tahoe REST-ful Web API

Enabling the web-API port

Every Tahoe node is capable of running a built-in HTTP server. To enable this, just write a port number into the "[node]web.port" line of your node's tahoe.cfg file. For example, writing "web.port = 3456" into the "[node]" section of $NODEDIR/tahoe.cfg will cause the node to run a webserver on port 3456.

This string is actually a Twisted "strports" specification, meaning you can get more control over the interface to which the server binds by supplying additional arguments. For more details, see the documentation on twisted.application.strports.

Writing "tcp:3456:interface=127.0.0.1" into the web.port line does the same but binds to the loopback interface, ensuring that only the programs on the local host can connect. Using "ssl:3456:privateKey=mykey.pem:certKey=cert.pem" runs an SSL server.

This webport can be set when the node is created by passing a –webport option to the 'tahoe create-node' command. By default, the node listens on port 3456, on the loopback (127.0.0.1) interface.

Basic Concepts: GET, PUT, DELETE, POST

As described in *Tahoe-LAFS Architecture*, each file and directory in a Tahoe-LAFS file store is referenced by an identifier that combines the designation of the object with the authority to do something with it (such as read or modify the contents). This identifier is called a "read-cap" or "write-cap", depending upon whether it enables read-only or read-write access. These "caps" are also referred to as URIs (which may be confusing because they are not currently RFC3986-compliant URIs).

The Tahoe web-based API is "REST-ful", meaning it implements the concepts of "REpresentational State Transfer": the original scheme by which the World Wide Web was intended to work. Each object (file or directory) is referenced by a URL that includes the read- or write- cap. HTTP methods (GET, PUT, and DELETE) are used to manipulate these objects. You can think of the URL as a noun, and the method as a verb.

In REST, the GET method is used to retrieve information about an object, or to retrieve some representation of the object itself. When the object is a file, the basic GET method will simply return the contents of that file. Other variations (generally implemented by adding query parameters to the URL) will return information about the object, such as metadata. GET operations are required to have no side-effects.

PUT is used to upload new objects into the file store, or to replace an existing link or the contents of a mutable file. DELETE is used to unlink objects from directories. Both PUT and DELETE are required to be idempotent: performing the same operation multiple times must have the same side-effects as only performing it once.

POST is used for more complicated actions that cannot be expressed as a GET, PUT, or DELETE. POST operations can be thought of as a method call: sending some message to the object referenced by the URL. In Tahoe, POST is also used for operations that must be triggered by an HTML form (including upload and unlinking), because otherwise

a regular web browser has no way to accomplish these tasks. In general, everything that can be done with a PUT or DELETE can also be done with a POST.

Tahoe-LAFS' web API is designed for two different kinds of consumer. The first is a program that needs to manipulate the file store. Such programs are expected to use the RESTful interface described above. The second is a human using a standard web browser to work with the file store. This user is presented with a series of HTML pages with links to download files, and forms that use POST actions to upload, rename, and unlink files.

When an error occurs, the HTTP response code will be set to an appropriate 400-series code (like 404 Not Found for an unknown childname, or 400 Bad Request when the parameters to a web-API operation are invalid), and the HTTP response body will usually contain a few lines of explanation as to the cause of the error and possible responses. Unusual exceptions may result in a 500 Internal Server Error as a catch-all, with a default response body containing a Nevow-generated HTML-ized representation of the Python exception stack trace that caused the problem. CLI programs which want to copy the response body to stderr should provide an "Accept: text/plain" header to their requests to get a plain text stack trace instead. If the Accept header contains `*/*`, or `text/*`, or text/html (or if there is no Accept header), HTML tracebacks will be generated.

URLs

Tahoe uses a variety of read- and write- caps to identify files and directories. The most common of these is the "immutable file read-cap", which is used for most uploaded files. These read-caps look like the following:

```
URI:CHK:ime6pvkaxuetdfah2p2f35pe54:4btz54xk3tew6nd4y2ojpxj4m6wxjqqlwnztgre6gnjgtucd5r4a:3:10:202
```

The next most common is a "directory write-cap", which provides both read and write access to a directory, and look like this:

```
URI:DIR2:djrdkfawoqihigoett4g6auz6a:jx5mplfpwexnoqff7y5e4zjus4lidm76dcuarpct7cckorh2dpgq
```

There are also "directory read-caps", which start with "URI:DIR2-RO:", and give read-only access to a directory. Finally there are also mutable file read- and write- caps, which start with "URI:SSK", and give access to mutable files.

(Later versions of Tahoe will make these strings shorter, and will remove the unfortunate colons, which must be escaped when these caps are embedded in URLs.)

To refer to any Tahoe object through the web API, you simply need to combine a prefix (which indicates the HTTP server to use) with the cap (which indicates which object inside that server to access). Since the default Tahoe webport is 3456, the most common prefix is one that will use a local node listening on this port:

```
http://127.0.0.1:3456/uri/ + $CAP
```

So, to access the directory named above, the URL would be:

```
http://127.0.0.1:3456/uri/URI%3ADIR2%3Adjrdkfawoqihigoett4g6auz6a
↪%3Ajx5mplfpwexnoqff7y5e4zjus4lidm76dcuarpct7cckorh2dpgq/
```

(note that the colons in the directory-cap are url-encoded into "%3A" sequences).

Likewise, to access the file named above, use:

```
http://127.0.0.1:3456/uri/URI%3ACHK%3Aime6pvkaxuetdfah2p2f35pe54
↪%3A4btz54xk3tew6nd4y2ojpxj4m6wxjqqlwnztgre6gnjgtucd5r4a%3A3%3A10%3A202
```

In the rest of this document, we'll use "$DIRCAP" as shorthand for a read-cap or write-cap that refers to a directory, and "$FILECAP" to abbreviate a cap that refers to a file (whether mutable or immutable). So those URLs above can be abbreviated as:

```
http://127.0.0.1:3456/uri/$DIRCAP/
http://127.0.0.1:3456/uri/$FILECAP
```

The operation summaries below will abbreviate these further, by eliding the server prefix. They will be displayed like this:

```
/uri/$DIRCAP/
/uri/$FILECAP
```

/cap can be used as a synonym for /uri. If interoperability with older web-API servers is required, /uri should be used.

Child Lookup

Tahoe directories contain named child entries, just like directories in a regular local filesystem. These child entries, called "dirnodes", consist of a name, metadata, a write slot, and a read slot. The write and read slots normally contain a write-cap and read-cap referring to the same object, which can be either a file or a subdirectory. The write slot may be empty (actually, both may be empty, but that is unusual).

If you have a Tahoe URL that refers to a directory, and want to reference a named child inside it, just append the child name to the URL. For example, if our sample directory contains a file named "welcome.txt", we can refer to that file with:

```
http://127.0.0.1:3456/uri/$DIRCAP/welcome.txt
```

(or http://127.0.0.1:3456/uri/URI%3ADIR2%3Adjrdkfawoqihigoett4g6auz6a%3Ajx5mplfpwexnoqff7y5e4zjus4lidm76dcuarpct7cckorh welcome.txt)

Multiple levels of subdirectories can be handled this way:

```
http://127.0.0.1:3456/uri/$DIRCAP/tahoe-source/docs/architecture.rst
```

In this document, when we need to refer to a URL that references a file using this child-of-some-directory format, we'll use the following string:

```
/uri/$DIRCAP/[SUBDIRS../]FILENAME
```

The "[SUBDIRS../]" part means that there are zero or more (optional) subdirectory names in the middle of the URL. The "FILENAME" at the end means that this whole URL refers to a file of some sort, rather than to a directory.

When we need to refer specifically to a directory in this way, we'll write:

```
/uri/$DIRCAP/[SUBDIRS../]SUBDIR
```

Note that all components of pathnames in URLs are required to be UTF-8 encoded, so "resume.doc" (with an acute accent on both E's) would be accessed with:

```
http://127.0.0.1:3456/uri/$DIRCAP/r%C3%A9sum%C3%A9.doc
```

Also note that the filenames inside upload POST forms are interpreted using whatever character set was provided in the conventional '_charset' field, and defaults to UTF-8 if not otherwise specified. The JSON representation of each directory contains native Unicode strings. Tahoe directories are specified to contain Unicode filenames, and cannot contain binary strings that are not representable as such.

All Tahoe operations that refer to existing files or directories must include a suitable read- or write- cap in the URL: the web-API server won't add one for you. If you don't know the cap, you can't access the file. This allows the security properties of Tahoe caps to be extended across the web-API interface.

Slow Operations, Progress, and Cancelling

Certain operations can be expected to take a long time. The "t=deep-check", described below, will recursively visit every file and directory reachable from a given starting point, which can take minutes or even hours for extremely large directory structures. A single long-running HTTP request is a fragile thing: proxies, NAT boxes, browsers, and users may all grow impatient with waiting and give up on the connection.

For this reason, long-running operations have an "operation handle", which can be used to poll for status/progress messages while the operation proceeds. This handle can also be used to cancel the operation. These handles are created by the client, and passed in as a an "ophandle=" query argument to the POST or PUT request which starts the operation. The following operations can then be used to retrieve status:

`GET /operations/$HANDLE?output=HTML (with or without t=status)`

`GET /operations/$HANDLE?output=JSON (same)`

These two retrieve the current status of the given operation. Each operation presents a different sort of information, but in general the page retrieved will indicate:

- whether the operation is complete, or if it is still running
- how much of the operation is complete, and how much is left, if possible

Note that the final status output can be quite large: a deep-manifest of a directory structure with 300k directories and 200k unique files is about 275MB of JSON, and might take two minutes to generate. For this reason, the full status is not provided until the operation has completed.

The HTML form will include a meta-refresh tag, which will cause a regular web browser to reload the status page about 60 seconds later. This tag will be removed once the operation has completed.

There may be more status information available under /operations/$HANDLE/$ETC : i.e., the handle forms the root of a URL space.

`POST /operations/$HANDLE?t=cancel`

This terminates the operation, and returns an HTML page explaining what was cancelled. If the operation handle has already expired (see below), this POST will return a 404, which indicates that the operation is no longer running (either it was completed or terminated). The response body will be the same as a GET /operations/$HANDLE on this operation handle, and the handle will be expired immediately afterwards.

The operation handle will eventually expire, to avoid consuming an unbounded amount of memory. The handle's time-to-live can be reset at any time, by passing a retain-for= argument (with a count of seconds) to either the initial POST that starts the operation, or the subsequent GET request which asks about the operation. For example, if a 'GET /operations/$HANDLE?output=JSON&retain-for=600' query is performed, the handle will remain active for 600 seconds (10 minutes) after the GET was received.

In addition, if the GET includes a release-after-complete=True argument, and the operation has completed, the operation handle will be released immediately.

If a retain-for= argument is not used, the default handle lifetimes are:

- handles will remain valid at least until their operation finishes
- uncollected handles for finished operations (i.e. handles for operations that have finished but for which the GET page has not been accessed since completion) will remain valid for four days, or for the total time consumed by the operation, whichever is greater.
- collected handles (i.e. the GET page has been retrieved at least once since the operation completed) will remain valid for one day.

Many "slow" operations can begin to use unacceptable amounts of memory when operating on large directory structures. The memory usage increases when the ophandle is polled, as the results must be copied into a JSON string, sent

over the wire, then parsed by a client. So, as an alternative, many "slow" operations have streaming equivalents. These equivalents do not use operation handles. Instead, they emit line-oriented status results immediately. Client code can cancel the operation by simply closing the HTTP connection.

Programmatic Operations

Now that we know how to build URLs that refer to files and directories in a Tahoe-LAFS file store, what sorts of operations can we do with those URLs? This section contains a catalog of GET, PUT, DELETE, and POST operations that can be performed on these URLs. This set of operations are aimed at programs that use HTTP to communicate with a Tahoe node. A later section describes operations that are intended for web browsers.

Reading a File

`GET /uri/$FILECAP`

`GET /uri/$DIRCAP/[SUBDIRS../]FILENAME`

This will retrieve the contents of the given file. The HTTP response body will contain the sequence of bytes that make up the file.

The "Range:" header can be used to restrict which portions of the file are returned (see RFC 2616 section 14.35.1 "Byte Ranges"), however Tahoe only supports a single "bytes" range and never provides a `multipart/byteranges` response. An attempt to begin a read past the end of the file will provoke a 416 Requested Range Not Satisfiable error, but normal overruns (reads which start at the beginning or middle and go beyond the end) are simply truncated.

To view files in a web browser, you may want more control over the Content-Type and Content-Disposition headers. Please see the next section "Browser Operations", for details on how to modify these URLs for that purpose.

Writing/Uploading a File

`PUT /uri/$FILECAP`

`PUT /uri/$DIRCAP/[SUBDIRS../]FILENAME`

Upload a file, using the data from the HTTP request body, and add whatever child links and subdirectories are necessary to make the file available at the given location. Once this operation succeeds, a GET on the same URL will retrieve the same contents that were just uploaded. This will create any necessary intermediate subdirectories.

To use the /uri/$FILECAP form, $FILECAP must be a write-cap for a mutable file.

In the /uri/$DIRCAP/[SUBDIRS../]FILENAME form, if the target file is a writeable mutable file, that file's contents will be overwritten in-place. If it is a read-cap for a mutable file, an error will occur. If it is an immutable file, the old file will be discarded, and a new one will be put in its place. If the target file is a writable mutable file, you may also specify an "offset" parameter – a byte offset that determines where in the mutable file the data from the HTTP request body is placed. This operation is relatively efficient for MDMF mutable files, and is relatively inefficient (but still supported) for SDMF mutable files. If no offset parameter is specified, then the entire file is replaced with the data from the HTTP request body. For an immutable file, the "offset" parameter is not valid.

When creating a new file, you can control the type of file created by specifying a format= argument in the query string. format=MDMF creates an MDMF mutable file. format=SDMF creates an SDMF mutable

file. format=CHK creates an immutable file. The value of the format argument is case-insensitive. If no format is specified, the newly-created file will be immutable (but see below).

For compatibility with previous versions of Tahoe-LAFS, the web-API will also accept a mutable=true argument in the query string. If mutable=true is given, then the new file will be mutable, and its format will be the default mutable file format, as configured by the [client]mutable.format option of tahoe.cfg on the Tahoe-LAFS node hosting the webapi server. Use of mutable=true is discouraged; new code should use format= instead of mutable=true (unless it needs to be compatible with web-API servers older than v1.9.0). If neither format= nor mutable=true are given, the newly-created file will be immutable.

This returns the file-cap of the resulting file. If a new file was created by this method, the HTTP response code (as dictated by rfc2616) will be set to 201 CREATED. If an existing file was replaced or modified, the response code will be 200 OK.

Note that the 'curl -T localfile http://127.0.0.1:3456/uri/\protect\T1\textdollarDIRCAP/foo.txt' command can be used to invoke this operation.

`PUT /uri`

This uploads a file, and produces a file-cap for the contents, but does not attach the file into the file store. No directories will be modified by this operation. The file-cap is returned as the body of the HTTP response.

This method accepts format= and mutable=true as query string arguments, and interprets those arguments in the same way as the linked forms of PUT described immediately above.

Creating a New Directory

`POST /uri?t=mkdir`

`PUT /uri?t=mkdir`

Create a new empty directory and return its write-cap as the HTTP response body. This does not make the newly created directory visible from the file store. The "PUT" operation is provided for backwards compatibility: new code should use POST.

This supports a format= argument in the query string. The format= argument, if specified, controls the format of the directory. format=MDMF indicates that the directory should be stored as an MDMF file; format=SDMF indicates that the directory should be stored as an SDMF file. The value of the format= argument is case-insensitive. If no format= argument is given, the directory's format is determined by the default mutable file format, as configured on the Tahoe-LAFS node responding to the request.

`POST /uri?t=mkdir-with-children`

Create a new directory, populated with a set of child nodes, and return its write-cap as the HTTP response body. The new directory is not attached to any other directory: the returned write-cap is the only reference to it.

The format of the directory can be controlled with the format= argument in the query string, as described above.

Initial children are provided as the body of the POST form (this is more efficient than doing separate mkdir and set_children operations). If the body is empty, the new directory will be empty. If not empty, the body will be interpreted as a UTF-8 JSON-encoded dictionary of children with which the new directory should be populated, using the same format as would be returned in the 'children' value of the t=json GET request, described below. Each dictionary key should be a child name, and each value should be a list of [TYPE, PROPDICT], where PROPDICT contains "rw_uri", "ro_uri", and "metadata" keys (all others are ignored). For example, the PUT request body could be:

```
{
  "Fran\u00e7ais": [ "filenode", {
      "ro_uri": "URI:CHK:...",
      "metadata": {
        "ctime": 1202777696.7564139,
        "mtime": 1202777696.7564139,
        "tahoe": {
          "linkcrtime": 1202777696.7564139,
          "linkmotime": 1202777696.7564139
          } } } ],
  "subdir":  [ "dirnode", {
      "rw_uri": "URI:DIR2:...",
      "ro_uri": "URI:DIR2-RO:...",
      "metadata": {
        "ctime": 1202778102.7589991,
        "mtime": 1202778111.2160511,
        "tahoe": {
          "linkcrtime": 1202777696.7564139,
          "linkmotime": 1202777696.7564139
        } } } ]
}
```

For forward-compatibility, a mutable directory can also contain caps in a format that is unknown to the web-API server. When such caps are retrieved from a mutable directory in a "ro_uri" field, they will be prefixed with the string "ro.", indicating that they must not be decoded without checking that they are read-only. The "ro." prefix must not be stripped off without performing this check. (Future versions of the web-API server will perform it where necessary.)

If both the "rw_uri" and "ro_uri" fields are present in a given PROPDICT, and the web-API server recognizes the rw_uri as a write cap, then it will reset the ro_uri to the corresponding read cap and discard the original contents of ro_uri (in order to ensure that the two caps correspond to the same object and that the ro_uri is in fact read-only). However this may not happen for caps in a format unknown to the web-API server. Therefore, when writing a directory the web-API client should ensure that the contents of "rw_uri" and "ro_uri" for a given PROPDICT are a consistent (write cap, read cap) pair if possible. If the web-API client only has one cap and does not know whether it is a write cap or read cap, then it is acceptable to set "rw_uri" to that cap and omit "ro_uri". The client must not put a write cap into a "ro_uri" field.

The metadata may have a "no-write" field. If this is set to true in the metadata of a link, it will not be possible to open that link for writing via the SFTP frontend; see *Tahoe-LAFS SFTP and FTP Frontends* for details. Also, if the "no-write" field is set to true in the metadata of a link to a mutable child, it will cause the link to be diminished to read-only.

Note that the web-API-using client application must not provide the "Content-Type: multipart/form-data" header that usually accompanies HTML form submissions, since the body is not formatted this way. Doing so will cause a server error as the lower-level code misparses the request body.

Child file names should each be expressed as a Unicode string, then used as keys of the dictionary. The dictionary should then be converted into JSON, and the resulting string encoded into UTF-8. This UTF-8 bytestring should then be used as the POST body.

`POST /uri?t=mkdir-immutable`

Like t=mkdir-with-children above, but the new directory will be deep-immutable. This means that the directory itself is immutable, and that it can only contain objects that are treated as being deep-immutable, like immutable files, literal files, and deep-immutable directories.

For forward-compatibility, a deep-immutable directory can also contain caps in a format that is unknown to the web-API server. When such caps are retrieved from a deep-immutable directory in a "ro_uri" field, they will be prefixed with the string "imm.", indicating that they must not be decoded without checking

that they are immutable. The "imm." prefix must not be stripped off without performing this check. (Future versions of the web-API server will perform it where necessary.)

The cap for each child may be given either in the "rw_uri" or "ro_uri" field of the PROPDICT (not both). If a cap is given in the "rw_uri" field, then the web-API server will check that it is an immutable read-cap of a *known* format, and give an error if it is not. If a cap is given in the "ro_uri" field, then the web-API server will still check whether known caps are immutable, but for unknown caps it will simply assume that the cap can be stored, as described above. Note that an attacker would be able to store any cap in an immutable directory, so this check when creating the directory is only to help non-malicious clients to avoid accidentally giving away more authority than intended.

A non-empty request body is mandatory, since after the directory is created, it will not be possible to add more children to it.

`POST /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=mkdir`

`PUT /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=mkdir`

Create new directories as necessary to make sure that the named target ($DIRCAP/SUBDIRS../SUBDIR) is a directory. This will create additional intermediate mutable directories as necessary. If the named target directory already exists, this will make no changes to it.

If the final directory is created, it will be empty.

This accepts a format= argument in the query string, which controls the format of the named target directory, if it does not already exist. format= is interpreted in the same way as in the POST /uri?t=mkdir form. Note that format= only controls the format of the named target directory; intermediate directories, if created, are created based on the default mutable type, as configured on the Tahoe-LAFS server responding to the request.

This operation will return an error if a blocking file is present at any of the parent names, preventing the server from creating the necessary parent directory; or if it would require changing an immutable directory.

The write-cap of the new directory will be returned as the HTTP response body.

`POST /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=mkdir-with-children`

Like /uri?t=mkdir-with-children, but the final directory is created as a child of an existing mutable directory. This will create additional intermediate mutable directories as necessary. If the final directory is created, it will be populated with initial children from the POST request body, as described above.

This accepts a format= argument in the query string, which controls the format of the target directory, if the target directory is created as part of the operation. format= is interpreted in the same way as in the POST/ uri?t=mkdir-with-children operation. Note that format= only controls the format of the named target directory; intermediate directories, if created, are created using the default mutable type setting, as configured on the Tahoe-LAFS server responding to the request.

This operation will return an error if a blocking file is present at any of the parent names, preventing the server from creating the necessary parent directory; or if it would require changing an immutable directory; or if the immediate parent directory already has a a child named SUBDIR.

`POST /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=mkdir-immutable`

Like /uri?t=mkdir-immutable, but the final directory is created as a child of an existing mutable directory. The final directory will be deep-immutable, and will be populated with the children specified as a JSON dictionary in the POST request body.

In Tahoe 1.6 this operation creates intermediate mutable directories if necessary, but that behaviour should not be relied on; see ticket #920.

This operation will return an error if the parent directory is immutable, or already has a child named SUBDIR.

`POST /uri/$DIRCAP/[SUBDIRS../]?t=mkdir&name=NAME`

Create a new empty mutable directory and attach it to the given existing directory. This will create additional intermediate directories as necessary.

This accepts a format= argument in the query string, which controls the format of the named target directory, if it does not already exist. format= is interpreted in the same way as in the POST /uri?t=mkdir form. Note that format= only controls the format of the named target directory; intermediate directories, if created, are created based on the default mutable type, as configured on the Tahoe-LAFS server responding to the request.

This operation will return an error if a blocking file is present at any of the parent names, preventing the server from creating the necessary parent directory, or if it would require changing any immutable directory.

The URL of this operation points to the parent of the bottommost new directory, whereas the /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=mkdir operation above has a URL that points directly to the bottommost new directory.

`POST /uri/$DIRCAP/[SUBDIRS../]?t=mkdir-with-children&name=NAME`

Like /uri/$DIRCAP/[SUBDIRS../]?t=mkdir&name=NAME, but the new directory will be populated with initial children via the POST request body. This command will create additional intermediate mutable directories as necessary.

This accepts a format= argument in the query string, which controls the format of the target directory, if the target directory is created as part of the operation. format= is interpreted in the same way as in the POST/ uri?t=mkdir-with-children operation. Note that format= only controls the format of the named target directory; intermediate directories, if created, are created using the default mutable type setting, as configured on the Tahoe-LAFS server responding to the request.

This operation will return an error if a blocking file is present at any of the parent names, preventing the server from creating the necessary parent directory; or if it would require changing an immutable directory; or if the immediate parent directory already has a a child named NAME.

Note that the name= argument must be passed as a queryarg, because the POST request body is used for the initial children JSON.

`POST /uri/$DIRCAP/[SUBDIRS../]?t=mkdir-immutable&name=NAME`

Like /uri/$DIRCAP/[SUBDIRS../]?t=mkdir-with-children&name=NAME, but the final directory will be deep-immutable. The children are specified as a JSON dictionary in the POST request body. Again, the name= argument must be passed as a queryarg.

In Tahoe 1.6 this operation creates intermediate mutable directories if necessary, but that behaviour should not be relied on; see ticket #920.

This operation will return an error if the parent directory is immutable, or already has a child named NAME.

Getting Information About a File Or Directory (as JSON)

`GET /uri/$FILECAP?t=json`

`GET /uri/$DIRCAP?t=json`

`GET /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=json`

`GET /uri/$DIRCAP/[SUBDIRS../]FILENAME?t=json`

This returns a machine-parseable JSON-encoded description of the given object. The JSON always contains a list, and the first element of the list is always a flag that indicates whether the referenced object is a file or a directory. If it is a capability to a file, then the information includes file size and URI, like this:

```
GET /uri/$FILECAP?t=json :

 [ "filenode", {
    "ro_uri": file_uri,
    "verify_uri": verify_uri,
    "size": bytes,
    "mutable": false,
    "format": "CHK"
   } ]
```

If it is a capability to a directory followed by a path from that directory to a file, then the information also includes metadata from the link to the file in the parent directory, like this:

```
GET /uri/$DIRCAP/[SUBDIRS../]FILENAME?t=json

 [ "filenode", {
    "ro_uri": file_uri,
    "verify_uri": verify_uri,
    "size": bytes,
    "mutable": false,
    "format": "CHK",
    "metadata": {
     "ctime": 1202777696.7564139,
     "mtime": 1202777696.7564139,
     "tahoe": {
      "linkcrtime": 1202777696.7564139,
      "linkmotime": 1202777696.7564139
     } } } ]
```

If it is a directory, then it includes information about the children of this directory, as a mapping from child name to a set of data about the child (the same data that would appear in a corresponding GET?t=json of the child itself). The child entries also include metadata about each child, including link-creation- and link-change- timestamps. The output looks like this:

```
GET /uri/$DIRCAP?t=json :
GET /uri/$DIRCAP/[SUBDIRS../]SUBDIR?t=json :

 [ "dirnode", {
   "rw_uri": read_write_uri,
   "ro_uri": read_only_uri,
   "verify_uri": verify_uri,
   "mutable": true,
   "format": "SDMF",
   "children": {
    "foo.txt": [ "filenode",
                 {
                   "ro_uri": uri,
                   "size": bytes,
                   "metadata": {
                     "ctime": 1202777696.7564139,
                     "mtime": 1202777696.7564139,
                     "tahoe": {
                       "linkcrtime": 1202777696.7564139,
                       "linkmotime": 1202777696.7564139
```

```
                    } } } ],
    "subdir":  [ "dirnode",
                 {
                   "rw_uri": rwuri,
                   "ro_uri": rouri,
                   "metadata": {
                     "ctime": 1202778102.7589991,
                     "mtime": 1202778111.2160511,
                     "tahoe": {
                       "linkcrtime": 1202777696.7564139,
                       "linkmotime": 1202777696.7564139
                     } } } ]
    } } ]
```

In the above example, note how 'children' is a dictionary in which the keys are child names and the values depend upon whether the child is a file or a directory. The value is mostly the same as the JSON representation of the child object (except that directories do not recurse – the "children" entry of the child is omitted, and the directory view includes the metadata that is stored on the directory edge).

The rw_uri field will be present in the information about a directory if and only if you have read-write access to that directory. The verify_uri field will be present if and only if the object has a verify-cap (non-distributed LIT files do not have verify-caps).

If the cap is of an unknown format, then the file size and verify_uri will not be available:

```
GET /uri/$UNKNOWNCAP?t=json :

 [ "unknown", {
     "ro_uri": unknown_read_uri
     } ]

GET /uri/$DIRCAP/[SUBDIRS../]UNKNOWNCHILDNAME?t=json :

 [ "unknown", {
     "rw_uri": unknown_write_uri,
     "ro_uri": unknown_read_uri,
     "mutable": true,
     "metadata": {
       "ctime": 1202777696.7564139,
       "mtime": 1202777696.7564139,
       "tahoe": {
         "linkcrtime": 1202777696.7564139,
         "linkmotime": 1202777696.7564139
       } } } ]
```

As in the case of file nodes, the metadata will only be present when the capability is to a directory followed by a path. The "mutable" field is also not always present; when it is absent, the mutability of the object is not known.

About the metadata

The value of the 'tahoe':'linkmotime' key is updated whenever a link to a child is set. The value of the 'tahoe':'linkcrtime' key is updated whenever a link to a child is created – i.e. when there was not previously a link under that name.

Note however, that if the edge in the Tahoe-LAFS file store points to a mutable file and the contents of that mutable file is changed, then the 'tahoe':'linkmotime' value on that edge will *not* be updated, since the edge itself wasn't updated

– only the mutable file was.

The timestamps are represented as a number of seconds since the UNIX epoch (1970-01-01 00:00:00 UTC), with leap seconds not being counted in the long term.

In Tahoe earlier than v1.4.0, 'mtime' and 'ctime' keys were populated instead of the 'tahoe':'linkmotime' and 'tahoe':'linkcrtime' keys. Starting in Tahoe v1.4.0, the 'linkmotime'/'linkcrtime' keys in the 'tahoe' sub-dict are populated. However, prior to Tahoe v1.7beta, a bug caused the 'tahoe' sub-dict to be deleted by web-API requests in which new metadata is specified, and not to be added to existing child links that lack it.

From Tahoe v1.7.0 onward, the 'mtime' and 'ctime' fields are no longer populated or updated (see ticket #924), except by "tahoe backup" as explained below. For backward compatibility, when an existing link is updated and 'tahoe':'linkcrtime' is not present in the previous metadata but 'ctime' is, the old value of 'ctime' is used as the new value of 'tahoe':'linkcrtime'.

The reason we added the new fields in Tahoe v1.4.0 is that there is a "set_children" API (described below) which you can use to overwrite the values of the 'mtime'/'ctime' pair, and this API is used by the "tahoe backup" command (in Tahoe v1.3.0 and later) to set the 'mtime' and 'ctime' values when backing up files from a local filesystem into the Tahoe-LAFS file store. As of Tahoe v1.4.0, the set_children API cannot be used to set anything under the 'tahoe' key of the metadata dict – if you include 'tahoe' keys in your 'metadata' arguments then it will silently ignore those keys.

Therefore, if the 'tahoe' sub-dict is present, you can rely on the 'linkcrtime' and 'linkmotime' values therein to have the semantics described above. (This is assuming that only official Tahoe clients have been used to write those links, and that their system clocks were set to what you expected – there is nothing preventing someone from editing their Tahoe client or writing their own Tahoe client which would overwrite those values however they like, and there is nothing to constrain their system clock from taking any value.)

When an edge is created or updated by "tahoe backup", the 'mtime' and 'ctime' keys on that edge are set as follows:

- 'mtime' is set to the timestamp read from the local filesystem for the "mtime" of the local file in question, which means the last time the contents of that file were changed.
- On Windows, 'ctime' is set to the creation timestamp for the file read from the local filesystem. On other platforms, 'ctime' is set to the UNIX "ctime" of the local file, which means the last time that either the contents or the metadata of the local file was changed.

There are several ways that the 'ctime' field could be confusing:

1. You might be confused about whether it reflects the time of the creation of a link in the Tahoe-LAFS file store (by a version of Tahoe < v1.7.0) or a timestamp copied in by "tahoe backup" from a local filesystem.
2. You might be confused about whether it is a copy of the file creation time (if "tahoe backup" was run on a Windows system) or of the last contents-or-metadata change (if "tahoe backup" was run on a different operating system).
3. You might be confused by the fact that changing the contents of a mutable file in Tahoe doesn't have any effect on any links pointing at that file in any directories, although "tahoe backup" sets the link 'ctime'/'mtime' to reflect timestamps about the local file corresponding to the Tahoe file to which the link points.
4. Also, quite apart from Tahoe, you might be confused about the meaning of the "ctime" in UNIX local filesystems, which people sometimes think means file creation time, but which actually means, in UNIX local filesystems, the most recent time that the file contents or the file metadata (such as owner, permission bits, extended attributes, etc.) has changed. Note that although "ctime" does not mean file creation time in UNIX, links created by a version of Tahoe prior to v1.7.0, and never written by "tahoe backup", will have 'ctime' set to the link creation time.

Attaching an Existing File or Directory by its read- or write-cap

```
PUT /uri/$DIRCAP/[SUBDIRS../]CHILDNAME?t=uri
```

This attaches a child object (either a file or directory) to a specified location in the Tahoe-LAFS file store. The child object is referenced by its read- or write- cap, as provided in the HTTP request body. This will create intermediate directories as necessary.

This is similar to a UNIX hardlink: by referencing a previously-uploaded file (or previously-created directory) instead of uploading/creating a new one, you can create two references to the same object.

The read- or write- cap of the child is provided in the body of the HTTP request, and this same cap is returned in the response body.

The default behavior is to overwrite any existing object at the same location. To prevent this (and make the operation return an error instead of overwriting), add a "replace=false" argument, as "?t=uri&replace=false". With replace=false, this operation will return an HTTP 409 "Conflict" error if there is already an object at the given location, rather than overwriting the existing object. To allow the operation to overwrite a file, but return an error when trying to overwrite a directory, use "replace=only-files" (this behavior is closer to the traditional UNIX "mv" command). Note that "true", "t", and "1" are all synonyms for "True", and "false", "f", and "0" are synonyms for "False", and the parameter is case-insensitive.

Note that this operation does not take its child cap in the form of separate "rw_uri" and "ro_uri" fields. Therefore, it cannot accept a child cap in a format unknown to the web-API server, unless its URI starts with "ro." or "imm.". This restriction is necessary because the server is not able to attenuate an unknown write cap to a read cap. Unknown URIs starting with "ro." or "imm.", on the other hand, are assumed to represent read caps. The client should not prefix a write cap with "ro." or "imm." and pass it to this operation, since that would result in granting the cap's write authority to holders of the directory read cap.

Adding Multiple Files or Directories to a Parent Directory at Once

`POST /uri/$DIRCAP/[SUBDIRS..]?t=set_children`

`POST /uri/$DIRCAP/[SUBDIRS..]?t=set-children` (Tahoe >= v1.6)

This command adds multiple children to a directory in a single operation. It reads the request body and interprets it as a JSON-encoded description of the child names and read/write-caps that should be added.

The body should be a JSON-encoded dictionary, in the same format as the "children" value returned by the "GET /uri/$DIRCAP?t=json" operation described above. In this format, each key is a child names, and the corresponding value is a tuple of (type, childinfo). "type" is ignored, and "childinfo" is a dictionary that contains "rw_uri", "ro_uri", and "metadata" keys. You can take the output of "GET /uri/$DIRCAP1?t=json" and use it as the input to "POST /uri/$DIRCAP2?t=set_children" to make DIR2 look very much like DIR1 (except for any existing children of DIR2 that were not overwritten, and any existing "tahoe" metadata keys as described below).

When the set_children request contains a child name that already exists in the target directory, this command defaults to overwriting that child with the new value (both child cap and metadata, but if the JSON data does not contain a "metadata" key, the old child's metadata is preserved). The command takes a boolean "overwrite=" query argument to control this behavior. If you use "?t=set_children&overwrite=false", then an attempt to replace an existing child will instead cause an error.

Any "tahoe" key in the new child's "metadata" value is ignored. Any existing "tahoe" metadata is preserved. The metadata["tahoe"] value is reserved for metadata generated by the tahoe node itself. The only two keys currently placed here are "linkcrtime" and "linkmotime". For details, see the section above entitled "Getting Information About a File Or Directory (as JSON)", in the "About the metadata" subsection.

Note that this command was introduced with the name "set_children", which uses an underscore rather than a hyphen as other multi-word command names do. The variant with a hyphen is now accepted, but clients that desire backward compatibility should continue to use "set_children".

Unlinking a File or Directory

`DELETE /uri/$DIRCAP/[SUBDIRS../]CHILDNAME`

This removes the given name from its parent directory. CHILDNAME is the name to be removed, and $DIRCAP/SUBDIRS.. indicates the directory that will be modified.

Note that this does not actually delete the file or directory that the name points to from the tahoe grid – it only unlinks the named reference from this directory. If there are other names in this directory or in other directories that point to the resource, then it will remain accessible through those paths. Even if all names pointing to this object are removed from their parent directories, then someone with possession of its read-cap can continue to access the object through that cap.

The object will only become completely unreachable once 1: there are no reachable directories that reference it, and 2: nobody is holding a read- or write- cap to the object. (This behavior is very similar to the way hardlinks and anonymous files work in traditional UNIX filesystems).

This operation will not modify more than a single directory. Intermediate directories which were implicitly created by PUT or POST methods will *not* be automatically removed by DELETE.

This method returns the file- or directory- cap of the object that was just removed.

Browser Operations: Human-oriented interfaces

This section describes the HTTP operations that provide support for humans running a web browser. Most of these operations use HTML forms that use POST to drive the Tahoe-LAFS node. This section is intended for HTML authors who want to write web pages containing user interfaces for manipulating the Tahoe-LAFS file store.

Note that for all POST operations, the arguments listed can be provided either as URL query arguments or as form body fields. URL query arguments are separated from the main URL by "?", and from each other by "&". For example, "POST /uri/$DIRCAP?t=upload&mutable=true". Form body fields are usually specified by using <input type="hidden"> elements. For clarity, the descriptions below display the most significant arguments as URL query args.

Viewing a Directory (as HTML)

`GET /uri/$DIRCAP/[SUBDIRS../]`

This returns an HTML page, intended to be displayed to a human by a web browser, which contains HREF links to all files and directories reachable from this directory. These HREF links do not have a t= argument, meaning that a human who follows them will get pages also meant for a human. It also contains forms to upload new files, and to unlink files and directories from their parent directory. Those forms use POST methods to do their job.

Viewing/Downloading a File

`GET /uri/$FILECAP`

`GET /uri/$DIRCAP/[SUBDIRS../]FILENAME`

`GET /named/$FILECAP/FILENAME`

These will retrieve the contents of the given file. The HTTP response body will contain the sequence of bytes that make up the file.

The `/named/` form is an alternative to `/uri/$FILECAP` which makes it easier to get the correct filename. The Tahoe server will provide the contents of the given file, with a Content-Type header derived from the given filename. This form is used to get browsers to use the "Save Link As" feature correctly, and also helps command-line tools like "wget" and "curl" use the right filename. Note that this form can *only* be used with file caps; it is an error to use a directory cap after the /named/ prefix.

URLs may also use /file/$FILECAP/FILENAME as a synonym for /named/$FILECAP/FILENAME. The use of "/file/" is deprecated in favor of "/named/" and support for "/file/" will be removed in a future release of Tahoe-LAFS.

If you use the first form (`/uri/$FILECAP`) and want the HTTP response to include a useful Content-Type header, add a "filename=foo" query argument, like "GET /uri/$FILECAP?filename=foo.jpg". The bare "GET /uri/$FILECAP" does not give the Tahoe node enough information to determine a Content-Type (since LAFS immutable files are merely sequences of bytes, not typed and named file objects).

If the URL has both filename= and "save=true" in the query arguments, then the server to add a "Content-Disposition: attachment" header, along with a filename= parameter. When a user clicks on such a link, most browsers will offer to let the user save the file instead of displaying it inline (indeed, most browsers will refuse to display it inline). "true", "t", "1", and other case-insensitive equivalents are all treated the same.

Character-set handling in URLs and HTTP headers is a *dubious art*. For maximum compatibility, Tahoe simply copies the bytes from the filename= argument into the Content-Disposition header's filename= parameter, without trying to interpret them in any particular way.

Getting Information About a File Or Directory (as HTML)

```
GET /uri/$FILECAP?t=info
```

```
GET /uri/$DIRCAP/?t=info
```

```
GET /uri/$DIRCAP/[SUBDIRS../]SUBDIR/?t=info
```

```
GET /uri/$DIRCAP/[SUBDIRS../]FILENAME?t=info
```

This returns a human-oriented HTML page with more detail about the selected file or directory object. This page contains the following items:

- object size
- storage index
- JSON representation
- raw contents (text/plain)
- access caps (URIs): verify-cap, read-cap, write-cap (for mutable objects)
- check/verify/repair form
- deep-check/deep-size/deep-stats/manifest (for directories)
- replace-contents form (for mutable files)

Creating a Directory

```
POST /uri?t=mkdir
```

This creates a new empty directory, but does not attach it to any other directory in the Tahoe-LAFS file store.

If a "redirect_to_result=true" argument is provided, then the HTTP response will cause the web browser to be redirected to a /uri/$DIRCAP page that gives access to the newly-created directory. If you bookmark this page, you'll be able to get back to the directory again in the future. This is the recommended way to start working with a Tahoe server: create a new unlinked directory (using redirect_to_result=true), then bookmark the resulting /uri/$DIRCAP page. There is a "create directory" button on the Welcome page to invoke this action.

This accepts a format= argument in the query string. Refer to the documentation of the PUT /uri?t=mkdir operation in *Creating A New Directory* for information on the behavior of the format= argument.

If "redirect_to_result=true" is not provided (or is given a value of "false"), then the HTTP response body will simply be the write-cap of the new directory.

`POST /uri/$DIRCAP/[SUBDIRS../]?t=mkdir&name=CHILDNAME`

This creates a new empty directory as a child of the designated SUBDIR. This will create additional intermediate directories as necessary.

This accepts a format= argument in the query string. Refer to the documentation of POST /uri/$DIRCAP/[SUBDIRS../]?t=mkdir&name=CHILDNAME in *Creating a New Directory* for information on the behavior of the format= argument.

If a "when_done=URL" argument is provided, the HTTP response will cause the web browser to redirect to the given URL. This provides a convenient way to return the browser to the directory that was just modified. Without a when_done= argument, the HTTP response will simply contain the write-cap of the directory that was just created.

Uploading a File

`POST /uri?t=upload`

This uploads a file, and produces a file-cap for the contents, but does not attach the file to any directory in the Tahoe-LAFS file store. That is, no directories will be modified by this operation.

The file must be provided as the "file" field of an HTML encoded form body, produced in response to an HTML form like this:

```
<form action="/uri" method="POST" enctype="multipart/form-data">
 <input type="hidden" name="t" value="upload" />
 <input type="file" name="file" />
 <input type="submit" value="Upload Unlinked" />
</form>
```

If a "when_done=URL" argument is provided, the response body will cause the browser to redirect to the given URL. If the when_done= URL has the string "%(uri)s" in it, that string will be replaced by a URL-escaped form of the newly created file-cap. (Note that without this substitution, there is no way to access the file that was just uploaded).

The default (in the absence of when_done=) is to return an HTML page that describes the results of the upload. This page will contain information about which storage servers were used for the upload, how long each operation took, etc.

This accepts format= and mutable=true query string arguments. Refer to *Writing/Uploading a File* for information on the behavior of format= and mutable=true.

`POST /uri/$DIRCAP/[SUBDIRS../]?t=upload`

This uploads a file, and attaches it as a new child of the given directory, which must be mutable. The file must be provided as the "file" field of an HTML-encoded form body, produced in response to an HTML form like this:

```
<form action="." method="POST" enctype="multipart/form-data">
 <input type="hidden" name="t" value="upload" />
 <input type="file" name="file" />
 <input type="submit" value="Upload" />
</form>
```

A "name=" argument can be provided to specify the new child's name, otherwise it will be taken from the "filename" field of the upload form (most web browsers will copy the last component of the original file's pathname into this field). To avoid confusion, name= is not allowed to contain a slash.

If there is already a child with that name, and it is a mutable file, then its contents are replaced with the data being uploaded. If it is not a mutable file, the default behavior is to remove the existing child before creating a new one. To prevent this (and make the operation return an error instead of overwriting the old child), add a "replace=false" argument, as "?t=upload&replace=false". With replace=false, this operation will return an HTTP 409 "Conflict" error if there is already an object at the given location, rather than overwriting the existing object. Note that "true", "t", and "1" are all synonyms for "True", and "false", "f", and "0" are synonyms for "False". the parameter is case-insensitive.

This will create additional intermediate directories as necessary, although since it is expected to be triggered by a form that was retrieved by "GET /uri/$DIRCAP/[SUBDIRS../]", it is likely that the parent directory will already exist.

This accepts format= and mutable=true query string arguments. Refer to *Writing/Uploading a File* for information on the behavior of format= and mutable=true.

If a "when_done=URL" argument is provided, the HTTP response will cause the web browser to redirect to the given URL. This provides a convenient way to return the browser to the directory that was just modified. Without a when_done= argument, the HTTP response will simply contain the file-cap of the file that was just uploaded (a write-cap for mutable files, or a read-cap for immutable files).

`POST /uri/$DIRCAP/[SUBDIRS../]FILENAME?t=upload`

This also uploads a file and attaches it as a new child of the given directory, which must be mutable. It is a slight variant of the previous operation, as the URL refers to the target file rather than the parent directory. It is otherwise identical: this accepts mutable= and when_done= arguments too.

`POST /uri/$FILECAP?t=upload`

This modifies the contents of an existing mutable file in-place. An error is signalled if $FILECAP does not refer to a mutable file. It behaves just like the "PUT /uri/$FILECAP" form, but uses a POST for the benefit of HTML forms in a web browser.

Attaching An Existing File Or Directory (by URI)

`POST /uri/$DIRCAP/[SUBDIRS../]?t=uri&name=CHILDNAME&uri=CHILDCAP`

This attaches a given read- or write- cap "CHILDCAP" to the designated directory, with a specified child name. This behaves much like the PUT t=uri operation, and is a lot like a UNIX hardlink. It is subject to the same restrictions as that operation on the use of cap formats unknown to the web-API server.

This will create additional intermediate directories as necessary, although since it is expected to be triggered by a form that was retrieved by "GET /uri/$DIRCAP/[SUBDIRS../]", it is likely that the parent directory will already exist.

This accepts the same replace= argument as POST t=upload.

Unlinking a Child

`POST /uri/$DIRCAP/[SUBDIRS../]?t=delete&name=CHILDNAME`

`POST /uri/$DIRCAP/[SUBDIRS../]?t=unlink&name=CHILDNAME` (Tahoe >= v1.9)

This instructs the node to remove a child object (file or subdirectory) from the given directory, which must be mutable. Note that the entire subtree is unlinked from the parent. Unlike deleting a subdirectory in a UNIX local filesystem, the subtree need not be empty; if it isn't, then other references into the subtree will see that the child subdirectories are not modified by this operation. Only the link from the given directory to its child is severed.

In Tahoe-LAFS v1.9.0 and later, t=unlink can be used as a synonym for t=delete. If interoperability with older web-API servers is required, t=delete should be used.

Renaming a Child

`POST /uri/$DIRCAP/[SUBDIRS../]?t=rename&from_name=OLD&to_name=NEW`

This instructs the node to rename a child of the given directory, which must be mutable. This has a similar effect to removing the child, then adding the same child-cap under the new name, except that it preserves metadata. This operation cannot move the child to a different directory.

The default behavior is to overwrite any existing link at the destination (replace=true). To prevent this (and make the operation return an error instead of overwriting), add a "replace=false" argument. With replace=false, this operation will return an HTTP 409 "Conflict" error if the destination is not the same link as the source and there is already a link at the destination, rather than overwriting the existing link. To allow the operation to overwrite a link to a file, but return an HTTP 409 error when trying to overwrite a link to a directory, use "replace=only-files" (this behavior is closer to the traditional UNIX "mv" command). Note that "true", "t", and "1" are all synonyms for "True"; "false", "f", and "0" are synonyms for "False"; and the parameter is case-insensitive.

Relinking ("Moving") a Child

`POST /uri/$DIRCAP/[SUBDIRS../]?t=relink&from_name=OLD&to_dir=$NEWDIRCAP/[NEWSUBDIRS../]&to_`
`[&replace=true|false|only-files]` (Tahoe >= v1.10)

This instructs the node to move a child of the given source directory, into a different directory and/or to a different name. The command is named `relink` because what it does is add a new link to the child from the new location, then remove the old link. Nothing is actually "moved": the child is still reachable through any path from which it was formerly reachable, and the storage space occupied by its ciphertext is not affected.

The source and destination directories must be writeable. If `to_dir` is not present, the child link is renamed within the same directory. If `to_name` is not present then it defaults to `from_name`. If the destination link (directory and name) is the same as the source link, the operation has no effect.

Metadata from the source directory entry is preserved. Multiple levels of descent in the source and destination paths are supported.

This operation will return an HTTP 404 "Not Found" error if `$DIRCAP/[SUBDIRS../]`, the child being moved, or the destination directory does not exist. It will return an HTTP 400 "Bad Request" error if any entry in the source or destination paths is not a directory.

The default behavior is to overwrite any existing link at the destination (replace=true). To prevent this (and make the operation return an error instead of overwriting), add a "replace=false" argument. With replace=false, this operation will return an HTTP 409 "Conflict" error if the destination is not the same link as the source and there is already a link at the destination, rather than overwriting the existing link. To allow the operation to overwrite

a link to a file, but return an HTTP 409 error when trying to overwrite a link to a directory, use "replace=only-files" (this behavior is closer to the traditional UNIX "mv" command). Note that "true", "t", and "1" are all synonyms for "True"; "false", "f", and "0" are synonyms for "False"; and the parameter is case-insensitive.

When relinking into a different directory, for safety, the child link is not removed from the old directory until it has been successfully added to the new directory. This implies that in case of a crash or failure, the link to the child will not be lost, but it could be linked at both the old and new locations.

The source link should not be the same as any link (directory and child name) in the `to_dir` path. This restriction is not enforced, but it may be enforced in a future version. If it were violated then the result would be to create a cycle in the directory structure that is not necessarily reachable from the root of the destination path (`$NEWDIRCAP`), which could result in data loss, as described in ticket #943.

Other Utilities

`GET /uri?uri=$CAP`

This causes a redirect to /uri/$CAP, and retains any additional query arguments (like filename= or save=). This is for the convenience of web forms which allow the user to paste in a read- or write- cap (obtained through some out-of-band channel, like IM or email).

Note that this form merely redirects to the specific file or directory indicated by the $CAP: unlike the GET /uri/$DIRCAP form, you cannot traverse to children by appending additional path segments to the URL.

`GET /uri/$DIRCAP/[SUBDIRS../]?t=rename-form&name=$CHILDNAME`

This provides a useful facility to browser-based user interfaces. It returns a page containing a form targetting the "POST $DIRCAP t=rename" functionality described above, with the provided $CHILDNAME present in the 'from_name' field of that form. I.e. this presents a form offering to rename $CHILDNAME, requesting the new name, and submitting POST rename. This same URL format can also be used with "move-form" with the expected results.

`GET /uri/$DIRCAP/[SUBDIRS../]CHILDNAME?t=uri`

This returns the file- or directory- cap for the specified object.

`GET /uri/$DIRCAP/[SUBDIRS../]CHILDNAME?t=readonly-uri`

This returns a read-only file- or directory- cap for the specified object. If the object is an immutable file, this will return the same value as t=uri.

Debugging and Testing Features

These URLs are less-likely to be helpful to the casual Tahoe user, and are mainly intended for developers.

`POST $URL?t=check`

This triggers the FileChecker to determine the current "health" of the given file or directory, by counting how many shares are available. The page that is returned will display the results. This can be used as a "show me detailed information about this file" page.

If a verify=true argument is provided, the node will perform a more intensive check, downloading and verifying every single bit of every share.

If an add-lease=true argument is provided, the node will also add (or renew) a lease to every share it encounters. Each lease will keep the share alive for a certain period of time (one month by default). Once the last lease expires or is explicitly cancelled, the storage server is allowed to delete the share.

If an output=JSON argument is provided, the response will be machine-readable JSON instead of human-oriented HTML. The data is a dictionary with the following keys:

```
storage-index: a base32-encoded string with the objects's storage index,
               or an empty string for LIT files
summary: a string, with a one-line summary of the stats of the file
results: a dictionary that describes the state of the file. For LIT files,
         this dictionary has only the 'healthy' key, which will always be
         True. For distributed files, this dictionary has the following
         keys:
  count-happiness: the servers-of-happiness level of the file, as
                   defined in doc/specifications/servers-of-happiness.
  count-shares-good: the number of good shares that were found
  count-shares-needed: 'k', the number of shares required for recovery
  count-shares-expected: 'N', the number of total shares generated
  count-good-share-hosts: the number of distinct storage servers with
                          good shares. Note that a high value does not
                          necessarily imply good share distribution,
                          because some of these servers may only hold
                          duplicate shares.
  count-wrong-shares: for mutable files, the number of shares for
                      versions other than the 'best' one (highest
                      sequence number, highest roothash). These are
                      either old, or created by an uncoordinated or
                      not fully successful write.
  count-recoverable-versions: for mutable files, the number of
                              recoverable versions of the file. For
                              a healthy file, this will equal 1.
  count-unrecoverable-versions: for mutable files, the number of
                                unrecoverable versions of the file.
                                For a healthy file, this will be 0.
  count-corrupt-shares: the number of shares with integrity failures
  list-corrupt-shares: a list of "share locators", one for each share
                       that was found to be corrupt. Each share locator
                       is a list of (serverid, storage_index, sharenum).
  servers-responding: list of base32-encoded storage server identifiers,
                      one for each server which responded to the share
                      query.
  healthy: (bool) True if the file is completely healthy, False otherwise.
           Healthy files have at least N good shares. Overlapping shares
           do not currently cause a file to be marked unhealthy. If there
           are at least N good shares, then corrupt shares do not cause the
           file to be marked unhealthy, although the corrupt shares will be
           listed in the results (list-corrupt-shares) and should be manually
           removed to wasting time in subsequent downloads (as the
           downloader rediscovers the corruption and uses alternate shares).
           Future compatibility: the meaning of this field may change to
           reflect whether the servers-of-happiness criterion is met
           (see ticket #614).
  sharemap: dict mapping share identifier to list of serverids
            (base32-encoded strings). This indicates which servers are
            holding which shares. For immutable files, the shareid is
            an integer (the share number, from 0 to N-1). For
            immutable files, it is a string of the form
            'seq%d-%s-sh%d', containing the sequence number, the
            roothash, and the share number.
```

Before Tahoe-LAFS v1.11, the `results` dictionary also had a `needs-rebalancing` field, but that has been removed since it was computed incorrectly.

`POST $URL?t=start-deep-check` (must add &ophandle=XYZ)

This initiates a recursive walk of all files and directories reachable from the target, performing a check on each one just like t=check. The result page will contain a summary of the results, including details on any file/directory that was not fully healthy.

t=start-deep-check can only be invoked on a directory. An error (400 BAD_REQUEST) will be signalled if it is invoked on a file. The recursive walker will deal with loops safely.

This accepts the same verify= and add-lease= arguments as t=check.

Since this operation can take a long time (perhaps a second per object), the ophandle= argument is required (see "Slow Operations, Progress, and Cancelling" above). The response to this POST will be a redirect to the corresponding /operations/$HANDLE page (with output=HTML or output=JSON to match the output= argument given to the POST). The deep-check operation will continue to run in the background, and the /operations page should be used to find out when the operation is done.

Detailed check results for non-healthy files and directories will be available under /operations/$HANDLE/$STORAGEINDEX, and the HTML status will contain links to these detailed results.

The HTML /operations/$HANDLE page for incomplete operations will contain a meta-refresh tag, set to 60 seconds, so that a browser which uses deep-check will automatically poll until the operation has completed.

The JSON page (/options/$HANDLE?output=JSON) will contain a machine-readable JSON dictionary with the following keys:

```
finished: a boolean, True if the operation is complete, else False. Some
          of the remaining keys may not be present until the operation
          is complete.
root-storage-index: a base32-encoded string with the storage index of the
                    starting point of the deep-check operation
count-objects-checked: count of how many objects were checked. Note that
                       non-distributed objects (i.e. small immutable LIT
                       files) are not checked, since for these objects,
                       the data is contained entirely in the URI.
count-objects-healthy: how many of those objects were completely healthy
count-objects-unhealthy: how many were damaged in some way
count-corrupt-shares: how many shares were found to have corruption,
                      summed over all objects examined
list-corrupt-shares: a list of "share identifiers", one for each share
                     that was found to be corrupt. Each share identifier
                     is a list of (serverid, storage_index, sharenum).
list-unhealthy-files: a list of (pathname, check-results) tuples, for
                      each file that was not fully healthy. 'pathname' is
                      a list of strings (which can be joined by "/"
                      characters to turn it into a single string),
                      relative to the directory on which deep-check was
                      invoked. The 'check-results' field is the same as
                      that returned by t=check&output=JSON, described
                      above.
stats: a dictionary with the same keys as the t=start-deep-stats command
       (described below)
```

`POST $URL?t=stream-deep-check`

This initiates a recursive walk of all files and directories reachable from the target, performing a check on each one just like t=check. For each unique object (duplicates are skipped), a single line of JSON is emitted to the HTTP response channel (or an error indication, see below). When the walk is complete, a final line of JSON is emitted which contains the accumulated file-size/count "deep-stats" data.

This command takes the same arguments as t=start-deep-check.

A CLI tool can split the response stream on newlines into "response units", and parse each response unit as JSON. Each such parsed unit will be a dictionary, and will contain at least the "type" key: a string, one of "file", "directory", or "stats".

For all units that have a type of "file" or "directory", the dictionary will contain the following keys:

```
"path": a list of strings, with the path that is traversed to reach the
        object
"cap": a write-cap URI for the file or directory, if available, else a
       read-cap URI
"verifycap": a verify-cap URI for the file or directory
"repaircap": an URI for the weakest cap that can still be used to repair
             the object
"storage-index": a base32 storage index for the object
"check-results": a copy of the dictionary which would be returned by
                 t=check&output=json, with three top-level keys:
                 "storage-index", "summary", and "results", and a variety
                 of counts and sharemaps in the "results" value.
```

Note that non-distributed files (i.e. LIT files) will have values of None for verifycap, repaircap, and storage-index, since these files can neither be verified nor repaired, and are not stored on the storage servers. Likewise the check-results dictionary will be limited: an empty string for storage-index, and a results dictionary with only the "healthy" key.

The last unit in the stream will have a type of "stats", and will contain the keys described in the "start-deep-stats" operation, below.

If any errors occur during the traversal (specifically if a directory is unrecoverable, such that further traversal is not possible), an error indication is written to the response body, instead of the usual line of JSON. This error indication line will begin with the string "ERROR:" (in all caps), and contain a summary of the error on the rest of the line. The remaining lines of the response body will be a python exception. The client application should look for the ERROR: and stop processing JSON as soon as it is seen. Note that neither a file being unrecoverable nor a directory merely being unhealthy will cause traversal to stop. The line just before the ERROR: will describe the directory that was untraversable, since the unit is emitted to the HTTP response body before the child is traversed.

```
POST $URL?t=check&repair=true
```

This performs a health check of the given file or directory, and if the checker determines that the object is not healthy (some shares are missing or corrupted), it will perform a "repair". During repair, any missing shares will be regenerated and uploaded to new servers.

This accepts the same verify=true and add-lease= arguments as t=check. When an output=JSON argument is provided, the machine-readable JSON response will contain the following keys:

```
storage-index: a base32-encoded string with the objects's storage index,
               or an empty string for LIT files
repair-attempted: (bool) True if repair was attempted
repair-successful: (bool) True if repair was attempted and the file was
                   fully healthy afterwards. False if no repair was
                   attempted, or if a repair attempt failed.
pre-repair-results: a dictionary that describes the state of the file
                    before any repair was performed. This contains exactly
                    the same keys as the 'results' value of the t=check
                    response, described above.
post-repair-results: a dictionary that describes the state of the file
                     after any repair was performed. If no repair was
                     performed, post-repair-results and pre-repair-results
                     will be the same. This contains exactly the same keys
```

```
                    as the 'results' value of the t=check response,
                    described above.
```

`POST $URL?t=start-deep-check&repair=true` (must add &ophandle=XYZ)

This triggers a recursive walk of all files and directories, performing a t=check&repair=true on each one.

Like t=start-deep-check without the repair= argument, this can only be invoked on a directory. An error (400 BAD_REQUEST) will be signalled if it is invoked on a file. The recursive walker will deal with loops safely.

This accepts the same verify= and add-lease= arguments as t=start-deep-check. It uses the same ophandle= mechanism as start-deep-check. When an output=JSON argument is provided, the response will contain the following keys:

```
finished: (bool) True if the operation has completed, else False
root-storage-index: a base32-encoded string with the storage index of the
                    starting point of the deep-check operation
count-objects-checked: count of how many objects were checked

count-objects-healthy-pre-repair: how many of those objects were completely
                                  healthy, before any repair
count-objects-unhealthy-pre-repair: how many were damaged in some way
count-objects-healthy-post-repair: how many of those objects were completely
                                   healthy, after any repair
count-objects-unhealthy-post-repair: how many were damaged in some way

count-repairs-attempted: repairs were attempted on this many objects.
count-repairs-successful: how many repairs resulted in healthy objects
count-repairs-unsuccessful: how many repairs resulted did not results in
                            completely healthy objects
count-corrupt-shares-pre-repair: how many shares were found to have
                                 corruption, summed over all objects
                                 examined, before any repair
count-corrupt-shares-post-repair: how many shares were found to have
                                  corruption, summed over all objects
                                  examined, after any repair
list-corrupt-shares: a list of "share identifiers", one for each share
                     that was found to be corrupt (before any repair).
                     Each share identifier is a list of (serverid,
                     storage_index, sharenum).
list-remaining-corrupt-shares: like list-corrupt-shares, but mutable shares
                               that were successfully repaired are not
                               included. These are shares that need
                               manual processing. Since immutable shares
                               cannot be modified by clients, all corruption
                               in immutable shares will be listed here.
list-unhealthy-files: a list of (pathname, check-results) tuples, for
                      each file that was not fully healthy. 'pathname' is
                      relative to the directory on which deep-check was
                      invoked. The 'check-results' field is the same as
                      that returned by t=check&repair=true&output=JSON,
                      described above.
stats: a dictionary with the same keys as the t=start-deep-stats command
       (described below)
```

`POST $URL?t=stream-deep-check&repair=true`

This triggers a recursive walk of all files and directories, performing a t=check&repair=true on each one.

For each unique object (duplicates are skipped), a single line of JSON is emitted to the HTTP response channel (or an error indication). When the walk is complete, a final line of JSON is emitted which contains the accumulated file-size/count "deep-stats" data.

This emits the same data as t=stream-deep-check (without the repair=true), except that the "check-results" field is replaced with a "check-and-repair-results" field, which contains the keys returned by t=check&repair=true&output=json (i.e. repair-attempted, repair-successful, pre-repair-results, and post-repair-results). The output does not contain the summary dictionary that is provied by t=start-deep-check&repair=true (the one with count-objects-checked and list-unhealthy-files), since the receiving client is expected to calculate those values itself from the stream of per-object check-and-repair-results.

Note that the "ERROR:" indication will only be emitted if traversal stops, which will only occur if an unrecoverable directory is encountered. If a file or directory repair fails, the traversal will continue, and the repair failure will be indicated in the JSON data (in the "repair-successful" key).

`POST $DIRURL?t=start-manifest` (must add &ophandle=XYZ)

This operation generates a "manfest" of the given directory tree, mostly for debugging. This is a table of (path, filecap/dircap), for every object reachable from the starting directory. The path will be slash-joined, and the filecap/dircap will contain a link to the object in question. This page gives immediate access to every object in the file store subtree.

This operation uses the same ophandle= mechanism as deep-check. The corresponding /operations/$HANDLE page has three different forms. The default is output=HTML.

If output=text is added to the query args, the results will be a text/plain list. The first line is special: it is either "finished: yes" or "finished: no"; if the operation is not finished, you must periodically reload the page until it completes. The rest of the results are a plaintext list, with one file/dir per line, slash-separated, with the filecap/dircap separated by a space.

If output=JSON is added to the queryargs, then the results will be a JSON-formatted dictionary with six keys. Note that because large directory structures can result in very large JSON results, the full results will not be available until the operation is complete (i.e. until output["finished"] is True):

```
finished (bool): if False then you must reload the page until True
origin_si (base32 str): the storage index of the starting point
manifest: list of (path, cap) tuples, where path is a list of strings.
verifycaps: list of (printable) verify cap strings
storage-index: list of (base32) storage index strings
stats: a dictionary with the same keys as the t=start-deep-stats command
       (described below)
```

`POST $DIRURL?t=start-deep-size` (must add &ophandle=XYZ)

This operation generates a number (in bytes) containing the sum of the filesize of all directories and immutable files reachable from the given directory. This is a rough lower bound of the total space consumed by this subtree. It does not include space consumed by mutable files, nor does it take expansion or encoding overhead into account. Later versions of the code may improve this estimate upwards.

The /operations/$HANDLE status output consists of two lines of text:

```
finished: yes
size: 1234
```

`POST $DIRURL?t=start-deep-stats` (must add &ophandle=XYZ)

This operation performs a recursive walk of all files and directories reachable from the given directory, and generates a collection of statistics about those objects.

The result (obtained from the /operations/$OPHANDLE page) is a JSON-serialized dictionary with the following keys (note that some of these keys may be missing until 'finished' is True):

```
finished: (bool) True if the operation has finished, else False
api-version: (int), number of deep-stats API version. Will be increased every
             time backwards-incompatible change is introduced.
             Current version is 1.
count-immutable-files: count of how many CHK files are in the set
count-mutable-files: same, for mutable files (does not include directories)
count-literal-files: same, for LIT files (data contained inside the URI)
count-files: sum of the above three
count-directories: count of directories
count-unknown: count of unrecognized objects (perhaps from the future)
size-immutable-files: total bytes for all CHK files in the set, =deep-size
size-mutable-files (TODO): same, for current version of all mutable files
size-literal-files: same, for LIT files
size-directories: size of directories (includes size-literal-files)
size-files-histogram: list of (minsize, maxsize, count) buckets,
                      with a histogram of filesizes, 5dB/bucket,
                      for both literal and immutable files
largest-directory: number of children in the largest directory
largest-immutable-file: number of bytes in the largest CHK file
```

size-mutable-files is not implemented, because it would require extra queries to each mutable file to get their size. This may be implemented in the future.

Assuming no sharing, the basic space consumed by a single root directory is the sum of size-immutable-files, size-mutable-files, and size-directories. The actual disk space used by the shares is larger, because of the following sources of overhead:

```
integrity data
expansion due to erasure coding
share management data (leases)
backend (ext3) minimum block size
```

`POST $URL?t=stream-manifest`

This operation performs a recursive walk of all files and directories reachable from the given starting point. For each such unique object (duplicates are skipped), a single line of JSON is emitted to the HTTP response channel (or an error indication, see below). When the walk is complete, a final line of JSON is emitted which contains the accumulated file-size/count "deep-stats" data.

A CLI tool can split the response stream on newlines into "response units", and parse each response unit as JSON. Each such parsed unit will be a dictionary, and will contain at least the "type" key: a string, one of "file", "directory", or "stats".

For all units that have a type of "file" or "directory", the dictionary will contain the following keys:

```
"path": a list of strings, with the path that is traversed to reach the
        object
"cap": a write-cap URI for the file or directory, if available, else a
       read-cap URI
"verifycap": a verify-cap URI for the file or directory
"repaircap": an URI for the weakest cap that can still be used to repair
             the object
"storage-index": a base32 storage index for the object
```

Note that non-distributed files (i.e. LIT files) will have values of None for verifycap, repaircap, and storage-index, since these files can neither be verified nor repaired, and are not stored on the storage servers.

The last unit in the stream will have a type of "stats", and will contain the keys described in the "start-deep-stats" operation, below.

If any errors occur during the traversal (specifically if a directory is unrecoverable, such that further traversal is not possible), an error indication is written to the response body, instead of the usual line of JSON. This error indication line will begin with the string "ERROR:" (in all caps), and contain a summary of the error on the rest of the line. The remaining lines of the response body will be a python exception. The client application should look for the ERROR: and stop processing JSON as soon as it is seen. The line just before the ERROR: will describe the directory that was untraversable, since the manifest entry is emitted to the HTTP response body before the child is traversed.

Other Useful Pages

The portion of the web namespace that begins with "/uri" (and "/named") is dedicated to giving users (both humans and programs) access to the Tahoe-LAFS file store. The rest of the namespace provides status information about the state of the Tahoe-LAFS node.

`GET /` (the root page)

This is the "Welcome Page", and contains a few distinct sections:

```
Node information: library versions, local nodeid, services being provided.

File store access forms: create a new directory, view a file/directory by
                         URI, upload a file (unlinked), download a file by
                         URI.

Grid status: introducer information, helper information, connected storage
             servers.
```

`GET /?t=json` (the json welcome page)

This is the "json Welcome Page", and contains connectivity status of the introducer(s) and storage server(s), here's an example:

```
{
 "introducers": {
  "statuses": []
 },
 "servers": [{
   "nodeid": "other_nodeid",
   "available_space": 123456,
   "nickname": "George \u263b",
   "version": "1.0",
   "connection_status": "summary",
   "last_received_data": 1487811257
  }]
}
```

The above json `introducers` section includes a list of introducer connectivity status messages.

The above json `servers` section is an array with map elements. Each map has the following properties:

1. `nodeid` - an identifier derived from the node's public key
2. `available_space` - the available space in bytes expressed as an integer
3. `nickname` - the storage server nickname

4. `version` - the storage server Tahoe-LAFS version
5. `connection_status` - connectivity status
6. `last_received_data` - the time when data was last received, expressed in seconds since epoch

`GET /status/`

This page lists all active uploads and downloads, and contains a short list of recent upload/download operations. Each operation has a link to a page that describes file sizes, servers that were involved, and the time consumed in each phase of the operation.

A GET of /status/?t=json will contain a machine-readable subset of the same data. It returns a JSON-encoded dictionary. The only key defined at this time is "active", with a value that is a list of operation dictionaries, one for each active operation. Once an operation is completed, it will no longer appear in data["active"] .

Each op-dict contains a "type" key, one of "upload", "download", "mapupdate", "publish", or "retrieve" (the first two are for immutable files, while the latter three are for mutable files and directories).

The "upload" op-dict will contain the following keys:

```
type (string): "upload"
storage-index-string (string): a base32-encoded storage index
total-size (int): total size of the file
status (string): current status of the operation
progress-hash (float): 1.0 when the file has been hashed
progress-ciphertext (float): 1.0 when the file has been encrypted.
progress-encode-push (float): 1.0 when the file has been encoded and
                              pushed to the storage servers. For helper
                              uploads, the ciphertext value climbs to 1.0
                              first, then encoding starts. For unassisted
                              uploads, ciphertext and encode-push progress
                              will climb at the same pace.
```

The "download" op-dict will contain the following keys:

```
type (string): "download"
storage-index-string (string): a base32-encoded storage index
total-size (int): total size of the file
status (string): current status of the operation
progress (float): 1.0 when the file has been fully downloaded
```

Front-ends which want to report progress information are advised to simply average together all the progress-* indicators. A slightly more accurate value can be found by ignoring the progress-hash value (since the current implementation hashes synchronously, so clients will probably never see progress-hash!=1.0).

`GET /helper_status/`

If the node is running a helper (i.e. if [helper]enabled is set to True in tahoe.cfg), then this page will provide a list of all the helper operations currently in progress. If "?t=json" is added to the URL, it will return a JSON-formatted list of helper statistics, which can then be used to produce graphs to indicate how busy the helper is.

`GET /statistics/`

This page provides "node statistics", which are collected from a variety of sources:

```
load_monitor: every second, the node schedules a timer for one second in
              the future, then measures how late the subsequent callback
```

```
              is. The "load_average" is this tardiness, measured in
              seconds, averaged over the last minute. It is an indication
              of a busy node, one which is doing more work than can be
              completed in a timely fashion. The "max_load" value is the
              highest value that has been seen in the last 60 seconds.

cpu_monitor: every minute, the node uses time.clock() to measure how much
             CPU time it has used, and it uses this value to produce
             1min/5min/15min moving averages. These values range from 0%
             (0.0) to 100% (1.0), and indicate what fraction of the CPU
             has been used by the Tahoe node. Not all operating systems
             provide meaningful data to time.clock(): they may report 100%
             CPU usage at all times.

uploader: this counts how many immutable files (and bytes) have been
          uploaded since the node was started

downloader: this counts how many immutable files have been downloaded
            since the node was started

publishes: this counts how many mutable files (including directories) have
           been modified since the node was started

retrieves: this counts how many mutable files (including directories) have
           been read since the node was started
```

There are other statistics that are tracked by the node. The "raw stats" section shows a formatted dump of all of them.

By adding "?t=json" to the URL, the node will return a JSON-formatted dictionary of stats values, which can be used by other tools to produce graphs of node behavior. The misc/munin/ directory in the source distribution provides some tools to produce these graphs.

`GET /` (introducer status)

For Introducer nodes, the welcome page displays information about both clients and servers which are connected to the introducer. Servers make "service announcements", and these are listed in a table. Clients will subscribe to hear about service announcements, and these subscriptions are listed in a separate table. Both tables contain information about what version of Tahoe is being run by the remote node, their advertised and outbound IP addresses, their nodeid and nickname, and how long they have been available.

By adding "?t=json" to the URL, the node will return a JSON-formatted dictionary of stats values, which can be used to produce graphs of connected clients over time. This dictionary has the following keys:

```
["subscription_summary"] : a dictionary mapping service name (like
                           "storage") to an integer with the number of
                           clients that have subscribed to hear about that
                           service
["announcement_summary"] : a dictionary mapping service name to an integer
                           with the number of servers which are announcing
                           that service
["announcement_distinct_hosts"] : a dictionary mapping service name to an
                                  integer which represents the number of
                                  distinct hosts that are providing that
                                  service. If two servers have announced
                                  FURLs which use the same hostnames (but
                                  different ports and tubids), they are
                                  considered to be on the same host.
```

Static Files in /public_html

The web-API server will take any request for a URL that starts with /static and serve it from a configurable directory which defaults to $BASEDIR/public_html . This is configured by setting the "[node]web.static" value in $BASEDIR/tahoe.cfg . If this is left at the default value of "public_html", then http://127.0.0.1:3456/static/subdir/foo.html will be served with the contents of the file $BASEDIR/public_html/subdir/foo.html .

This can be useful to serve a javascript application which provides a prettier front-end to the rest of the Tahoe web-API.

Safety and Security Issues – Names vs. URIs

Summary: use explicit file- and dir- caps whenever possible, to reduce the potential for surprises when the file store structure is changed.

Tahoe-LAFS provides a mutable file store, but the ways that the store can change are limited. The only things that can change are:

- the mapping from child names to child objects inside mutable directories (by adding a new child, removing an existing child, or changing an existing child to point to a different object)
- the contents of mutable files

Obviously if you query for information about the file store and then act to change it (such as by getting a listing of the contents of a mutable directory and then adding a file to the directory), then the store might have been changed after you queried it and before you acted upon it. However, if you use the URI instead of the pathname of an object when you act upon the object, then it will be the same object; only its contents can change (if it is mutable). If, on the other hand, you act upon the object using its pathname, then a different object might be in that place, which can result in more kinds of surprises.

For example, suppose you are writing code which recursively downloads the contents of a directory. The first thing your code does is fetch the listing of the contents of the directory. For each child that it fetched, if that child is a file then it downloads the file, and if that child is a directory then it recurses into that directory. Now, if the download and the recurse actions are performed using the child's name, then the results might be wrong, because for example a child name that pointed to a subdirectory when you listed the directory might have been changed to point to a file (in which case your attempt to recurse into it would result in an error), or a child name that pointed to a file when you listed the directory might now point to a subdirectory (in which case your attempt to download the child would result in a file containing HTML text describing the subdirectory!).

If your recursive algorithm uses the URI of the child instead of the name of the child, then those kinds of mistakes just can't happen. Note that both the child's name and the child's URI are included in the results of listing the parent directory, so it isn't any harder to use the URI for this purpose.

The read and write caps in a given directory node are separate URIs, and can't be assumed to point to the same object even if they were retrieved in the same operation (although the web-API server attempts to ensure this in most cases). If you need to rely on that property, you should explicitly verify it. More generally, you should not make assumptions about the internal consistency of the contents of mutable directories. As a result of the signatures on mutable object versions, it is guaranteed that a given version was written in a single update, but – as in the case of a file – the contents may have been chosen by a malicious writer in a way that is designed to confuse applications that rely on their consistency.

In general, use names if you want "whatever object (whether file or directory) is found by following this name (or sequence of names) when my request reaches the server". Use URIs if you want "this particular object".

Concurrency Issues

Tahoe uses both mutable and immutable files. Mutable files can be created explicitly by doing an upload with ?mutable=true added, or implicitly by creating a new directory (since a directory is just a special way to interpret a given mutable file).

Mutable files suffer from the same consistency-vs-availability tradeoff that all distributed data storage systems face. It is not possible to simultaneously achieve perfect consistency and perfect availability in the face of network partitions (servers being unreachable or faulty).

Tahoe tries to achieve a reasonable compromise, but there is a basic rule in place, known as the Prime Coordination Directive: "Don't Do That". What this means is that if write-access to a mutable file is available to several parties, then those parties are responsible for coordinating their activities to avoid multiple simultaneous updates. This could be achieved by having these parties talk to each other and using some sort of locking mechanism, or by serializing all changes through a single writer.

The consequences of performing uncoordinated writes can vary. Some of the writers may lose their changes, as somebody else wins the race condition. In many cases the file will be left in an "unhealthy" state, meaning that there are not as many redundant shares as we would like (reducing the reliability of the file against server failures). In the worst case, the file can be left in such an unhealthy state that no version is recoverable, even the old ones. It is this small possibility of data loss that prompts us to issue the Prime Coordination Directive.

Tahoe nodes implement internal serialization to make sure that a single Tahoe node cannot conflict with itself. For example, it is safe to issue two directory modification requests to a single tahoe node's web-API server at the same time, because the Tahoe node will internally delay one of them until after the other has finished being applied. (This feature was introduced in Tahoe-1.1; back with Tahoe-1.0 the web client was responsible for serializing web requests themselves).

For more details, please see the "Consistency vs Availability" and "The Prime Coordination Directive" sections of *Mutable Files*.

Access Blacklist

Gateway nodes may find it necessary to prohibit access to certain files. The web-API has a facility to block access to filecaps by their storage index, returning a 403 "Forbidden" error instead of the original file.

This blacklist is recorded in $NODEDIR/access.blacklist, and contains one blocked file per line. Comment lines (starting with #) are ignored. Each line consists of the storage-index (in the usual base32 format as displayed by the "More Info" page, or by the "tahoe debug dump-cap" command), followed by whitespace, followed by a reason string, which will be included in the 403 error message. This could hold a URL to a page that explains why the file is blocked, for example.

So for example, if you found a need to block access to a file with filecap `URI:CHK:n7r3m6wmomelk4sep3kw5cvduq:os7ijw5c3maek7pg65e5254k2fzjflavtpejjyhshpsxuqzhcwwq:3:2` you could do the following:

```
tahoe debug dump-cap␣
↪URI:CHK:n7r3m6wmomelk4sep3kw5cvduq:os7ijw5c3maek7pg65e5254k2fzjflavtpejjyhshpsxuqzhcwwq:3:20:14861
-> storage index: whpepioyrnff7orecjolvbudeu
echo "whpepioyrnff7orecjolvbudeu my puppy told me to" >>$NODEDIR/access.blacklist
tahoe restart $NODEDIR
tahoe get␣
↪URI:CHK:n7r3m6wmomelk4sep3kw5cvduq:os7ijw5c3maek7pg65e5254k2fzjflavtpejjyhshpsxuqzhcwwq:3:20:14861
-> error, 403 Access Prohibited: my puppy told me to
```

The `access.blacklist` file will be checked each time a file or directory is accessed: the file's `mtime` is used to decide whether it need to be reloaded. Therefore no node restart is necessary when creating the initial blacklist, nor when adding second, third, or additional entries to the list. When modifying the file, be careful to update it atomically, otherwise a request may arrive while the file is only halfway written, and the partial file may be incorrectly parsed.

The blacklist is applied to all access paths (including SFTP, FTP, and CLI operations), not just the web-API. The blacklist also applies to directories. If a directory is blacklisted, the gateway will refuse access to both that directory and any child files/directories underneath it, when accessed via "DIRCAP/SUBDIR/FILENAME" -style URLs. Users who go directly to the child file/dir will bypass the blacklist.

The node will log the SI of the file being blocked, and the reason code, into the `logs/twistd.log` file.

URLs and HTTP and UTF-8

HTTP does not provide a mechanism to specify the character set used to encode non-ASCII names in URLs (RFC3986#2.1). We prefer the convention that the `filename=` argument shall be a URL-escaped UTF-8 encoded Unicode string. For example, suppose we want to provoke the server into using a filename of "f i a n c e-acute e" (i.e. f i a n c U+00E9 e). The UTF-8 encoding of this is 0x66 0x69 0x61 0x6e 0x63 0xc3 0xa9 0x65 (or "fianc\xC3\xA9e", as python's `repr()` function would show). To encode this into a URL, the non-printable characters must be escaped with the urlencode `%XX` mechanism, giving us "fianc%C3%A9e". Thus, the first line of the HTTP request will be "`GET /uri/CAP...?save=true&filename=fianc%C3%A9e HTTP/1.1`". Not all browsers provide this: IE7 by default uses the Latin-1 encoding, which is "fianc%E9e" (although it has a configuration option to send URLs as UTF-8).

The response header will need to indicate a non-ASCII filename. The actual mechanism to do this is not clear. For ASCII filenames, the response header would look like:

```
Content-Disposition: attachment; filename="english.txt"
```

If Tahoe were to enforce the UTF-8 convention, it would need to decode the URL argument into a Unicode string, and then encode it back into a sequence of bytes when creating the response header. One possibility would be to use unencoded UTF-8. Developers suggest that IE7 might accept this:

```
#1: Content-Disposition: attachment; filename="fianc\xC3\xA9e"
  (note, the last four bytes of that line, not including the newline, are
  0xC3 0xA9 0x65 0x22)
```

RFC2231#4 (dated 1997): suggests that the following might work, and some developers have reported that it is supported by Firefox (but not IE7):

```
#2: Content-Disposition: attachment; filename*=utf-8''fianc%C3%A9e
```

My reading of RFC2616#19.5.1 (which defines Content-Disposition) says that the filename= parameter is defined to be wrapped in quotes (presumably to allow spaces without breaking the parsing of subsequent parameters), which would give us:

```
#3: Content-Disposition: attachment; filename*=utf-8''"fianc%C3%A9e"
```

However this is contrary to the examples in the email thread listed above.

Developers report that IE7 (when it is configured for UTF-8 URL encoding, which is not the default in Asian countries), will accept:

```
#4: Content-Disposition: attachment; filename=fianc%C3%A9e
```

However, for maximum compatibility, Tahoe simply copies bytes from the URL into the response header, rather than enforcing the UTF-8 convention. This means it does not try to decode the filename from the URL argument, nor does it encode the filename into the response header.

CHAPTER 9

Tahoe-LAFS SFTP and FTP Frontends

SFTP/FTP Background

FTP is the venerable internet file-transfer protocol, first developed in 1971. The FTP server usually listens on port 21. A separate connection is used for the actual data transfers, either in the same direction as the initial client-to-server connection (for PORT mode), or in the reverse direction (for PASV) mode. Connections are unencrypted, so passwords, file names, and file contents are visible to eavesdroppers.

SFTP is the modern replacement, developed as part of the SSH "secure shell" protocol, and runs as a subchannel of the regular SSH connection. The SSH server usually listens on port 22. All connections are encrypted.

Both FTP and SFTP were developed assuming a UNIX-like server, with accounts and passwords, octal file modes (user/group/other, read/write/execute), and ctime/mtime timestamps.

We recommend SFTP over FTP, because the protocol is better, and the server implementation in Tahoe-LAFS is more complete. See *Known Issues*, below, for details.

Tahoe-LAFS Support

All Tahoe-LAFS client nodes can run a frontend SFTP server, allowing regular SFTP clients (like `/usr/bin/sftp`, the `sshfs` FUSE plugin, and many others) to access the file store. They can also run an FTP server, so FTP clients (like `/usr/bin/ftp`, `ncftp`, and others) can too. These frontends sit at the same level as the web-API interface.

Since Tahoe-LAFS does not use user accounts or passwords, the SFTP/FTP servers must be configured with a way to first authenticate a user (confirm that a prospective client has a legitimate claim to whatever authorities we might grant a particular user), and second to decide what directory cap should be used as the root directory for a log-in by the authenticated user. A username and password can be used; as of Tahoe-LAFS v1.11, RSA or DSA public key authentication is also supported.

Tahoe-LAFS provides two mechanisms to perform this user-to-cap mapping. The first (recommended) is a simple flat file with one account per line. The second is an HTTP-based login mechanism.

Creating an Account File

To use the first form, create a file (for example `BASEDIR/private/accounts`) in which each non-comment/non-blank line is a space-separated line of (USERNAME, PASSWORD, ROOTCAP), like so:

```
% cat BASEDIR/private/accounts
# This is a password line: username password cap
alice password␣
 ↪URI:DIR2:ioej8xmzrwilg772gzj4fhdg7a:wtiizszzz2rgmczv4wl6bqvbv33ag4kvbr6prz3u6w3geixa6m6a
bob sekrit␣
 ↪URI:DIR2:6bdmeitystckbl9yqlw7g56f4e:serp5ioqxnh34mlbmzwvkp3odehsyrr7eytt5f64we3k9hhcrcja

# This is a public key line: username keytype pubkey cap
# (Tahoe-LAFS v1.11 or later)
carol ssh-rsa AAAA...␣
 ↪URI:DIR2:ovjy4yhylqlfoqg2vcze36dhde:4d4f47qko2xm5g7osgo2yyidi5m4muyo2vjjy53q4vjju2u55mfa
```

For public key authentication, the keytype may be either "ssh-rsa" or "ssh-dsa". To avoid ambiguity between passwords and public key types, a password cannot start with "ssh-".

Now add an `accounts.file` directive to your `tahoe.cfg` file, as described in the next sections.

Running An Account Server (accounts.url)

The accounts.url directive allows access requests to be controlled by an HTTP-based login service, useful for centralized deployments. This was used by AllMyData to provide web-based file access, where the service used a simple PHP script and database lookups to map an account email address and password to a Tahoe-LAFS directory cap. The service will receive a multipart/form-data POST, just like one created with a <form> and <input> fields, with three parameters:

- action: "authenticate" (this is a static string)
- email: USERNAME (Tahoe-LAFS has no notion of email addresses, but the authentication service uses them as account names, so the interface presents this argument as "email" rather than "username").
- passwd: PASSWORD

It should return a single string that either contains a Tahoe-LAFS directory cap (URI:DIR2:...), or "0" to indicate a login failure.

Tahoe-LAFS recommends the service be secure, preferably localhost-only. This makes it harder for attackers to brute force the password or use DNS poisoning to cause the Tahoe-LAFS gateway to talk with the wrong server, thereby revealing the usernames and passwords.

Public key authentication is not supported when an account server is used.

Configuring SFTP Access

The Tahoe-LAFS SFTP server requires a host keypair, just like the regular SSH server. It is important to give each server a distinct keypair, to prevent one server from masquerading as different one. The first time a client program talks to a given server, it will store the host key it receives, and will complain if a subsequent connection uses a different key. This reduces the opportunity for man-in-the-middle attacks to just the first connection.

Exercise caution when connecting to the SFTP server remotely. The AES implementation used by the SFTP code does not have defenses against timing attacks. The code for encrypting the SFTP connection was not written by the Tahoe-LAFS team, and we have not reviewed it as carefully as we have reviewed the code for encrypting files and directories in Tahoe-LAFS itself. (See Twisted ticket #4633 for a possible fix to this issue.)

If you can connect to the SFTP server (which is provided by the Tahoe-LAFS gateway) only from a client on the same host, then you would be safe from any problem with the SFTP connection security. The examples given below enforce this policy by including ":interface=127.0.0.1" in the "port" option, which causes the server to only accept connections from localhost.

You will use directives in the tahoe.cfg file to tell the SFTP code where to find these keys. To create one, use the `ssh-keygen` tool (which comes with the standard OpenSSH client distribution):

```
% cd BASEDIR
% ssh-keygen -f private/ssh_host_rsa_key
```

The server private key file must not have a passphrase.

Then, to enable the SFTP server with an accounts file, add the following lines to the BASEDIR/tahoe.cfg file:

```
[sftpd]
enabled = true
port = tcp:8022:interface=127.0.0.1
host_pubkey_file = private/ssh_host_rsa_key.pub
host_privkey_file = private/ssh_host_rsa_key
accounts.file = private/accounts
```

The SFTP server will listen on the given port number and on the loopback interface only. The "accounts.file" pathname will be interpreted relative to the node's BASEDIR.

Or, to use an account server instead, do this:

```
[sftpd]
enabled = true
port = tcp:8022:interface=127.0.0.1
host_pubkey_file = private/ssh_host_rsa_key.pub
host_privkey_file = private/ssh_host_rsa_key
accounts.url = https://example.com/login
```

You can provide both accounts.file and accounts.url, although it probably isn't very useful except for testing.

For further information on SFTP compatibility and known issues with various clients and with the sshfs filesystem, see wiki:SftpFrontend

Configuring FTP Access

To enable the FTP server with an accounts file, add the following lines to the BASEDIR/tahoe.cfg file:

```
[ftpd]
enabled = true
port = tcp:8021:interface=127.0.0.1
accounts.file = private/accounts
```

The FTP server will listen on the given port number and on the loopback interface only. The "accounts.file" pathname will be interpreted relative to the node's BASEDIR.

To enable the FTP server with an account server instead, provide the URL of that server in an "accounts.url" directive:

```
[ftpd]
enabled = true
port = tcp:8021:interface=127.0.0.1
accounts.url = https://example.com/login
```

You can provide both accounts.file and accounts.url, although it probably isn't very useful except for testing.

FTP provides no security, and so your password or caps could be eavesdropped if you connect to the FTP server remotely. The examples above include ":interface=127.0.0.1" in the "port" option, which causes the server to only accept connections from localhost.

Public key authentication is not supported for FTP.

Dependencies

The Tahoe-LAFS SFTP server requires the Twisted "Conch" component (a "conch" is a twisted shell, get it?). Many Linux distributions package the Conch code separately: debian puts it in the "python-twisted-conch" package. Conch requires the "pycrypto" package, which is a Python+C implementation of many cryptographic functions (the debian package is named "python-crypto").

Note that "pycrypto" is different than the "pycryptopp" package that Tahoe-LAFS uses (which is a Python wrapper around the C++ -based Crypto++ library, a library that is frequently installed as /usr/lib/libcryptopp.a, to avoid problems with non-alphanumerics in filenames).

Immutable and Mutable Files

All files created via SFTP (and FTP) are immutable files. However, files can only be created in writeable directories, which allows the directory entry to be relinked to a different file. Normally, when the path of an immutable file is opened for writing by SFTP, the directory entry is relinked to another file with the newly written contents when the file handle is closed. The old file is still present on the grid, and any other caps to it will remain valid. (See *Garbage Collection in Tahoe* for how to reclaim the space used by files that are no longer needed.)

The 'no-write' metadata field of a directory entry can override this behaviour. If the 'no-write' field holds a true value, then a permission error will occur when trying to write to the file, even if it is in a writeable directory. This does not prevent the directory entry from being unlinked or replaced.

When using sshfs, the 'no-write' field can be set by clearing the 'w' bits in the Unix permissions, for example using the command `chmod 444 path/to/file`. Note that this does not mean that arbitrary combinations of Unix permissions are supported. If the 'w' bits are cleared on a link to a mutable file or directory, that link will become read-only.

If SFTP is used to write to an existing mutable file, it will publish a new version when the file handle is closed.

Known Issues

Known Issues in the SFTP Frontend

Upload errors may not be reported when writing files using SFTP via sshfs (ticket #1059).

Non-ASCII filenames are supported with SFTP only if the client encodes filenames as UTF-8 (ticket #1089).

See also wiki:SftpFrontend.

Known Issues in the FTP Frontend

Mutable files are not supported by the FTP frontend (ticket #680).

Non-ASCII filenames are not supported by FTP (ticket #682).

The FTP frontend sometimes fails to report errors, for example if an upload fails because it does meet the "servers of happiness" threshold (ticket #1081).

CHAPTER 10

Tahoe-LAFS Magic Folder Frontend

Introduction

The Magic Folder frontend synchronizes local directories on two or more clients, using 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 currently works on Linux and Windows.

Magic Folder is not currently in as mature a state as the other 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, and suggestions to improve its usability, functionality, and reliability.

Configuration

The Magic Folder frontend runs as part of a gateway node. To set it up, you must use the tahoe magic-folder CLI. For detailed information see our *Magic-Folder CLI design documentation*. For a given Magic-Folder collective directory you need to run the `tahoe magic-folder create` command. After that the `tahoe magic-folder`

`invite` command must used to generate an *invite code* for each member of the magic-folder collective. A confidential, authenticated communications channel should be used to transmit the invite code to each member, who will be joining using the `tahoe magic-folder join` command.

These settings are persisted in the `[magic_folder]` section of the gateway's `tahoe.cfg` file.

```
[magic_folder]

enabled = (boolean, optional)
```

If this is `True`, Magic Folder will be enabled. The default value is `False`.

```
local.directory = (UTF-8 path)
```

This specifies the local directory to be monitored for new or changed files. If the path contains non-ASCII characters, it should be encoded in UTF-8 regardless of the system's filesystem encoding. Relative paths will be interpreted starting from the node's base directory.

You should not normally need to set these fields manually because they are set by the `tahoe magic-folder create` and/or `tahoe magic-folder join` commands. Use the `--help` option to these commands for more information.

After setting up a Magic Folder collective and starting or restarting each gateway, you can confirm that the feature is working by copying a file into any local directory, and checking that it appears on other clients. Large files may take some time to appear.

The 'Operational Statistics' page linked from the Welcome page shows counts of the number of files uploaded, the number of change events currently queued, and the number of failed uploads. The 'Recent Uploads and Downloads' page and the node *log* may be helpful to determine the cause of any failures.

Known Issues and Limitations With Magic-Folder

This feature only works on Linux and Windows. There is a ticket to add support for Mac OS X and BSD-based systems (#1432).

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. So, 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.

The `private/magic_folder_dircap` and `private/collective_dircap` files cannot use an alias or path to specify the upload directory. (#1711)

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 in Tahoe* 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, when a node has Magic Folder enabled, it is unresponsive to Ctrl-C (it can only be killed using Task Manager or similar). (#2218)

CHAPTER 11

Download status

Introduction

The WUI will display the "status" of uploads and downloads.

The Welcome Page has a link entitled "Recent Uploads and Downloads" which goes to this URL:

http://\protect\T1\textdollarGATEWAY/status

Each entry in the list of recent operations has a "status" link which will take you to a page describing that operation.

For immutable downloads, the page has a lot of information, and this document is to explain what it all means. It was written by Brian Warner, who wrote the v1.8.0 downloader code and the code which generates this status report about the v1.8.0 downloader's behavior. Brian posted it to the trac: https://tahoe-lafs.org/trac/tahoe-lafs/ticket/1169#comment:1

Then Zooko lightly edited it while copying it into the docs/ directory.

What's involved in a download?

Downloads are triggered by read() calls, each with a starting offset (defaults to 0) and a length (defaults to the whole file). A regular web-API GET request will result in a whole-file read() call.

Each read() call turns into an ordered sequence of get_segment() calls. A whole-file read will fetch all segments, in order, but partial reads or multiple simultaneous reads will result in random-access of segments. Segment reads always return ciphertext: the layer above that (in read()) is responsible for decryption.

Before we can satisfy any segment reads, we need to find some shares. ("DYHB" is an abbreviation for "Do You Have Block", and is the message we send to storage servers to ask them if they have any shares for us. The name is historical, from Mojo Nation/Mnet/Mountain View, but nicely distinctive. Tahoe-LAFS's actual message name is remote_get_buckets().). Responses come back eventually, or don't.

Once we get enough positive DYHB responses, we have enough shares to start downloading. We send "block requests" for various pieces of the share. Responses come back eventually, or don't.

When we get enough block-request responses for a given segment, we can decode the data and satisfy the segment read.

When the segment read completes, some or all of the segment data is used to satisfy the read() call (if the read call started or ended in the middle of a segment, we'll only use part of the data, otherwise we'll use all of it).

Data on the download-status page

DYHB Requests

This shows every Do-You-Have-Block query sent to storage servers and their results. Each line shows the following:

- the serverid to which the request was sent
- the time at which the request was sent. Note that all timestamps are relative to the start of the first read() call and indicated with a "+" sign
- the time at which the response was received (if ever)
- the share numbers that the server has, if any
- the elapsed time taken by the request

Also, each line is colored according to the serverid. This color is also used in the "Requests" section below.

Read Events

This shows all the FileNode read() calls and their overall results. Each line shows:

- the range of the file that was requested (as [OFFSET:+LENGTH]). A whole-file GET will start at 0 and read the entire file.
- the time at which the read() was made
- the time at which the request finished, either because the last byte of data was returned to the read() caller, or because they cancelled the read by calling stopProducing (i.e. closing the HTTP connection)
- the number of bytes returned to the caller so far
- the time spent on the read, so far
- the total time spent in AES decryption
- total time spend paused by the client (pauseProducing), generally because the HTTP connection filled up, which most streaming media players will do to limit how much data they have to buffer
- effective speed of the read(), not including paused time

Segment Events

This shows each get_segment() call and its resolution. This table is not well organized, and my post-1.8.0 work will clean it up a lot. In its present form, it records "request" and "delivery" events separately, indicated by the "type" column.

Each request shows the segment number being requested and the time at which the get_segment() call was made.

Each delivery shows:

- segment number

- range of file data (as [OFFSET:+SIZE]) delivered
- elapsed time spent doing ZFEC decoding
- overall elapsed time fetching the segment
- effective speed of the segment fetch

Requests

This shows every block-request sent to the storage servers. Each line shows:

- the server to which the request was sent
- which share number it is referencing
- the portion of the share data being requested (as [OFFSET:+SIZE])
- the time the request was sent
- the time the response was received (if ever)
- the amount of data that was received (which might be less than SIZE if we tried to read off the end of the share)
- the elapsed time for the request (RTT=Round-Trip-Time)

Also note that each Request line is colored according to the serverid it was sent to. And all timestamps are shown relative to the start of the first read() call: for example the first DYHB message was sent at +0.001393s about 1.4 milliseconds after the read() call started everything off.

See also *cautions.rst*.

CHAPTER 12

Known Issues

Below is a list of known issues in recent releases of Tahoe-LAFS, and how to manage them. The current version of this file can be found at https://github.com/tahoe-lafs/tahoe-lafs/blob/master/docs/known_issues.rst .

If you've been using Tahoe-LAFS since v1.1 (released 2008-06-11) or if you're just curious about what sort of mistakes we've made in the past, then you might want to read the "historical known issues" document in `docs/historical/historical_known_issues.txt`.

Known Issues in Tahoe-LAFS v1.10.3, released 30-Mar-2016

Unauthorized access by JavaScript in unrelated files

If you view a file stored in Tahoe-LAFS through a web user interface, JavaScript embedded in that file can, in some circumstances, access other files or directories stored in Tahoe-LAFS that you view through the same web user interface. Such a script would be able to send the contents of those other files or directories to the author of the script, and if you have the ability to modify the contents of those files or directories, then that script could modify or delete those files or directories.

This attack is known to be possible when an attacking tab or window could reach a tab or window containing a Tahoe URI by navigating back or forward in the history, either from itself or from any frame with a known name (as specified by the "target" attribute of an HTML link). It might be possible in other cases depending on the browser.

how to manage it

For future versions of Tahoe-LAFS, we are considering ways to close off this leakage of authority while preserving ease of use – the discussion of this issue is ticket #615.

For the present, either do not view files stored in Tahoe-LAFS through a web user interface, or turn off JavaScript in your web browser before doing so, or limit your viewing to files which you know don't contain malicious JavaScript.

Disclosure of file through embedded hyperlinks or JavaScript in that file

If there is a file stored on a Tahoe-LAFS storage grid, and that file gets downloaded and displayed in a web browser, then JavaScript or hyperlinks within that file can leak the capability to that file to a third party, which means that third party gets access to the file.

If there is JavaScript in the file, then it could deliberately leak the capability to the file out to some remote listener.

If there are hyperlinks in the file, and they get followed, then whichever server they point to receives the capability to the file. Note that IMG tags are typically followed automatically by web browsers, so being careful which hyperlinks you click on is not sufficient to prevent this from happening.

how to manage it

For future versions of Tahoe-LAFS, we are considering ways to close off this leakage of authority while preserving ease of use – the discussion of this issue is ticket #127.

For the present, a good work-around is that if you want to store and view a file on Tahoe-LAFS and you want that file to remain private, then remove from that file any hyperlinks pointing to other people's servers and remove any JavaScript unless you are sure that the JavaScript is not written to maliciously leak access.

Command-line arguments are leaked to other local users

Remember that command-line arguments are visible to other users (through the 'ps' command, or the windows Process Explorer tool), so if you are using a Tahoe-LAFS node on a shared host, other users on that host will be able to see (and copy) any caps that you pass as command-line arguments. This includes directory caps that you set up with the "tahoe add-alias" command.

how to manage it

As of Tahoe-LAFS v1.3.0 there is a "tahoe create-alias" command that does the following technique for you.

Bypass add-alias and edit the NODEDIR/private/aliases file directly, by adding a line like this:

```
fun: URI:DIR2:ovjy4yhylqlfoqg2vcze36dhde:4d4f47qko2xm5g7osgo2yyidi5m4muyo2vjjy53q4vjju2u55mfa
```

By entering the dircap through the editor, the command-line arguments are bypassed, and other users will not be able to see them. Once you've added the alias, if you use that alias instead of a cap itself on the command-line, then no secrets are passed through the command line. Then other processes on the system can still see your filenames and other arguments you type there, but not the caps that Tahoe-LAFS uses to permit access to your files and directories.

Capabilities may be leaked to web browser phishing filter / "safe browsing" servers

Firefox, Internet Explorer, and Chrome include a "phishing filter" or "safe browing" component, which is turned on by default, and which sends any URLs that it deems suspicious to a central server.

Microsoft gives a brief description of their filter's operation. Firefox and Chrome both use Google's "safe browsing API" (specification).

This of course has implications for the privacy of general web browsing (especially in the cases of Firefox and Chrome, which send your main personally identifying Google cookie along with these requests without your explicit consent, as described in Firefox bugzilla ticket #368255.

The reason for documenting this issue here, though, is that when using the Tahoe-LAFS web user interface, it could also affect confidentiality and integrity by leaking capabilities to the filter server.

Since IE's filter sends URLs by SSL/TLS, the exposure of caps is limited to the filter server operators (or anyone able to hack the filter server) rather than to network eavesdroppers. The "safe browsing API" protocol used by Firefox and Chrome, on the other hand, is *not* encrypted, although the URL components are normally hashed.

Opera also has a similar facility that is disabled by default. A previous version of this file stated that Firefox had abandoned their phishing filter; this was incorrect.

how to manage it

If you use any phishing filter or "safe browsing" feature, consider either disabling it, or not using the WUI via that browser. Phishing filters have very limited effectiveness , and phishing or malware attackers have learnt how to bypass them.

To disable the filter in IE7 or IE8:

- Click Internet Options from the Tools menu.
- Click the Advanced tab.
- If an "Enable SmartScreen Filter" option is present, uncheck it. If a "Use Phishing Filter" or "Phishing Filter" option is present, set it to Disable.
- Confirm (click OK or Yes) out of all dialogs.

If you have a version of IE that splits the settings between security zones, do this for all zones.

To disable the filter in Firefox:

- Click Options from the Tools menu.
- Click the Security tab.
- Uncheck both the "Block reported attack sites" and "Block reported web forgeries" options.
- Click OK.

To disable the filter in Chrome:

- Click Options from the Tools menu.
- Click the "Under the Hood" tab and find the "Privacy" section.
- Uncheck the "Enable phishing and malware protection" option.
- Click Close.

Known issues in the FTP and SFTP frontends

These are documented in *Tahoe-LAFS SFTP and FTP Frontends* and on the SftpFrontend page on the wiki.

Traffic analysis based on sizes of files/directories, storage indices, and timing

Files and directories stored by Tahoe-LAFS are encrypted, but the ciphertext reveals the exact size of the original file or directory representation. This information is available to passive eavesdroppers and to server operators.

For example, a large data set with known file sizes could probably be identified with a high degree of confidence.

Uploads and downloads of the same file or directory can be linked by server operators, even without making assumptions based on file size. Anyone who knows the introducer furl for a grid may be able to act as a server operator. This implies that if such an attacker knows which file/directory is being accessed in a particular request (by some other form of surveillance, say), then they can identify later or earlier accesses of the same file/directory.

Observing requests during a directory traversal (such as a deep-check operation) could reveal information about the directory structure, i.e. which files and subdirectories are linked from a given directory.

Attackers can combine the above information with inferences based on timing correlations. For instance, two files that are accessed close together in time are likely to be related even if they are not linked in the directory structure. Also, users that access the same files may be related to each other.

Privacy leak via Google Chart API link in map-update timing web page

The Tahoe web-based user interface includes a diagnostic page known as the "map-update timing page". It is reached through the "Recent and Active Operations" link on the front welcome page, then through the "Status" column for "map-update" operations (which occur when mutable files, including directories, are read or written). This page contains per-server response times, as lines of text, and includes an image which displays the response times in graphical form. The image is generated by constructing a URL for the Google Chart API, which is then served by the *chart.apis.google.com* internet server.

When you view this page, several parties may learn information about your Tahoe activities. The request will typically include a "Referer" header, revealing the URL of the mapupdate status page (which is typically something like "http://127.0.0.1:3456/status/mapupdate-123") to network observers and the Google API server. The image returned by this server is typically a PNG file, but either the server or a MitM attacker could replace it with something malicious that attempts to exploit a browser rendering bug or buffer overflow. (Note that browsers do not execute scripts inside IMG tags, even for SVG images).

In addition, if your Tahoe node connects to its grid over Tor or i2p, but the web browser you use to access your node does not, then this image link may reveal your use of Tahoe (and that grid) to the outside world. It is not recommended to use a browser in this way, because other links in Tahoe-stored content would reveal even more information (e.g. an attacker could store an HTML file with unique CSS references into a shared Tahoe grid, then send your pseudonym a message with its URI, then observe your browser loading that CSS file, and thus link the source IP address of your web client to that pseudonym).

A future version of Tahoe will probably replace the Google Chart API link (which was deprecated by Google in April 2012) with client-side javascript using d3.js, removing the information leak but requiring JS to see the chart. See ticket #1942 for details.

Known Issues in Tahoe-LAFS v1.9.0, released 31-Oct-2011

Integrity Failure during Mutable Downloads

Under certain circumstances, the integrity-verification code of the mutable downloader could be bypassed. Clients who receive carefully crafted shares (from attackers) will emit incorrect file contents, and the usual share-corruption errors would not be raised. This only affects mutable files (not immutable), and only affects downloads that use doctored shares. It is not persistent: the threat is resolved once you upgrade your client to a version without the bug. However, read-modify-write operations (such as directory manipulations) performed by vulnerable clients could cause the attacker's modifications to be written back out to the mutable file, making the corruption permanent.

The attacker's ability to manipulate the file contents is limited. They can modify FEC-encoded ciphertext in all but one share. This gives them the ability to blindly flip bits in roughly 2/3rds of the file (for the default k=3 encoding parameter). Confidentiality remains intact, unless the attacker can deduce the file's contents by observing your reactions to corrupted downloads.

This bug was introduced in 1.9.0, as part of the MDMF-capable downloader, and affects both SDMF and MDMF files. It was not present in 1.8.3.

how to manage it

There are three options:

- Upgrade to 1.9.1, which fixes the bug
- Downgrade to 1.8.3, which does not contain the bug
- If using 1.9.0, do not trust the contents of mutable files (whether SDMF or MDMF) that the 1.9.0 client emits, and do not modify directories (which could write the corrupted data back into place, making the damage persistent)

Known Issues in Tahoe-LAFS v1.8.2, released 30-Jan-2011

Unauthorized deletion of an immutable file by its storage index

Due to a flaw in the Tahoe-LAFS storage server software in v1.3.0 through v1.8.2, a person who knows the "storage index" that identifies an immutable file can cause the server to delete its shares of that file.

If an attacker can cause enough shares to be deleted from enough storage servers, this deletes the file.

This vulnerability does not enable anyone to read file contents without authorization (confidentiality), nor to change the contents of a file (integrity).

A person could learn the storage index of a file in several ways:

1. By being granted the authority to read the immutable file: i.e. by being granted a read capability to the file. They can determine the file's storage index from its read capability.
2. By being granted a verify capability to the file. They can determine the file's storage index from its verify capability. This case probably doesn't happen often because users typically don't share verify caps.
3. By operating a storage server, and receiving a request from a client that has a read cap or a verify cap. If the client attempts to upload, download, or verify the file with their storage server, even if it doesn't actually have the file, then they can learn the storage index of the file.

4. By gaining read access to an existing storage server's local filesystem, and inspecting the directory structure that it stores its shares in. They can thus learn the storage indexes of all files that the server is holding at least one share of. Normally only the operator of an existing storage server would be able to inspect its local filesystem, so this requires either being such an operator of an existing storage server, or somehow gaining the ability to inspect the local filesystem of an existing storage server.

how to manage it

Tahoe-LAFS version v1.8.3 or newer (except v1.9a1) no longer has this flaw; if you upgrade a storage server to a fixed release then that server is no longer vulnerable to this problem.

Note that the issue is local to each storage server independently of other storage servers: when you upgrade a storage server then that particular storage server can no longer be tricked into deleting its shares of the target file.

If you can't immediately upgrade your storage server to a version of Tahoe-LAFS that eliminates this vulnerability, then you could temporarily shut down your storage server. This would of course negatively impact availability – clients would not be able to upload or download shares to that particular storage server while it was shut down – but it would protect the shares already stored on that server from being deleted as long as the server is shut down.

If the servers that store shares of your file are running a version of Tahoe-LAFS with this vulnerability, then you should think about whether someone can learn the storage indexes of your files by one of the methods described above. A person can not exploit this vulnerability unless they have received a read cap or verify cap, or they control a storage server that has been queried about this file by a client that has a read cap or a verify cap.

Tahoe-LAFS does not currently have a mechanism to limit which storage servers can connect to your grid, but it does have a way to see which storage servers have been connected to the grid. The Introducer's front page in the Web User Interface has a list of all storage servers that the Introducer has ever seen and the first time and the most recent time that it saw them. Each Tahoe-LAFS gateway maintains a similar list on its front page in its Web User Interface, showing all of the storage servers that it learned about from the Introducer, when it first connected to that storage server, and when it most recently connected to that storage server. These lists are stored in memory and are reset to empty when the process is restarted.

See ticket #1528 for technical details.

CHAPTER 13

How To Configure A Server

Many Tahoe-LAFS nodes run as "servers", meaning they provide services for other machines (i.e. "clients"). The two most important kinds are the Introducer, and Storage Servers.

To be useful, servers must be reachable by clients. Tahoe servers can listen on TCP ports, and advertise their "location" (hostname and TCP port number) so clients can connect to them. They can also listen on Tor "onion services" and I2P ports.

Storage servers advertise their location by announcing it to the Introducer, which then broadcasts the location to all clients. So once the location is determined, you don't need to do anything special to deliver it.

The Introducer itself has a location, which must be manually delivered to all storage servers and clients. You might email it to the new members of your grid. This location (along with other important cryptographic identifiers) is written into a file named `private/introducer.furl` in the Introducer's base directory, and should be provided as the `--introducer=` argument to `tahoe create-client` or `tahoe create-node`.

The first step when setting up a server is to figure out how clients will reach it. Then you need to configure the server to listen on some ports, and then configure the location properly.

Manual Configuration

Each server has two settings in their `tahoe.cfg` file: `tub.port`, and `tub.location`. The "port" controls what the server node listens to: this is generally a TCP port.

The "location" controls what is advertised to the outside world. This is a "foolscap connection hint", and it includes both the type of the connection (tcp, tor, or i2p) and the connection details (hostname/address, port number). Various proxies, port-forwardings, and privacy networks might be involved, so it's not uncommon for `tub.port` and `tub.location` to look different.

You can directly control the `tub.port` and `tub.location` configuration settings by providing `--port=` and `--location=` when running `tahoe create-node`.

Automatic Configuration

Instead of providing `--port=/--location=`, you can use `--listen=`. Servers can listen on TCP, Tor, I2P, a combination of those, or none at all. The `--listen=` argument controls which kinds of listeners the new server will use.

`--listen=none` means the server should not listen at all. This doesn't make sense for a server, but is appropriate for a client-only node. The `tahoe create-client` command automatically includes `--listen=none`.

`--listen=tcp` is the default, and turns on a standard TCP listening port. Using `--listen=tcp` requires a `--hostname=` argument too, which will be incorporated into the node's advertised location. We've found that computers cannot reliably determine their externally-reachable hostname, so rather than having the server make a guess (or scanning its interfaces for IP addresses that might or might not be appropriate), node creation requires the user to provide the hostname.

`--listen=tor` will talk to a local Tor daemon and create a new "onion server" address (which look like `alzrgrdvxct6c63z.onion`). Likewise `--listen=i2p` will talk to a local I2P daemon and create a new server address. See *Using Tahoe-LAFS with an anonymizing network: Tor, I2P* for details.

You could listen on all three by using `--listen=tcp,tor,i2p`.

Deployment Scenarios

The following are some suggested scenarios for configuring servers using various network transports. These examples do not include specifying an introducer FURL which normally you would want when provisioning storage nodes. For these and other configuration details please refer to *Configuring a Tahoe-LAFS node*.

1. *Server has a public DNS name*
2. *Server has a public IPv4/IPv6 address*
3. *Server is behind a firewall with port forwarding*
4. *Using I2P/Tor to Avoid Port-Forwarding*

Server has a public DNS name

The simplest case is where your server host is directly connected to the internet, without a firewall or NAT box in the way. Most VPS (Virtual Private Server) and colocated servers are like this, although some providers block many inbound ports by default.

For these servers, all you need to know is the external hostname. The system administrator will tell you this. The main requirement is that this hostname can be looked up in DNS, and it will map to an IPv4 or IPv6 address which will reach the machine.

If your hostname is `example.net`, then you'll create the introducer like this:

```
tahoe create-introducer --hostname example.com ~/introducer
```

or a storage server like:

```
tahoe create-node --hostname=example.net
```

These will allocate a TCP port (e.g. 12345), assign `tub.port` to be `tcp:12345`, and `tub.location` will be `tcp:example.com:12345`.

Ideally this should work for IPv6-capable hosts too (where the DNS name provides an "AAAA" record, or both "A" and "AAAA"). However Tahoe-LAFS support for IPv6 is new, and may still have problems. Please see ticket #867 for details.

Server has a public IPv4/IPv6 address

If the host has a routeable (public) IPv4 address (e.g. `203.0.113.1`), but no DNS name, you will need to choose a TCP port (e.g. `3457`), and use the following:

```
tahoe create-node --port=tcp:3457 --location=tcp:203.0.113.1:3457
```

`--port` is an "endpoint specification string" that controls which local port the node listens on. `--location` is the "connection hint" that it advertises to others, and describes the outbound connections that those clients will make, so it needs to work from their location on the network.

Tahoe-LAFS nodes listen on all interfaces by default. When the host is multi-homed, you might want to make the listening port bind to just one specific interface by adding a `interface=` option to the `--port=` argument:

```
tahoe create-node --port=tcp:3457:interface=203.0.113.1 --location=tcp:203.0.113.
↪1:3457
```

If the host's public address is IPv6 instead of IPv4, use square brackets to wrap the address, and change the endpoint type to `tcp6`:

```
tahoe create-node --port=tcp6:3457 --location=tcp:[2001:db8::1]:3457
```

You can use `interface=` to bind to a specific IPv6 interface too, however you must backslash-escape the colons, because otherwise they are interpreted as delimiters by the Twisted "endpoint" specification language. The `--location=` argument does not need colons to be escaped, because they are wrapped by the square brackets:

```
tahoe create-node --port=tcp6:3457:interface=2001\:db8\:\:1 --
↪location=tcp:[2001:db8::1]:3457
```

For IPv6-only hosts with AAAA DNS records, if the simple `--hostname=` configuration does not work, they can be told to listen specifically on an IPv6-enabled port with this:

```
tahoe create-node --port=tcp6:3457 --location=tcp:example.net:3457
```

Server is behind a firewall with port forwarding

To configure a storage node behind a firewall with port forwarding you will need to know:

- public IPv4 address of the router
- the TCP port that is available from outside your network
- the TCP port that is the forwarding destination
- internal IPv4 address of the storage node (the storage node itself is unaware of this address, and it is not used during `tahoe create-node`, but the firewall must be configured to send connections to this)

The internal and external TCP port numbers could be the same or different depending on how the port forwarding is configured. If it is mapping ports 1-to-1, and the public IPv4 address of the firewall is 203.0.113.1 (and perhaps the internal IPv4 address of the storage node is 192.168.1.5), then use a CLI command like this:

```
tahoe create-node --port=tcp:3457 --location=tcp:203.0.113.1:3457
```

If however the firewall/NAT-box forwards external port *6656* to internal port 3457, then do this:

```
tahoe create-node --port=tcp:3457 --location=tcp:203.0.113.1:6656
```

Using I2P/Tor to Avoid Port-Forwarding

I2P and Tor onion services, among other great properties, also provide NAT penetration without port-forwarding, hostnames, or IP addresses. So setting up a server that listens only on Tor is simple:

```
tahoe create-node --listen=tor
```

For more information about using Tahoe-LAFS with I2p and Tor see *Using Tahoe-LAFS with an anonymizing network: Tor, I2P*

CHAPTER 14

The Tahoe Upload Helper

Overview

As described in the "Swarming Download, Trickling Upload" section of *Tahoe-LAFS Architecture*, Tahoe uploads require more bandwidth than downloads: you must push the redundant shares during upload, but you do not need to retrieve them during download. With the default 3-of-10 encoding parameters, this means that an upload will require about 3.3x the traffic as a download of the same file.

Unfortunately, this "expansion penalty" occurs in the same upstream direction that most consumer DSL lines are slow anyways. Typical ADSL lines get 8 times as much download capacity as upload capacity. When the ADSL upstream penalty is combined with the expansion penalty, the result is uploads that can take up to 32 times longer than downloads.

The "Helper" is a service that can mitigate the expansion penalty by arranging for the client node to send data to a central Helper node instead of sending it directly to the storage servers. It sends ciphertext to the Helper, so the security properties remain the same as with non-Helper uploads. The Helper is responsible for applying the erasure encoding algorithm and placing the resulting shares on the storage servers.

Of course, the helper cannot mitigate the ADSL upstream penalty.

The second benefit of using an upload helper is that clients who lose their network connections while uploading a file (because of a network flap, or because they shut down their laptop while an upload was in progress) can resume their upload rather than needing to start again from scratch. The helper holds the partially-uploaded ciphertext on disk, and when the client tries to upload the same file a second time, it discovers that the partial ciphertext is already present. The client then only needs to upload the remaining ciphertext. This reduces the "interrupted upload penalty" to a minimum.

This also serves to reduce the number of active connections between the client and the outside world: most of their traffic flows over a single TCP connection to the helper. This can improve TCP fairness, and should allow other applications that are sharing the same uplink to compete more evenly for the limited bandwidth.

Setting Up A Helper

Who should consider running a helper?

- Benevolent entities which wish to provide better upload speed for clients that have slow uplinks
- Folks which have machines with upload bandwidth to spare.
- Server grid operators who want clients to connect to a small number of helpers rather than a large number of storage servers (a "multi-tier" architecture)

What sorts of machines are good candidates for running a helper?

- The Helper needs to have good bandwidth to the storage servers. In particular, it needs to have at least 3.3x better upload bandwidth than the client does, or the client might as well upload directly to the storage servers. In a commercial grid, the helper should be in the same colo (and preferably in the same rack) as the storage servers.
- The Helper will take on most of the CPU load involved in uploading a file. So having a dedicated machine will give better results.
- The Helper buffers ciphertext on disk, so the host will need at least as much free disk space as there will be simultaneous uploads. When an upload is interrupted, that space will be used for a longer period of time.

To turn a Tahoe-LAFS node into a helper (i.e. to run a helper service in addition to whatever else that node is doing), edit the tahoe.cfg file in your node's base directory and set "enabled = true" in the section named "[helper]".

Then restart the node. This will signal the node to create a Helper service and listen for incoming requests. Once the node has started, there will be a file named private/helper.furl which contains the contact information for the helper: you will need to give this FURL to any clients that wish to use your helper.

```
cat $BASEDIR/private/helper.furl | mail -s "helper furl" friend@example.com
```

You can tell if your node is running a helper by looking at its web status page. Assuming that you've set up the 'webport' to use port 3456, point your browser at `http://localhost:3456/`. The welcome page will say "Helper: 0 active uploads" or "Not running helper" as appropriate. The http://localhost:3456/helper_status page will also provide details on what the helper is currently doing.

The helper will store the ciphertext that is is fetching from clients in $BASEDIR/helper/CHK_incoming/. Once all the ciphertext has been fetched, it will be moved to $BASEDIR/helper/CHK_encoding/ and erasure-coding will commence. Once the file is fully encoded and the shares are pushed to the storage servers, the ciphertext file will be deleted.

If a client disconnects while the ciphertext is being fetched, the partial ciphertext will remain in CHK_incoming/ until they reconnect and finish sending it. If a client disconnects while the ciphertext is being encoded, the data will remain in CHK_encoding/ until they reconnect and encoding is finished. For long-running and busy helpers, it may be a good idea to delete files in these directories that have not been modified for a week or two. Future versions of tahoe will try to self-manage these files a bit better.

Using a Helper

Who should consider using a Helper?

- clients with limited upstream bandwidth, such as a consumer ADSL line
- clients who believe that the helper will give them faster uploads than they could achieve with a direct upload
- clients who experience problems with TCP connection fairness: if other programs or machines in the same home are getting less than their fair share of upload bandwidth. If the connection is being shared fairly, then a Tahoe upload that is happening at the same time as a single FTP upload should get half the bandwidth.
- clients who have been given the helper.furl by someone who is running a Helper and is willing to let them use it

To take advantage of somebody else's Helper, take the helper furl that they give you, and edit your tahoe.cfg file. Enter the helper's furl into the value of the key "helper.furl" in the "[client]" section of tahoe.cfg, as described in the "Client Configuration" section of *Configuring a Tahoe-LAFS node*.

Then restart the node. This will signal the client to try and connect to the helper. Subsequent uploads will use the helper rather than using direct connections to the storage server.

If the node has been configured to use a helper, that node's HTTP welcome page (`http://localhost:3456/`) will say "Helper: $HELPERFURL" instead of "Helper: None". If the helper is actually running and reachable, the bullet to the left of "Helper" will be green.

The helper is optional. If a helper is connected when an upload begins, the upload will use the helper. If there is no helper connection present when an upload begins, that upload will connect directly to the storage servers. The client will automatically attempt to reconnect to the helper if the connection is lost, using the same exponential-backoff algorithm as all other tahoe/foolscap connections.

The upload/download status page (`http://localhost:3456/status`) will announce the using-helper-or-not state of each upload, in the "Helper?" column.

Other Helper Modes

The Tahoe Helper only currently helps with one kind of operation: uploading immutable files. There are three other things it might be able to help with in the future:

- downloading immutable files
- uploading mutable files (such as directories)
- downloading mutable files (like directories)

Since mutable files are currently limited in size, the ADSL upstream penalty is not so severe for them. There is no ADSL penalty to downloads, but there may still be benefit to extending the helper interface to assist with them: fewer connections to the storage servers, and better TCP fairness.

A future version of the Tahoe helper might provide assistance with these other modes. If it were to help with all four modes, then the clients would not need direct connections to the storage servers at all: clients would connect to helpers, and helpers would connect to servers. For a large grid with tens of thousands of clients, this might make the grid more scalable.

CHAPTER 15

The Convergence Secret

What Is It?

The identifier of a file (also called the "capability" to a file) is derived from two pieces of information when the file is uploaded: the content of the file and the upload client's "convergence secret". By default, the convergence secret is randomly generated by the client when it first starts up, then stored in the client's base directory (<Tahoe's node dir>/private/convergence) and re-used after that. So the same file content uploaded from the same client will always have the same cap. Uploading the file from a different client with a different convergence secret would result in a different cap – and in a second copy of the file's contents stored on the grid. If you want files you upload to converge (also known as "deduplicate") with files uploaded by someone else, just make sure you're using the same convergence secret when you upload files as them.

The advantages of deduplication should be clear, but keep in mind that the convergence secret was created to protect confidentiality. There are two attacks that can be used against you by someone who knows the convergence secret you use.

The first one is called the "Confirmation-of-a-File Attack". Someone who knows the convergence secret that you used when you uploaded a file, and who has a copy of that file themselves, can check whether you have a copy of that file. This is usually not a problem, but it could be if that file is, for example, a book or movie that is banned in your country.

The second attack is more subtle. It is called the "Learn-the-Remaining-Information Attack". Suppose you've received a confidential document, such as a PDF from your bank which contains many pages of boilerplate text as well as containing your bank account number and balance. Someone who knows your convergence secret can generate a file with all of the boilerplate text (perhaps they would open an account with the same bank so they receive the same document with their account number and balance). Then they can try a "brute force search" to find your account number and your balance.

The defense against these attacks is that only someone who knows the convergence secret that you used on each file can perform these attacks on that file.

Both of these attacks and the defense are described in more detail in Drew Perttula's Hack Tahoe-LAFS Hall Of Fame entry

What If I Change My Convergence Secret?

All your old file capabilities will still work, but the new data that you upload will not be deduplicated with the old data. If you upload all of the same things to the grid, you will end up using twice the space until garbage collection kicks in (if it's enabled). Changing the convergence secret that a storage client uses for uploads can be though of as moving the client to a new "deduplication domain".

How To Use It

To enable deduplication between different clients, **securely** copy the convergence secret file from one client to all the others.

For example, if you are on host A and have an account on host B and you have scp installed, run:

scp ~/.tahoe/private/convergence my_other_account@B:.tahoe/private/convergence

If you have two different clients on a single computer, say one for each disk, you would do:

cp /tahoe1/private/convergence /tahoe2/private/convergence

After you change the convergence secret file, you must restart the client before it will stop using the old one and read the new one from the file.

CHAPTER 16

Garbage Collection in Tahoe

Overview

When a file or directory in a Tahoe-LAFS file store is no longer referenced, the space that its shares occupied on each storage server can be freed, making room for other shares. Tahoe currently uses a garbage collection ("GC") mechanism to implement this space-reclamation process. Each share has one or more "leases", which are managed by clients who want the file/directory to be retained. The storage server accepts each share for a pre-defined period of time, and is allowed to delete the share if all of the leases expire.

Garbage collection is not enabled by default: storage servers will not delete shares without being explicitly configured to do so. When GC is enabled, clients are responsible for renewing their leases on a periodic basis at least frequently enough to prevent any of the leases from expiring before the next renewal pass.

There are several tradeoffs to be considered when choosing the renewal timer and the lease duration, and there is no single optimal pair of values. See the following diagram to get an idea of the tradeoffs involved:

If lease renewal occurs quickly and with 100% reliability, than any renewal time that is shorter than the lease duration will suffice, but a larger ratio of duration-over-renewal-time will be more robust in the face of occasional delays or failures.

The current recommended values for a small Tahoe grid are to renew the leases once a week, and give each lease a duration of 31 days. In the current release, there is not yet a way to create a lease with a different duration, but the server can use the `expire.override_lease_duration` configuration setting to increase or decrease the effective duration (when the lease is processed) to something other than 31 days.

Renewing leases can be expected to take about one second per file/directory, depending upon the number of servers and the network speeds involved.

Client-side Renewal

If all of the files and directories which you care about are reachable from a single starting point (usually referred to as a "rootcap"), and you store that rootcap as an alias (via "`tahoe create-alias`" for example), then the simplest way to renew these leases is with the following CLI command:

```
tahoe deep-check --add-lease ALIAS:
```

This will recursively walk every directory under the given alias and renew the leases on all files and directories. (You may want to add a `--repair` flag to perform repair at the same time.) Simply run this command once a week (or whatever other renewal period your grid recommends) and make sure it completes successfully. As a side effect, a manifest of all unique files and directories will be emitted to stdout, as well as a summary of file sizes and counts. It may be useful to track these statistics over time.

Note that newly uploaded files (and newly created directories) get an initial lease too: the `--add-lease` process is only needed to ensure that all older objects have up-to-date leases on them.

A separate "rebalancing manager/service" is also planned – see ticket #543. The exact details of what this service will do are not settled, but it is likely to work by acquiring manifests from rootcaps on a periodic basis, keeping track of checker results, managing lease-addition, and prioritizing repair and rebalancing of shares. Eventually it may use multiple worker nodes to perform these jobs in parallel.

Server Side Expiration

Expiration must be explicitly enabled on each storage server, since the default behavior is to never expire shares. Expiration is enabled by adding config keys to the `[storage]` section of the `tahoe.cfg` file (as described below) and restarting the server node.

Each lease has two parameters: a create/renew timestamp and a duration. The timestamp is updated when the share is first uploaded (i.e. the file or directory is created), and updated again each time the lease is renewed (i.e. "`tahoe check --add-lease`" is performed). The duration is currently fixed at 31 days, and the "nominal lease expiration time" is simply $duration seconds after the $create_renew timestamp. (In a future release of Tahoe, the client will get to request a specific duration, and the server will accept or reject the request depending upon its local configuration, so that servers can achieve better control over their storage obligations.)

The lease-expiration code has two modes of operation. The first is age-based: leases are expired when their age is greater than their duration. This is the preferred mode: as long as clients consistently update their leases on a periodic basis, and that period is shorter than the lease duration, then all active files and directories will be preserved, and the garbage will collected in a timely fashion.

Since there is not yet a way for clients to request a lease duration of other than 31 days, there is a `tahoe.cfg` setting to override the duration of all leases. If, for example, this alternative duration is set to 60 days, then clients could safely renew their leases with an add-lease operation perhaps once every 50 days: even though nominally their leases would expire 31 days after the renewal, the server would not actually expire the leases until 60 days after renewal.

The other mode is an absolute-date-cutoff: it compares the create/renew timestamp against some absolute date, and expires any lease which was not created or renewed since the cutoff date. If all clients have performed an add-lease some time after March 20th, you could tell the storage server to expire all leases that were created or last renewed on March 19th or earlier. This is most useful if you have a manual (non-periodic) add-lease process. Note that there is not much point to running a storage server in this mode for a long period of time: once the lease-checker has examined

all shares and expired whatever it is going to expire, the second and subsequent passes are not going to find any new leases to remove.

The `tahoe.cfg` file uses the following keys to control lease expiration:

`[storage]`

`expire.enabled = (boolean, optional)`

If this is `True`, the storage server will delete shares on which all leases have expired. Other controls dictate when leases are considered to have expired. The default is `False`.

`expire.mode = (string, "age" or "cutoff-date", required if expiration enabled)`

If this string is "age", the age-based expiration scheme is used, and the `expire.override_lease_duration` setting can be provided to influence the lease ages. If it is "cutoff-date", the absolute-date-cutoff mode is used, and the `expire.cutoff_date` setting must be provided to specify the cutoff date. The mode setting currently has no default: you must provide a value.

In a future release, this setting is likely to default to "age", but in this release it was deemed safer to require an explicit mode specification.

`expire.override_lease_duration = (duration string, optional)`

When age-based expiration is in use, a lease will be expired if its `lease.create_renew` timestamp plus its `lease.duration` time is earlier/older than the current time. This key, if present, overrides the duration value for all leases, changing the algorithm from:

```
if (lease.create_renew_timestamp + lease.duration) < now:
    expire_lease()
```

to:

```
if (lease.create_renew_timestamp + override_lease_duration) < now:
    expire_lease()
```

The value of this setting is a "duration string", which is a number of days, months, or years, followed by a units suffix, and optionally separated by a space, such as one of the following:

```
7days
31day
60 days
2mo
3 month
12 months
2years
```

This key is meant to compensate for the fact that clients do not yet have the ability to ask for leases that last longer than 31 days. A grid which wants to use faster or slower GC than a 31-day lease timer permits can use this parameter to implement it.

This key is only valid when age-based expiration is in use (i.e. when `expire.mode = age` is used). It will be rejected if cutoff-date expiration is in use.

`expire.cutoff_date = (date string, required if mode=cutoff-date)`

When cutoff-date expiration is in use, a lease will be expired if its create/renew timestamp is older than the cutoff date. This string will be a date in the following format:

```
2009-01-16   (January 16th, 2009)
2008-02-02
2007-12-25
```

The actual cutoff time shall be midnight UTC at the beginning of the given day. Lease timers should naturally be generous enough to not depend upon differences in timezone: there should be at least a few days between the last renewal time and the cutoff date.

This key is only valid when cutoff-based expiration is in use (i.e. when "expire.mode = cutoff-date"). It will be rejected if age-based expiration is in use.

expire.immutable = (boolean, optional)

If this is False, then immutable shares will never be deleted, even if their leases have expired. This can be used in special situations to perform GC on mutable files but not immutable ones. The default is True.

expire.mutable = (boolean, optional)

If this is False, then mutable shares will never be deleted, even if their leases have expired. This can be used in special situations to perform GC on immutable files but not mutable ones. The default is True.

Expiration Progress

In the current release, leases are stored as metadata in each share file, and no separate database is maintained. As a result, checking and expiring leases on a large server may require multiple reads from each of several million share files. This process can take a long time and be very disk-intensive, so a "share crawler" is used. The crawler limits the amount of time looking at shares to a reasonable percentage of the storage server's overall usage: by default it uses no more than 10% CPU, and yields to other code after 100ms. A typical server with 1.1M shares was observed to take 3.5 days to perform this rate-limited crawl through the whole set of shares, with expiration disabled. It is expected to take perhaps 4 or 5 days to do the crawl with expiration turned on.

The crawler's status is displayed on the "Storage Server Status Page", a web page dedicated to the storage server. This page resides at $NODEURL/storage, and there is a link to it from the front "welcome" page. The "Lease Expiration crawler" section of the status page shows the progress of the current crawler cycle, expected completion time, amount of space recovered, and details of how many shares have been examined.

The crawler's state is persistent: restarting the node will not cause it to lose significant progress. The state file is located in two files ($BASEDIR/storage/lease_checker.state and lease_checker.history), and the crawler can be forcibly reset by stopping the node, deleting these two files, then restarting the node.

Future Directions

Tahoe's GC mechanism is undergoing significant changes. The global mark-and-sweep garbage-collection scheme can require considerable network traffic for large grids, interfering with the bandwidth available for regular uploads and downloads (and for non-Tahoe users of the network).

A preferable method might be to have a timer-per-client instead of a timer-per-lease: the leases would not be expired until/unless the client had not checked in with the server for a pre-determined duration. This would reduce the network traffic considerably (one message per week instead of thousands), but retain the same general failure characteristics.

In addition, using timers is not fail-safe (from the client's point of view), in that a client which leaves the network for an extended period of time may return to discover that all of their files have been garbage-collected. (It *is* fail-safe from the server's point of view, in that a server is not obligated to provide disk space in perpetuity to an unresponsive client). It may be useful to create a "renewal agent" to which a client can pass a list of renewal-caps: the agent then takes the responsibility for keeping these leases renewed, so the client can go offline safely. Of course, this requires a certain amount of coordination: the renewal agent should not be keeping files alive that the client has actually deleted.

The client can send the renewal-agent a manifest of renewal caps, and each new manifest should replace the previous set.

The GC mechanism is also not immediate: a client which deletes a file will nevertheless be consuming extra disk space (and might be charged or otherwise held accountable for it) until the ex-file's leases finally expire on their own.

In the current release, these leases are each associated with a single "node secret" (stored in $BASEDIR/private/secret), which is used to generate renewal-secrets for each lease. Two nodes with different secrets will produce separate leases, and will not be able to renew each others' leases.

Once the Accounting project is in place, leases will be scoped by a sub-delegatable "account id" instead of a node secret, so clients will be able to manage multiple leases per file. In addition, servers will be able to identify which shares are leased by which clients, so that clients can safely reconcile their idea of which files/directories are active against the server's list, and explicitly cancel leases on objects that aren't on the active list.

By reducing the size of the "lease scope", the coordination problem is made easier. In general, mark-and-sweep is easier to implement (it requires mere vigilance, rather than coordination), so unless the space used by deleted files is not expiring fast enough, the renew/expire timed lease approach is recommended.

CHAPTER 17

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 Tahoe developers promise never to change Tahoe-LAFS 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 Tahoe-LAFS' 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 Tahoe-LAFS 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 Tahoe-LAFS developers agree with this statement:

David-Sarah Hopwood [Daira Hopwood]

Zooko Wilcox-O'Hearn

Brian Warner

Kevan Carstensen

Frédéric Marti

Jack Lloyd

François Deppierraz

Yu Xue

Marc Tooley

Peter Secor

Shawn Willden

Terrell Russell

——BEGIN PGP SIGNED MESSAGE—— Hash: SHA256

CHAPTER 18

Donations

Donations to the Tahoe-LAFS project are welcome, and can be made to the following Bitcoin address:

1PxiFvW1jyLM5T6Q1YhpkCLxUh3Fw8saF3

The funds currently available to the project are visible through the blockchain explorer:

https://blockchain.info/address/1PxiFvW1jyLM5T6Q1YhpkCLxUh3Fw8saF3

Governance

The Tahoe-LAFS Software Foundation manages these funds. Our intention is to use them for operational expenses (website hosting, test infrastructure, EC2 instance rental, and SSL certificates). Future uses might include developer summit expenses, bug bounties, contract services (e.g. graphic design for the web site, professional security review of codebases, development of features outside the core competencies of the main developers), and student sponsorships.

The Foundation currently consists of secorp (Peter Secor), warner (Brian Warner), and zooko (Zooko Wilcox).

Transparent Accounting

Our current plan is to leave all funds in the main *1Pxi* key until they are spent. For each declared budget item, we will allocate a new public key, and transfer funds to that specific key before distributing them to the ultimate recipient. All expenditures can thus be tracked on the blockchain.

Some day, we might choose to move the funds into a more sophisticated type of key (e.g. a 2-of-3 multisig address). If/when that happens, we will publish the new donation address, and transfer all funds to it. We will continue the plan of keeping all funds in the (new) primary donation address until they are spent.

Expenditure Addresses

This lists the public key used for each declared budget item. The individual payments will be recorded in a separate file (see *docs/expenses.rst*), which is not signed. All transactions from the main *1Pxi* key should be to some key on this list.

- Initial testing (warner) 1387fFG7Jg1iwCzfmQ34FwUva7RnC6ZHYG one-time 0.01 BTC deposit+withdrawal
- tahoe-lafs.org DNS registration (paid by warner) 1552pt6wpudVCRcJaU14T7tAk8grpUza4D ~$15/yr for DNS
- tahoe-lafs.org SSL certificates (paid by warner) $0-$50/yr, ending 2015 (when we switched to LetsEncrypt) 1EkT8yLvQhnjnLpJ6bNFCfAHJrM9yDjsqa
- website/dev-server hosting (on Linode, paid by secorp) ~$20-$25/mo, 2007-present 1MSWNt1R1fohYxgaMV7gJSWbYjkGbXzKWu (<= may-2016) 1NHgVsq1nAU9x1Bb8Rs5K3SNtzEH95C5kU (>= jun-2016)
- 2016 Tahoe Summit expenses: venue rental, team dinners (paid by warner) ~$1020 1DskmM8uCvmvTKjPbeDgfmVsGifZCmxouG

Historical Donation Addresses

The Tahoe project has had a couple of different donation addresses over the years, managed by different people. All of these funds have been (or will be) transferred to the current primary donation address (*1Pxi*).

- 13GrdS9aLXoEbcptBLQi7ffTsVsPR7ubWE (21-Aug-2010 - 23-Aug-2010) Managed by secorp, total receipts: 17 BTC
- 19jzBxijUeLvcMVpUYXcRr5kGG3ThWgx4P (23-Aug-2010 - 29-Jan-2013) Managed by secorp, total receipts: 358.520276 BTC
- 14WTbezUqWSD3gLhmXjHD66jVg7CwqkgMc (24-May-2013 - 21-Mar-2016) Managed by luckyredhot, total receipts: 3.97784278 BTC stored in 19jXek4HRL54JrEwNPyEzitPnkew8XPkd8
- 1PxiFvW1jyLM5T6Q1YhpkCLxUh3Fw8saF3 (21-Mar-2016 - present) Managed by warner, backups with others

Validation

This document is signed by the Tahoe-LAFS Release-Signing Key (GPG keyid 2048R/68666A7A, fingerprint E34E 62D0 6D0E 69CF CA41 79FF BDE0 D31D 6866 6A7A). It is also committed to the Tahoe source tree (https://github.com/tahoe-lafs/tahoe-lafs.git) as *docs/donations.rst*. Both actions require access to secrets held closely by Tahoe developers.

signed: Brian Warner, 10-Nov-2016

——BEGIN PGP SIGNATURE—— Version: GnuPG v2

iQEcBAEBCAAGBQJYJVQBAAoJEL3g0x1oZmp6/8gIAJ5N2jLRQgpfIQTbVvhpnnOc
MGV/kTN5yiN88laX91BPiX8HoAYrBcrzVH/If/2qGkQOGt8RW/91XJC++85JopzN
Gw8uoyhxFB2b4+Yw2WLBSFKx58CyNoq47ZSwLUpard7P/qNrN+Szb26X0jDLo+7V
XL6kXphL82b775xbFxW6afSNSjFJzdbozU+imTqxCu+WqIRW8iD2vjQxx6T6SSrA
q0aLSlZpmD2mHGG3C3K2yYnX7C0BoGR9j4HAN9HbXtTKdVxq98YZOh11jmU1RVV/ nT-
ncD4E1CMrv/QqmktjXw/2shiGihYX+3ZqTO5BAZerORn0MkxPOIvESSVUhHVw= =Oj0C ——END PGP SIGNATURE——

CHAPTER 19

Expenses paid by donated BTC

docs/donations.rst describes the "Transparent Accounting" that we use for BTC that has been donated to the Tahoe project. That document lists the budget items for which we intend to spend these funds, and a Bitcoin public key for each one. It is signed by the Tahoe-LAFS Release Signing Key, and gets re-signed each time a new budget item is added.

For every expense that get paid, the BTC will first be moved from the primary donation key into the budget-item -specific subkey, then moved from that subkey to whatever vendor or individual is being paid.

This document tracks the actual payments made to each vendor. This file changes more frequently than *donations.rst*, hence it is *not* signed. However this file should never reference a budget item or public key which is not present in *donations.rst*. And every payment in this file should correspond to a transaction visible on the Bitcoin block chain explorer:

https://blockchain.info/address/1PxiFvW1jyLM5T6Q1YhpkCLxUh3Fw8saF3

Budget Items

Initial Testing

This was a small transfer to obtain proof-of-spendability for the new wallet.

- Budget: trivial
- Recipient: warner
- Address: 1387fFG7Jg1iwCzfmQ34FwUva7RnC6ZHYG

Expenses/Transactions:

- 17-Mar-2016: deposit+withdrawal of 0.01 BTC
- bcad5f46ebf9fd5d2d7a6a9bed81acf6382cd7216ceddbb5b5f5d968718ec139 (in)
- 13c7f4abf9d6e7f2223c20fefdc47837779bebf3bd95dbb1f225f0d2a2d62c44 (out 1/2)
- 7ca0828ea11fa2f93ab6b8afd55ebdca1415c82c567119d9bd943adbefccce84 (out 2/2)

DNS Registration

Yearly registration of the *tahoe-lafs.org* domain name.

- Budget: ~$15/yr
- Recipient: warner
- Address: 1552pt6wpudVCRcJaU14T7tAk8grpUza4D

Expenses/Transactions:

- 21-Aug-2012: 1 year, GANDI: $12.50
- 20-Aug-2013: 4 years, GANDI: $64.20
- 4ee7fbcb07f758d51187b6856eaf9999f14a7f3d816fe3afb7393f110814ae5e 0.11754609 BTC (@$653.41) = $76.70, plus 0.000113 tx-fee

TLS certificates

Yearly payment for TLS certificates from various vendors. We plan to move to Lets Encrypt, so 2015 should be last time we pay for a cert.

- Budget: $0-$50/yr
- Recipient: warner
- Address: 1EkT8yLvQhnjnLpJ6bNFCfAHJrM9yDjsqa

Expenses/Transactions:

- 29-Oct-2012: RapidSSL: $49
- 02-Nov-2013: GlobalSign, free for open source projects: $0
- 14-Nov-2014: GANDI: $16
- 28-Oct-2015: GANDI: $16
- e8d1b78fab163baa45de0ec592f8d7547329343181e35c2cdb30e427a442337e 0.12400489 BTC (@$653.20) = $81, plus 0.000113 tx-fee

Web/Developer Server Hosting

This pays for the rental of a VPS (currently from Linode) for tahoe-lafs.org, running the project website, Trac, buildbot, and other development tools.

- Budget: $20-$25/month, 2007-present
- Recipient: secorp
- Addresses: 1MSWNt1R1fohYxgaMV7gJSWbYjkGbXzKWu (<= may-2016) 1NHgVsq1nAU9x1Bb8Rs5K3SNtzEH95C5kU (>= jun-2016)

Expenses/Transactions:

- Invoice 311312, 12 Feb 2010: $339.83
- Invoice 607395, 05 Jan 2011: $347.39
- Invoice 1183568, 01 Feb 2012: $323.46
- Invoice 1973091, 01 Feb 2013: $323.46

- Invoice 2899489, 01 Feb 2014: $324.00
- Invoice 3387159, 05 July 2014: $6.54 (add backups)
- Multiple invoices monthly 01 Aug 2014 - 01 May 2016: $7.50*22 = $165.00
- Invoice 4083422, 01 Feb 2015: $324.00
- Invoice 5650991, 01 Feb 2016: $324.00
- – Total through 01 May 2016: $2477.68
- 5861efda59f9ae10952389cf52f968bb469019c77a3642e276a9e35131c36600 3.78838567 BTC (@$654.02) = $2477.68, plus 0.000113 tx-fee
-
- June 2016 - Oct 2016 $27.45/mo, total $137.25
- 8975b03002166b20782b0f023116b3a391ac5176de1a27e851891bee29c11957 0.19269107 BTC (@$712.28) = $137.25, plus 0.000113 tx-fee
- (Oops, I forgot the process, and sent the BTC directly secorp's key. I should have stuck with the 1MSWN key as the intermediary. Next time I'll go back to doing it that way.)

Tahoe Summit

This pays for office space rental and team dinners for each day of the developer summit.

- Recipient: warner
- Address: 1DskmM8uCvmvTKjPbeDgfmVsGifZCmxouG
- 2016 Summit (Nov 8-9, San Francisco)
- Rental of the Mechanics Institute Library "Board Room": $300/day*2
- Team Dinner (Cha Cha Cha): $164.49
- Team Dinner (Rasoi): $255.34
- – total: $1019.83
- dcd468fb2792b018e9ebc238e9b93992ad5a8fce48a8ff71db5d79ccbbe30a92 0.01403961 (@$712.28) = $10, plus 0.000113 tx-fee
- acdfc299c35eed3bb27f7463ad8cdfcdcd4dcfd5184f290f87530c2be999de3e 1.41401086 (@$714.16) = $1009.83, plus 0.000133 tx-fee

CHAPTER 20

Things To Be Careful About As We Venture Boldly Forth

See also *Known Issues*.

Timing Attacks

Asymmetric-key cryptography operations are particularly sensitive to side-channel attacks. Unless the library is carefully hardened against timing attacks, it is dangerous to allow an attacker to measure how long signature and pubkey-derivation operations take. With enough samples, the attacker can deduce the private signing key from these measurements. (Note that verification operations are only sensitive if the verifying key is secret, which is not the case for anything in Tahoe).

We currently use private-key operations in mutable-file writes, and anticipate using them in signed-introducer announcements and accounting setup.

Mutable-file writes can reveal timing information to the attacker because the signature operation takes place in the middle of a read-modify-write cycle. Modifying a directory requires downloading the old contents of the mutable file, modifying the contents, signing the new contents, then uploading the new contents. By observing the elapsed time between the receipt of the last packet for the download, and the emission of the first packet of the upload, the attacker will learn information about how long the signature took. The attacker might ensure that they run one of the servers, and delay responding to the download request so that their packet is the last one needed by the client. They might also manage to be the first server to which a new upload packet is sent. This attack gives the adversary timing information about one signature operation per mutable-file write. Note that the UCWE automatic-retry response (used by default in directory modification code) can cause multiple mutable-file read-modify-write cycles per user-triggered operation, giving the adversary a slightly higher multiplier.

The signed-introducer announcement involves a signature made as the client node is booting, before the first connection is established to the Introducer. This might reveal timing information if any information is revealed about the client's exact boot time: the signature operation starts a fixed number of cycles after node startup, and the first packet to the Introducer is sent a fixed number of cycles after the signature is made. An adversary who can compare the node boot time against the transmission time of the first packet will learn information about the signature operation, one measurement per reboot. We currently do not provide boot-time information in Introducer messages or other client-to-server data.

In general, we are not worried about these leakages, because timing-channel attacks typically require thousands or millions of measurements to detect the (presumably) small timing variations exposed by our asymmetric crypto operations, which would require thousands of mutable-file writes or thousands of reboots to be of use to the adversary. However, future authors should take care to not make changes that could provide additional information to attackers.

CHAPTER 21

Avoiding Write Collisions in Tahoe

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.

If mutable parts of a file store are accessed via sshfs, only a single sshfs mount should be used. There may be data loss if mutable files or directories are accessed via two sshfs mounts, or written both via sshfs and from other clients.

CHAPTER 22

Magic Folder Set-up Howto

This document

This is preliminary documentation of how to set up Magic Folder using a test grid on a single Linux or Windows machine, with two clients and one server. It is aimed at a fairly technical audience.

For an introduction to Magic Folder and how to configure it more generally, see *Tahoe-LAFS Magic Folder Frontend*.

It it possible to adapt these instructions 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 (if the Tahoe-LAFS encoding parameters are changed).

Setting up a local test grid

Linux

Run these commands:

```
mkdir ../grid
bin/tahoe create-introducer ../grid/introducer
bin/tahoe start ../grid/introducer
export FURL=`cat ../grid/introducer/private/introducer.furl`
bin/tahoe create-node --introducer="$FURL" ../grid/server
bin/tahoe create-client --introducer="$FURL" ../grid/alice
bin/tahoe create-client --introducer="$FURL" ../grid/bob
```

Windows

Run:

```
mkdir ..\grid
bin\tahoe create-introducer ..\grid\introducer
bin\tahoe start ..\grid\introducer
```

Leave the introducer running in that Command Prompt, and in a separate Command Prompt (with the same current directory), run:

```
set /p FURL=<..\grid\introducer\private\introducer.furl
bin\tahoe create-node --introducer=%FURL% ..\grid\server
bin\tahoe create-client --introducer=%FURL% ..\grid\alice
bin\tahoe create-client --introducer=%FURL% ..\grid\bob
```

Both Linux and Windows

(Replace / with \ for Windows paths.)

Edit `../grid/alice/tahoe.cfg`, and make the following changes to the `[node]` and `[client]` sections:

```
[node]
nickname = alice
web.port = tcp:3457:interface=127.0.0.1

[client]
shares.needed = 1
shares.happy = 1
shares.total = 1
```

Edit `../grid/bob/tahoe.cfg`, and make the following change to the `[node]` section, and the same change as above to the `[client]` section:

```
[node]
nickname = bob
web.port = tcp:3458:interface=127.0.0.1
```

Note that when running nodes on a single machine, unique port numbers must be used for each node (and they must not clash with ports used by other server software). Here we have used the default of 3456 for the server, 3457 for alice, and 3458 for bob.

Now start all of the nodes (the introducer should still be running from above):

```
bin/tahoe start ../grid/server
bin/tahoe start ../grid/alice
bin/tahoe start ../grid/bob
```

On Windows, a separate Command Prompt is needed to run each node.

Open a web browser on http://127.0.0.1:3457/ and verify that alice is connected to the introducer and one storage server. Then do the same for http://127.0.0.1:3568/ to verify that bob is connected. Leave all of the nodes running for the next stage.

Setting up Magic Folder

Linux

Run:

```
mkdir -p ../local/alice ../local/bob
bin/tahoe -d ../grid/alice magic-folder create magic: alice ../local/alice
bin/tahoe -d ../grid/alice magic-folder invite magic: bob >invitecode
export INVITECODE=`cat invitecode`
bin/tahoe -d ../grid/bob magic-folder join "$INVITECODE" ../local/bob

bin/tahoe restart ../grid/alice
bin/tahoe restart ../grid/bob
```

Windows

Run:

```
mkdir ..\local\alice ..\local\bob
bin\tahoe -d ..\grid\alice magic-folder create magic: alice ..\local\alice
bin\tahoe -d ..\grid\alice magic-folder invite magic: bob >invitecode
set /p INVITECODE=<invitecode
bin\tahoe -d ..\grid\bob magic-folder join %INVITECODE% ..\local\bob
```

Then close the Command Prompt windows that are running the alice and bob nodes, and open two new ones in which to run:

```
bin\tahoe start ..\grid\alice
bin\tahoe start ..\grid\bob
```

Testing

You can now experiment with creating files and directories in `../local/alice` and `/local/bob`; any changes 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 With Magic-Folder*.

As mentioned earlier, 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 (if the Tahoe-LAFS encoding parameters are changed).

Configuration

There will be a `[magic_folder]` section in your `tahoe.cfg` file after setting up Magic Folder.

There is an option you can add to this called `poll_interval=` to control how often (in seconds) the Downloader will check for new things to download.

CHAPTER 23

The Tahoe BackupDB

Overview

To speed up backup operations, Tahoe maintains a small database known as the "backupdb". This is used to avoid re-uploading files which have already been uploaded recently.

This database lives in `~/.tahoe/private/backupdb.sqlite`, and is a SQLite single-file database. It is used by the "`tahoe backup`" command. In the future, it may optionally be used by other commands such as "`tahoe cp`".

The purpose of this database is twofold: to manage the file-to-cap translation (the "upload" step) and the directory-to-cap translation (the "mkdir-immutable" step).

The overall goal of optimizing backup is to reduce the work required when the source disk has not changed (much) since the last backup. In the ideal case, running "`tahoe backup`" twice in a row, with no intervening changes to the disk, will not require any network traffic. Minimal changes to the source disk should result in minimal traffic.

This database is optional. If it is deleted, the worst effect is that a subsequent backup operation may use more effort (network bandwidth, CPU cycles, and disk IO) than it would have without the backupdb.

The database uses sqlite3, which is included as part of the standard Python library with Python 2.5 and later. For Python 2.4, Tahoe will try to install the "pysqlite" package at build-time, but this will succeed only if sqlite3 with development headers is already installed. On Debian and Debian derivatives you can install the "python-pysqlite2" package (which, despite the name, actually provides sqlite3 rather than sqlite2). On old distributions such as Debian etch (4.0 "oldstable") or Ubuntu Edgy (6.10) the "python-pysqlite2" package won't work, but the "sqlite3-dev" package will.

Schema

The database contains the following tables:

```
CREATE TABLE version
(
 version integer  # contains one row, set to 1
);

CREATE TABLE local_files
(
 path  varchar(1024),  PRIMARY KEY -- index, this is an absolute UTF-8-encoded local␣
↪filename
 size  integer,         -- os.stat(fn)[stat.ST_SIZE]
 mtime number,          -- os.stat(fn)[stat.ST_MTIME]
 ctime number,          -- os.stat(fn)[stat.ST_CTIME]
 fileid integer
);

CREATE TABLE caps
(
 fileid integer PRIMARY KEY AUTOINCREMENT,
 filecap varchar(256) UNIQUE    -- URI:CHK:...
);

CREATE TABLE last_upload
(
 fileid INTEGER PRIMARY KEY,
 last_uploaded TIMESTAMP,
 last_checked TIMESTAMP
);

CREATE TABLE directories
(
 dirhash varchar(256) PRIMARY KEY,
 dircap varchar(256),
 last_uploaded TIMESTAMP,
 last_checked TIMESTAMP
);
```

Upload Operation

The upload process starts with a pathname (like `~/.emacs`) and wants to end up with a file-cap (like `URI:CHK:...`).

The first step is to convert the path to an absolute form (`/home/warner/.emacs`) and do a lookup in the local_files table. If the path is not present in this table, the file must be uploaded. The upload process is:

1. record the file's size, ctime (which is the directory-entry change time or file creation time depending on OS) and modification time
2. upload the file into the grid, obtaining an immutable file read-cap
3. add an entry to the 'caps' table, with the read-cap, to get a fileid
4. add an entry to the 'last_upload' table, with the current time

5. add an entry to the 'local_files' table, with the fileid, the path, and the local file's size/ctime/mtime

If the path *is* present in 'local_files', the easy-to-compute identifying information is compared: file size and ctime/mtime. If these differ, the file must be uploaded. The row is removed from the local_files table, and the upload process above is followed.

If the path is present but ctime or mtime differs, the file may have changed. If the size differs, then the file has certainly changed. At this point, a future version of the "backup" command might hash the file and look for a match in an as-yet-defined table, in the hopes that the file has simply been moved from somewhere else on the disk. This enhancement requires changes to the Tahoe upload API before it can be significantly more efficient than simply handing the file to Tahoe and relying upon the normal convergence to notice the similarity.

If ctime, mtime, or size is different, the client will upload the file, as above.

If these identifiers are the same, the client will assume that the file is unchanged (unless the `--ignore-timestamps` option is provided, in which case the client always re-uploads the file), and it may be allowed to skip the upload. For safety, however, we require the client periodically perform a filecheck on these probably-already-uploaded files, and re-upload anything that doesn't look healthy. The client looks the fileid up in the 'last_checked' table, to see how long it has been since the file was last checked.

A "random early check" algorithm should be used, in which a check is performed with a probability that increases with the age of the previous results. E.g. files that were last checked within a month are not checked, files that were checked 5 weeks ago are re-checked with 25% probability, 6 weeks with 50%, more than 8 weeks are always checked. This reduces the "thundering herd" of filechecks-on-everything that would otherwise result when a backup operation is run one month after the original backup. If a filecheck reveals the file is not healthy, it is re-uploaded.

If the filecheck shows the file is healthy, or if the filecheck was skipped, the client gets to skip the upload, and uses the previous filecap (from the 'caps' table) to add to the parent directory.

If a new file is uploaded, a new entry is put in the 'caps' and 'last_upload' table, and an entry is made in the 'local_files' table to reflect the mapping from local disk pathname to uploaded filecap. If an old file is re-uploaded, the 'last_upload' entry is updated with the new timestamps. If an old file is checked and found healthy, the 'last_upload' entry is updated.

Relying upon timestamps is a compromise between efficiency and safety: a file which is modified without changing the timestamp or size will be treated as unmodified, and the "`tahoe backup`" command will not copy the new contents into the grid. The `--no-timestamps` option can be used to disable this optimization, forcing every byte of the file to be hashed and encoded.

Directory Operations

Once the contents of a directory are known (a filecap for each file, and a dircap for each directory), the backup process must find or create a tahoe directory node with the same contents. The contents are hashed, and the hash is queried in the 'directories' table. If found, the last-checked timestamp is used to perform the same random-early-check algorithm described for files above, but no new upload is performed. Since "`tahoe backup`" creates immutable directories, it is perfectly safe to re-use a directory from a previous backup.

If not found, the web-API "mkdir-immutable" operation is used to create a new directory, and an entry is stored in the table.

The comparison operation ignores timestamps and metadata, and pays attention solely to the file names and contents.

By using a directory-contents hash, the "`tahoe backup`" command is able to re-use directories from other places in the backed up data, or from old backups. This means that renaming a directory and moving a subdirectory to a new parent both count as "minor changes" and will result in minimal Tahoe operations and subsequent network traffic (new directories will be created for the modified directory and all of its ancestors). It also means that you can perform a backup ("#1"), delete a file or directory, perform a backup ("#2"), restore it, and then the next backup ("#3") will re-use the directories from backup #1.

The best case is a null backup, in which nothing has changed. This will result in minimal network bandwidth: one directory read and two modifies. The `Archives/` directory must be read to locate the latest backup, and must be modified to add a new snapshot, and the `Latest/` directory will be updated to point to that same snapshot.

CHAPTER 24

Using Tahoe-LAFS with an anonymizing network: Tor, I2P

Overview

Tor is an anonymizing network used to help hide the identity of internet clients and servers. Please see the Tor Project's website for more information: https://www.torproject.org/

I2P is a decentralized anonymizing network that focuses on end-to-end anonymity between clients and servers. Please see the I2P website for more information: https://geti2p.net/

Use cases

There are three potential use-cases for Tahoe-LAFS on the client side:

1. User wishes to always use an anonymizing network (Tor, I2P) to protect their anonymity when connecting to Tahoe-LAFS storage grids (whether or not the storage servers are anonymous).
2. User does not care to protect their anonymity but they wish to connect to Tahoe-LAFS storage servers which are accessible only via Tor Hidden Services or I2P.
   - Tor is only used if a server connection hint uses `tor:`. These hints generally have a `.onion` address.
   - I2P is only used if a server connection hint uses `i2p:`. These hints generally have a `.i2p` address.
3. User does not care to protect their anonymity or to connect to anonymous storage servers. This document is not useful to you... so stop reading.

For Tahoe-LAFS storage servers there are three use-cases:

1. The operator wishes to protect their anonymity by making their Tahoe server accessible only over I2P, via Tor Hidden Services, or both.
2. The operator does not *require* anonymity for the storage server, but they want it to be available over both publicly routed TCP/IP and through an anonymizing network (I2P, Tor Hidden Services). One possible reason to do this is because being reachable through an anonymizing network is a convenient way to bypass NAT or firewall that prevents publicly routed TCP/IP connections to your server (for clients capable of connecting to such servers). Another is that making your storage server reachable through an anonymizing network can provide better protection for your clients who themselves use that anonymizing network to protect their anonymity.
3. Storage server operator does not care to protect their own anonymity nor to help the clients protect theirs. Stop reading this document and run your Tahoe-LAFS storage server using publicly routed TCP/IP.

   See this Tor Project page for more information about Tor Hidden Services: https://www.torproject.org/docs/hidden-services.html.en

   See this I2P Project page for more information about I2P: https://geti2p.net/en/about/intro

Software Dependencies

Tor

Clients who wish to connect to Tor-based servers must install the following.

- Tor (tor) must be installed. See here: https://www.torproject.org/docs/installguide.html.en . On Debian/Ubuntu, use `apt-get install tor`. You can also install and run the Tor Browser Bundle.
- Tahoe-LAFS must be installed with the `[tor]` "extra" enabled. This will install `txtorcon`

```
pip install tahoe-lafs[tor]
```

Manually-configured Tor-based servers must install Tor, but do not need `txtorcon` or the `[tor]` extra. Automatic configuration, when implemented, will need these, just like clients.

I2P

Clients who wish to connect to I2P-based servers must install the following. As with Tor, manually-configured I2P-based servers need the I2P daemon, but no special Tahoe-side supporting libraries.

- I2P must be installed. See here: https://geti2p.net/en/download
- The SAM API must be enabled.
  - Start I2P.

  - Visit http://127.0.0.1:7657/configclients in your browser.
  - Under "Client Configuration", check the "Run at Startup?" box for "SAM application bridge".
  - Click "Save Client Configuration".
  - Click the "Start" control for "SAM application bridge", or restart I2P.
- Tahoe-LAFS must be installed with the `[i2p]` extra enabled, to get `txi2p`

```
pip install tahoe-lafs[i2p]
```

Both Tor and I2P

Clients who wish to connect to both Tor- and I2P-based servers must install all of the above. In particular, Tahoe-LAFS must be installed with both extras enabled:

```
pip install tahoe-lafs[tor,i2p]
```

Connection configuration

See *Connection Management* for a description of the `[tor]` and `[i2p]` sections of `tahoe.cfg`. These control how the Tahoe client will connect to a Tor/I2P daemon, and thus make connections to Tor/I2P -based servers.

The `[tor]` and `[i2p]` sections only need to be modified to use unusual configurations, or to enable automatic server setup.

The default configuration will attempt to contact a local Tor/I2P daemon listening on the usual ports (9050/9150 for Tor, 7656 for I2P). As long as there is a daemon running on the local host, and the necessary support libraries were installed, clients will be able to use Tor-based servers without any special configuration.

However note that this default configuration does not improve the client's anonymity: normal TCP connections will still be made to any server that offers a regular address (it fulfills the second client use case above, not the third). To protect their anonymity, users must configure the `[connections]` section as follows:

```
[connections]
tcp = tor
```

With this in place, the client will use Tor (instead of an IP-address -revealing direct connection) to reach TCP-based servers.

Anonymity configuration

Tahoe-LAFS provides a configuration "safety flag" for explicitly stating whether or not IP-address privacy is required for a node:

```
[node]
reveal-IP-address = (boolean, optional)
```

When `reveal-IP-address = False`, Tahoe-LAFS will refuse to start if any of the configuration options in `tahoe.cfg` would reveal the node's network location:

- `[connections] tcp = tor` is required: otherwise the client would make direct connections to the Introducer, or any TCP-based servers it learns from the Introducer, revealing its IP address to those servers and a network eavesdropper. With this in place, Tahoe-LAFS will only make outgoing connections through a supported anonymizing network.
- `tub.location` must either be disabled, or contain safe values. This value is advertised to other nodes via the Introducer: it is how a server advertises it's location so clients can connect to it. In private mode, it is an error to include a `tcp:` hint in `tub.location`. Private mode rejects the default value of `tub.location` (when the key is missing entirely), which is `AUTO`, which uses `ifconfig` to guess the node's external IP address, which would reveal it to the server and other clients.

This option is **critical** to preserving the client's anonymity (client use-case 3 from *Use cases*, above). It is also necessary to preserve a server's anonymity (server use-case 3).

This flag can be set (to False) by providing the `--hide-ip` argument to the `create-node`, `create-client`, or `create-introducer` commands.

Note that the default value of `reveal-IP-address` is True, because unfortunately hiding the node's IP address requires additional software to be installed (as described above), and reduces performance.

Client anonymity

To configure a client node for anonymity, `tahoe.cfg` **must** contain the following configuration flags:

```
[node]
reveal-IP-address = False
tub.port = disabled
tub.location = disabled
```

Once the Tahoe-LAFS node has been restarted, it can be used anonymously (client use-case 3).

Server anonymity, manual configuration

To configure a server node to listen on an anonymizing network, we must first configure Tor to run an "Onion Service", and route inbound connections to the local Tahoe port. Then we configure Tahoe to advertise the `.onion` address to clients. We also configure Tahoe to not make direct TCP connections.

- Decide on a local listening port number, named PORT. This can be any unused port from about 1024 up to 65535 (depending upon the host's kernel/network config). We will tell Tahoe to listen on this port, and we'll tell Tor to route inbound connections to it.
- Decide on an external port number, named VIRTPORT. This will be used in the advertised location, and revealed to clients. It can be any number from 1 to 65535. It can be the same as PORT, if you like.
- Decide on a "hidden service directory", usually in `/var/lib/tor/NAME`. We'll be asking Tor to save the onion-service state here, and Tor will write the `.onion` address here after it is generated.

Then, do the following:

- Create the Tahoe server node (with `tahoe create-node`), but do **not** launch it yet.
- Edit the Tor config file (typically in `/etc/tor/torrc`). We need to add a section to define the hidden service. If our PORT is 2000, VIRTPORT is 3000, and we're using `/var/lib/tor/tahoe` as the hidden service directory, the section should look like:

```
HiddenServiceDir /var/lib/tor/tahoe
HiddenServicePort 3000 127.0.0.1:2000
```

- Restart Tor, with `systemctl restart tor`. Wait a few seconds.
- Read the `hostname` file in the hidden service directory (e.g. `/var/lib/tor/tahoe/hostname`). This will be a `.onion` address, like `u33m4y7klhz3b.onion`. Call this ONION.
- Edit `tahoe.cfg` to set `tub.port` to use `tcp:PORT:interface=127.0.0.1`, and `tub.location` to use `tor:ONION.onion:VIRTPORT`. Using the examples above, this would be:

```
[node]
reveal-IP-address = false
tub.port = tcp:2000:interface=127.0.0.1
tub.location = tor:u33m4y7klhz3b.onion:3000
[connections]
tcp = tor
```

- Launch the Tahoe server with `tahoe start $NODEDIR`

The `tub.port` section will cause the Tahoe server to listen on PORT, but bind the listening socket to the loopback interface, which is not reachable from the outside world (but *is* reachable by the local Tor daemon). Then the `tcp = tor` section causes Tahoe to use Tor when connecting to the Introducer, hiding it's IP address. The node will then announce itself to all clients using `tub.location`, so clients will know that they must use Tor to reach this server (and not revealing it's IP address through the announcement). When clients connect to the onion address, their packets will flow through the anonymizing network and eventually land on the local Tor daemon, which will then make a connection to PORT on localhost, which is where Tahoe is listening for connections.

Follow a similar process to build a Tahoe server that listens on I2P. The same process can be used to listen on both Tor and I2P (`tub.location = tor:ONION.onion:VIRTPORT,i2p:ADDR.i2p`). It can also listen on both Tor and plain TCP (use-case 2), with `tub.port = tcp:PORT`, `tub.location = tcp:HOST:PORT,tor:ONION.onion:VIRTPORT`, and `anonymous = false` (and omit the `tcp = tor` setting, as the address is already being broadcast through the location announcement).

Server anonymity, automatic configuration

To configure a server node to listen on an anonymizing network, create the node with the `--listen=tor` option. This requires a Tor configuration that either launches a new Tor daemon, or has access to the Tor control port (and enough authority to create a new onion service). On Debian/Ubuntu systems, do `apt install tor`, add yourself to the control group with `adduser YOURUSERNAME debian-tor`, and then logout and log back in: if the `groups` command includes `debian-tor` in the output, you should have permission to use the unix-domain control port at `/var/run/tor/control`.

This option will set `reveal-IP-address = False` and `[connections] tcp = tor`. It will allocate the necessary ports, instruct Tor to create the onion service (saving the private key somewhere inside NODEDIR/private/), obtain the `.onion` address, and populate `tub.port` and `tub.location` correctly.

Performance and security issues

If you are running a server which does not itself need to be anonymous, should you make it reachable via an anonymizing network or not? Or should you make it reachable *both* via an anonymizing network and as a publicly traceable TCP/IP server?

There are several trade-offs effected by this decision.

NAT/Firewall penetration

Making a server be reachable via Tor or I2P makes it reachable (by Tor/I2P-capable clients) even if there are NATs or firewalls preventing direct TCP/IP connections to the server.

Anonymity

Making a Tahoe-LAFS server accessible *only* via Tor or I2P can be used to guarantee that the Tahoe-LAFS clients use Tor or I2P to connect (specifically, the server should only advertise Tor/I2P addresses in the `tub.location` config key). This prevents misconfigured clients from accidentally de-anonymizing themselves by connecting to your server through the traceable Internet.

Clearly, a server which is available as both a Tor/I2P service *and* a regular TCP address is not itself anonymous: the .onion address and the real IP address of the server are easily linkable.

Also, interaction, through Tor, with a Tor Hidden Service may be more protected from network traffic analysis than interaction, through Tor, with a publicly traceable TCP/IP server.

XXX is there a document maintained by Tor developers which substantiates or refutes this belief? If so we need to link to it. If not, then maybe we should explain more here why we think this?

Linkability

As of 1.12.0, the node uses a single persistent Tub key for outbound connections to the Introducer, and inbound connections to the Storage Server (and Helper). For clients, a new Tub key is created for each storage server we learn about, and these keys are *not* persisted (so they will change each time the client reboots).

Clients traversing directories (from rootcap to subdirectory to filecap) are likely to request the same storage-indices (SIs) in the same order each time. A client connected to multiple servers will ask them all for the same SI at about the same time. And two clients which are sharing files or directories will visit the same SIs (at various times).

As a result, the following things are linkable, even with `reveal-IP-address = false`:

- Storage servers can link recognize multiple connections from the same not-yet-rebooted client. (Note that the upcoming Accounting feature may cause clients to present a persistent client-side public key when connecting, which will be a much stronger linkage).
- Storage servers can probably deduce which client is accessing data, by looking at the SIs being requested. Multiple servers can collude to determine that the same client is talking to all of them, even though the TubIDs are different for each connection.
- Storage servers can deduce when two different clients are sharing data.
- The Introducer could deliver different server information to each subscribed client, to partition clients into distinct sets according to which server connections they eventually make. For client+server nodes, it can also correlate the server announcement with the deduced client identity.

Performance

A client connecting to a publicly traceable Tahoe-LAFS server through Tor incurs substantially higher latency and sometimes worse throughput than the same client connecting to the same server over a normal traceable TCP/IP connection. When the server is on a Tor Hidden Service, it incurs even more latency, and possibly even worse throughput.

Connecting to Tahoe-LAFS servers which are I2P servers incurs higher latency and worse throughput too.

Positive and negative effects on other Tor users

Sending your Tahoe-LAFS traffic over Tor adds cover traffic for other Tor users who are also transmitting bulk data. So that is good for them – increasing their anonymity.

However, it makes the performance of other Tor users' interactive sessions – e.g. ssh sessions – much worse. This is because Tor doesn't currently have any prioritization or quality-of-service features, so someone else's ssh keystrokes may have to wait in line while your bulk file contents get transmitted. The added delay might make other people's interactive sessions unusable.

Both of these effects are doubled if you upload or download files to a Tor Hidden Service, as compared to if you upload or download files over Tor to a publicly traceable TCP/IP server.

Positive and negative effects on other I2P users

Sending your Tahoe-LAFS traffic over I2P adds cover traffic for other I2P users who are also transmitting data. So that is good for them – increasing their anonymity. It will not directly impair the performance of other I2P users' interactive sessions, because the I2P network has several congestion control and quality-of-service features, such as prioritizing smaller packets.

However, if many users are sending Tahoe-LAFS traffic over I2P, and do not have their I2P routers configured to participate in much traffic, then the I2P network as a whole will suffer degradation. Each Tahoe-LAFS router using I2P has their own anonymizing tunnels that their data is sent through. On average, one Tahoe-LAFS node requires 12 other I2P routers to participate in their tunnels.

It is therefore important that your I2P router is sharing bandwidth with other routers, so that you can give back as you use I2P. This will never impair the performance of your Tahoe-LAFS node, because your I2P router will always prioritize your own traffic.

CHAPTER 25

Node Keys in Tahoe-LAFS

"Node Keys" are cryptographic signing/verifying keypairs used to identify Tahoe-LAFS nodes (client-only and client+server). The private signing key is stored in NODEDIR/private/node.privkey , and is used to sign the announcements that are distributed to all nodes by the Introducer. The public verifying key is used to identify the sending node from those other systems: it is displayed as a "Node ID" that looks like "v0-abc234xyz567..", which ends with a long base32-encoded string.

These node keys were introduced in the 1.10 release (April 2013), as part of ticket #466. In previous releases, announcements were unsigned, and nodes were identified by their Foolscap "Tub ID" (a somewhat shorter base32 string, with no "v0-" prefix).

Why Announcements Are Signed

All nodes (both client-only and client+server) publish announcements to the Introducer, which then relays them to all other nodes. These announcements contain information about the publishing node's nickname, how to reach the node, what services it offers, and what version of code it is running.

The new private node key is used to sign these announcements, preventing the Introducer from modifying their contents en-route. This will enable future versions of Tahoe-LAFS to use other forms of introduction (gossip, multiple introducers) without weakening the security model.

The Node ID is useful as a handle with which to talk about a node. For example, when clients eventually gain the ability to control which storage servers they are willing to use (#467), the configuration file might simply include a list of Node IDs for the approved servers.

TubIDs are currently also suitable for this job, but they depend upon having a Foolscap connection to the server. Since our goal is to move away from Foolscap towards a simpler (faster and more portable) protocol, we want to reduce our dependence upon TubIDs. Node IDs and Ed25519 signatures can be used for non-Foolscap non-SSL based protocols.

How The Node ID Is Computed

The long-form Node ID is the Ed25519 public verifying key, 256 bits (32 bytes) long, base32-encoded, with a "v0-" prefix appended, and the trailing "=" padding removed, like so:

v0-rlj3jnxqv4ee5rtpyngvzbhmhuikjfenjve7j5mzmfcxytwmyf6q

The Node ID is displayed in this long form on the node's front Welcome page, and on the Introducer's status page. In most other places (share-placement lists, file health displays), the "short form" is used instead. This is simply the first 8 characters of the base32 portion, frequently enclosed in square brackets, like this:

[rlj3jnxq]

In contrast, old-style TubIDs are usually displayed with just 6 base32 characters.

Version Compatibility, Fallbacks For Old Versions

Since Tahoe-LAFS 1.9 does not know about signed announcements, 1.10 includes backwards-compatibility code to allow old and new versions to interoperate. There are three relevant participants: the node publishing an announcement, the Introducer which relays them, and the node receiving the (possibly signed) announcement.

When a 1.10 node connects to an old Introducer (version 1.9 or earlier), it sends downgraded non-signed announcements. It likewise accepts non-signed announcements from the Introducer. The non-signed announcements use TubIDs to identify the sending node. The new 1.10 Introducer, when it connects to an old node, downgrades any signed announcements to non-signed ones before delivery.

As a result, the only way to receive signed announcements is for all three systems to be running the new 1.10 code. In a grid with a mixture of old and new nodes, if the Introducer is old, then all nodes will see unsigned TubIDs. If the Introducer is new, then nodes will see signed Node IDs whenever possible.

Share Placement

Tahoe-LAFS uses a "permuted ring" algorithm to decide where to place shares for any given file. For each potential server, it uses that server's "permutation seed" to compute a pseudo-random but deterministic location on a ring, then walks the ring in clockwise order, asking each server in turn to hold a share until all are placed. When downloading a file, the servers are accessed in the same order. This minimizes the number of queries that must be done to download a file, and tolerates "churn" (nodes being added and removed from the grid) fairly well.

This property depends upon server nodes having a stable permutation seed. If a server's permutation seed were to change, it would effectively wind up at a randomly selected place on the permuted ring. Downloads would still complete, but clients would spend more time asking other servers before querying the correct one.

In the old 1.9 code, the permutation-seed was always equal to the TubID. In 1.10, servers include their permutation-seed as part of their announcement. To improve stability for existing grids, if an old server (one with existing shares) is upgraded to run the 1.10 codebase, it will use its old TubID as its permutation-seed. When a new empty server runs the 1.10 code, it will use its Node ID instead. In both cases, once the node has picked a permutation-seed, it will continue using that value forever.

To be specific, when a node wakes up running the 1.10 code, it will look for a recorded NODEDIR/permutation-seed file, and use its contents if present. If that file does not exist, it creates it (with the TubID if it has any shares, otherwise with the Node ID), and uses the contents as the permutation-seed.

There is one unfortunate consequence of this pattern. If new 1.10 server is created in a grid that has an old client, or has a new client but an old Introducer, then that client will see downgraded non-signed announcements, and thus will first upload shares with the TubID-based permutation-seed. Later, when the client and/or Introducer is upgraded, the client

will start seeing signed announcements with the NodeID-based permutation-seed, and will then look for shares in the wrong place. This will hurt performance in a large grid, but should not affect reliability. This effect shouldn't even be noticeable in grids for which the number of servers is close to the "N" shares.total number (e.g. where num-servers < 3*N). And the as-yet-unimplemented "share rebalancing" feature should repair the misplacement.

If you wish to avoid this effect, try to upgrade both Introducers and clients at about the same time. (Upgrading servers does not matter: they will continue to use the old permutation-seed).

CHAPTER 26

Performance costs for some common operations

1. *Publishing an A-byte immutable file*
2. *Publishing an A-byte mutable file*
3. *Downloading B bytes of an A-byte immutable file*
4. *Downloading B bytes of an A-byte mutable file*
5. *Modifying B bytes of an A-byte mutable file*
6. *Inserting/Removing B bytes in an A-byte mutable file*
7. *Adding an entry to an A-entry directory*
8. *Listing an A entry directory*
9. *Checking an A-byte file*
10. *Verifying an A-byte file (immutable)*
11. *Repairing an A-byte file (mutable or immutable)*

`K` indicates the number of shares required to reconstruct the file (default: 3)

`N` indicates the total number of shares produced (default: 10)

`S` indicates the segment size (default: 128 KiB)

`A` indicates the number of bytes in a file

`B` indicates the number of bytes of a file that are being read or written

`G` indicates the number of storage servers on your grid

Most of these cost estimates may have a further constant multiplier: when a formula says `N/K*S`, the cost may actually be `2*N/K*S` or `3*N/K*S`. Also note that all references to mutable files are for SDMF-formatted files; this document has not yet been updated to describe the MDMF format.

Publishing an A-byte immutable file

when the file is already uploaded

If the file is already uploaded with the exact same contents, same erasure coding parameters (K, N), and same added convergence secret, then it reads the whole file from disk one time while hashing it to compute the storage index, then contacts about N servers to ask each one to store a share. All of the servers reply that they already have a copy of that share, and the upload is done.

disk: A

cpu: ~A

network: ~N

memory footprint: S

when the file is not already uploaded

If the file is not already uploaded with the exact same contents, same erasure coding parameters (K, N), and same added convergence secret, then it reads the whole file from disk one time while hashing it to compute the storage index, then contacts about N servers to ask each one to store a share. Then it uploads each share to a storage server.

disk: 2*A

cpu: 2*~A

network: N/K*A

memory footprint: N/K*S

Publishing an A-byte mutable file

cpu: ~A + a large constant for RSA keypair generation

network: A

memory footprint: N/K*A

notes: Tahoe-LAFS generates a new RSA keypair for each mutable file that it publishes to a grid. This takes up to 1 or 2 seconds on a typical desktop PC.

Part of the process of encrypting, encoding, and uploading a mutable file to a Tahoe-LAFS grid requires that the entire file be in memory at once. For larger files, this may cause Tahoe-LAFS to have an unacceptably large memory footprint (at least when uploading a mutable file).

Downloading B bytes of an A-byte immutable file

cpu: ~B

network: B

notes: When Tahoe-LAFS 1.8.0 or later is asked to read an arbitrary range of an immutable file, only the S-byte segments that overlap the requested range will be downloaded.

(Earlier versions would download from the beginning of the file up until the end of the requested range, and then continue to download the rest of the file even after the request was satisfied.)

Downloading B bytes of an A-byte mutable file

cpu: ~A

network: A

memory footprint: A

notes: As currently implemented, mutable files must be downloaded in their entirety before any part of them can be read. We are exploring fixes for this; see ticket #393 for more information.

Modifying B bytes of an A-byte mutable file

cpu: ~A

network: A

memory footprint: N/K*A

notes: If you upload a changed version of a mutable file that you earlier put onto your grid with, say, 'tahoe put –mutable', Tahoe-LAFS will replace the old file with the new file on the grid, rather than attempting to modify only those portions of the file that have changed. Modifying a file in this manner is essentially uploading the file over again, except that it re-uses the existing RSA keypair instead of generating a new one.

Inserting/Removing B bytes in an A-byte mutable file

cpu: ~A

network: A

memory footprint: N/K*A

notes: Modifying any part of a mutable file in Tahoe-LAFS requires that the entire file be downloaded, modified, held in memory while it is encrypted and encoded, and then re-uploaded. A future version of the mutable file layout ("LDMF") may provide efficient inserts and deletes. Note that this sort of modification is mostly used internally for directories, and isn't something that the WUI, CLI, or other interfaces will do – instead, they will simply overwrite the file to be modified, as described in "Modifying B bytes of an A-byte mutable file".

Adding an entry to an A-entry directory

cpu: ~A

network: ~A

memory footprint: N/K*~A

notes: In Tahoe-LAFS, directories are implemented as specialized mutable files. So adding an entry to a directory is essentially adding B (actually, 300-330) bytes somewhere in an existing mutable file.

Listing an A entry directory

cpu: ~A

network: ~A

memory footprint: N/K*~A

notes: Listing a directory requires that the mutable file storing the directory be downloaded from the grid. So listing an A entry directory requires downloading a (roughly) 330 * A byte mutable file, since each directory entry is about 300-330 bytes in size.

Checking an A-byte file

cpu: ~G

network: ~G

memory footprint: negligible

notes: To check a file, Tahoe-LAFS queries all the servers that it knows about. Note that neither of these values directly depend on the size of the file. This is relatively inexpensive, compared to the verify and repair operations.

Verifying an A-byte file (immutable)

cpu: ~N/K*A

network: N/K*A

memory footprint: N/K*S

notes: To verify a file, Tahoe-LAFS downloads all of the ciphertext shares that were originally uploaded to the grid and integrity checks them. This is (for grids with good redundancy) more expensive than downloading an A-byte file, since only a fraction of these shares would be necessary to recover the file.

Verifying an A-byte file (mutable)

cpu: ~N/K*A

network: N/K*A

memory footprint: N/K*A

notes: To verify a file, Tahoe-LAFS downloads all of the ciphertext shares that were originally uploaded to the grid and integrity checks them. This is (for grids with good redundancy) more expensive than downloading an A-byte file, since only a fraction of these shares would be necessary to recover the file.

Repairing an A-byte file (mutable or immutable)

cpu: variable, between ~A and ~N/K*A

network: variable; between A and N/K*A

memory footprint (immutable): (1+N/K)*S (SDMF mutable): (1+N/K)*A

notes: To repair a file, Tahoe-LAFS downloads the file, and generates/uploads missing shares in the same way as when it initially uploads the file. So, depending on how many shares are missing, this can cost as little as a download or as much as a download followed by a full upload.

Since SDMF files have only one segment, which must be processed in its entirety, repair requires a full-file download followed by a full-file upload.

CHAPTER 27

Tahoe Logging

Overview

Tahoe uses the Foolscap logging mechanism (known as the "flog" subsystem) to record information about what is happening inside the Tahoe node. This is primarily for use by programmers and grid operators who want to find out what went wrong.

The Foolscap logging system is documented at https://github.com/warner/foolscap/blob/latest-release/doc/logging.rst.

The Foolscap distribution includes a utility named "`flogtool`" that is used to get access to many Foolscap logging features. `flogtool` should get installed into the same virtualenv as the `tahoe` command.

Realtime Logging

When you are working on Tahoe code, and want to see what the node is doing, the easiest tool to use is "`flogtool tail`". This connects to the Tahoe node and subscribes to hear about all log events. These events are then displayed to stdout, and optionally saved to a file.

"flogtool tail" connects to the "logport", for which the FURL is stored in BASEDIR/private/logport.furl . The following command will connect to this port and start emitting log information:

```
flogtool tail BASEDIR/private/logport.furl
```

The --save-to FILENAME option will save all received events to a file, where then can be examined later with "flogtool dump" or "flogtool web-viewer". The --catch-up option will ask the node to dump all stored events before subscribing to new ones (without --catch-up, you will only hear about events that occur after the tool has connected and subscribed).

Incidents

Foolscap keeps a short list of recent events in memory. When something goes wrong, it writes all the history it has (and everything that gets logged in the next few seconds) into a file called an "incident". These files go into BASEDIR/logs/incidents/ , in a file named "incident-TIMESTAMP-UNIQUE.flog.bz2". The default definition of "something goes wrong" is the generation of a log event at the log.WEIRD level or higher, but other criteria could be implemented.

The typical "incident report" we've seen in a large Tahoe grid is about 40kB compressed, representing about 1800 recent events.

These "flogfiles" have a similar format to the files saved by "flogtool tail --save-to". They are simply lists of log events, with a small header to indicate which event triggered the incident.

The "flogtool dump FLOGFILE" command will take one of these .flog.bz2 files and print their contents to stdout, one line per event. The raw event dictionaries can be dumped by using "flogtool dump --verbose FLOGFILE".

The "flogtool web-viewer" command can be used to examine the flogfile in a web browser. It runs a small HTTP server and emits the URL on stdout. This view provides more structure than the output of "flogtool dump": the parent/child relationships of log events is displayed in a nested format. "flogtool web-viewer" is still fairly immature.

Working with flogfiles

The "flogtool filter" command can be used to take a large flogfile (perhaps one created by the log-gatherer, see below) and copy a subset of events into a second file. This smaller flogfile may be easier to work with than the original. The arguments to "flogtool filter" specify filtering criteria: a predicate that each event must match to be copied into the target file. --before and --after are used to exclude events outside a given window of time. --above will retain events above a certain severity level. --from retains events send by a specific tubid. --strip-facility removes events that were emitted with a given facility (like foolscap.negotiation or tahoe.upload).

Gatherers

In a deployed Tahoe grid, it is useful to get log information automatically transferred to a central log-gatherer host. This offloads the (admittedly modest) storage requirements to a different host and provides access to logfiles from multiple nodes (web-API, storage, or helper) in a single place.

There are two kinds of gatherers: "log gatherer" and "stats gatherer". Each produces a FURL which needs to be placed in the NODEDIR/tahoe.cfg file of each node that is to publish to the gatherer, under the keys "log_gatherer.furl"

and "stats_gatherer.furl" respectively. When the Tahoe node starts, it will connect to the configured gatherers and offer its logport: the gatherer will then use the logport to subscribe to hear about events.

The gatherer will write to files in its working directory, which can then be examined with tools like "`flogtool dump`" as described above.

Incident Gatherer

The "incident gatherer" only collects Incidents: records of the log events that occurred just before and slightly after some high-level "trigger event" was recorded. Each incident is classified into a "category": a short string that summarizes what sort of problem took place. These classification functions are written after examining a new/unknown incident. The idea is to recognize when the same problem is happening multiple times.

A collection of classification functions that are useful for Tahoe nodes are provided in `misc/incident-gatherer/support_classifiers.py` . There is roughly one category for each `log.WEIRD`-or-higher level event in the Tahoe source code.

The incident gatherer is created with the "`flogtool create-incident-gatherer WORKDIR`" command, and started with "`tahoe start`". The generated "`gatherer.tac`" file should be modified to add classifier functions.

The incident gatherer writes incident names (which are simply the relative pathname of the `incident-\*.flog.bz2` file) into `classified/CATEGORY`. For example, the `classified/mutable-retrieve-uncoordinated-write-error` file contains a list of all incidents which were triggered by an uncoordinated write that was detected during mutable file retrieval (caused when somebody changed the contents of the mutable file in between the node's mapupdate step and the retrieve step). The `classified/unknown` file contains a list of all incidents that did not match any of the classification functions.

At startup, the incident gatherer will automatically reclassify any incident report which is not mentioned in any of the `classified/\*` files. So the usual workflow is to examine the incidents in `classified/unknown`, add a new classification function, delete `classified/unknown`, then bound the gatherer with "`tahoe restart WORKDIR`". The incidents which can be classified with the new functions will be added to their own `classified/FOO` lists, and the remaining ones will be put in `classified/unknown`, where the process can be repeated until all events are classifiable.

The incident gatherer is still fairly immature: future versions will have a web interface and an RSS feed, so operations personnel can track problems in the storage grid.

In our experience, each incident takes about two seconds to transfer from the node that generated it to the gatherer. The gatherer will automatically catch up to any incidents which occurred while it is offline.

Log Gatherer

The "Log Gatherer" subscribes to hear about every single event published by the connected nodes, regardless of severity. This server writes these log events into a large flogfile that is rotated (closed, compressed, and replaced with a new one) on a periodic basis. Each flogfile is named according to the range of time it represents, with names like "`from-2008-08-26-132256--to-2008-08-26-162256.flog.bz2`". The flogfiles contain events from many different sources, making it easier to correlate things that happened on multiple machines (such as comparing a client node making a request with the storage servers that respond to that request).

Create the Log Gatherer with the "`flogtool create-gatherer WORKDIR`" command, and start it with "`tahoe start`". Then copy the contents of the `log_gatherer.furl` file it creates into the `BASEDIR/tahoe.cfg` file (under the key `log_gatherer.furl` of the section `[node]`) of all nodes that should be sending it log events. (See *Configuring a Tahoe-LAFS node*)

The "`flogtool filter`" command, described above, is useful to cut down the potentially large flogfiles into a more focussed form.

Busy nodes, particularly web-API nodes which are performing recursive deep-size/deep-stats/deep-check operations, can produce a lot of log events. To avoid overwhelming the node (and using an unbounded amount of memory for the outbound TCP queue), publishing nodes will start dropping log events when the outbound queue grows too large. When this occurs, there will be gaps (non-sequential event numbers) in the log-gatherer's flogfiles.

Adding log messages

When adding new code, the Tahoe developer should add a reasonable number of new log events. For details, please see the Foolscap logging documentation, but a few notes are worth stating here:

- use a facility prefix of "`tahoe.`", like "`tahoe.mutable.publish`"
- assign each severe (`log.WEIRD` or higher) event a unique message identifier, as the `umid=` argument to the `log.msg()` call. The `misc/coding_tools/make_umid` script may be useful for this purpose. This will make it easier to write a classification function for these messages.
- use the `parent=` argument whenever the event is causally/temporally clustered with its parent. For example, a download process that involves three sequential hash fetches could announce the send and receipt of those hash-fetch messages with a `parent=` argument that ties them to the overall download process. However, each new web-API download request should be unparented.
- use the `format=` argument in preference to the `message=` argument. E.g. use `log.msg(format="got %(n)d shares, need %(k)d", n=n, k=k)` instead of `log.msg("got %d shares, need %d" % (n,k))`. This will allow later tools to analyze the event without needing to scrape/reconstruct the structured data out of the formatted string.
- Pass extra information as extra keyword arguments, even if they aren't included in the `format=` string. This information will be displayed in the "`flogtool dump --verbose`" output, as well as being available to other tools. The `umid=` argument should be passed this way.
- use `log.err` for the catch-all `addErrback` that gets attached to the end of any given Deferred chain. When used in conjunction with `LOGTOTWISTED=1`, `log.err()` will tell Twisted about the error-nature of the log message, causing Trial to flunk the test (with an "ERROR" indication that prints a copy of the Failure, including a traceback). Don't use `log.err` for events that are `BAD` but handled (like hash failures: since these are often deliberately provoked by test code, they should not cause test failures): use `log.msg(level=BAD)` for those instead.

Log Messages During Unit Tests

If a test is failing and you aren't sure why, start by enabling `FLOGTOTWISTED=1` like this:

```
make test FLOGTOTWISTED=1
```

With `FLOGTOTWISTED=1`, sufficiently-important log events will be written into `_trial_temp/test.log`, which may give you more ideas about why the test is failing.

By default, `_trial_temp/test.log` will not receive messages below the `level=OPERATIONAL` threshold. You can change the threshold via the `FLOGLEVEL` variable, e.g.:

```
make test FLOGLEVEL=10 FLOGTOTWISTED=1
```

(The level numbers are listed in src/allmydata/util/log.py.)

To look at the detailed foolscap logging messages, run the tests like this:

```
make test FLOGFILE=flog.out.bz2 FLOGLEVEL=1 FLOGTOTWISTED=1
```

The first environment variable will cause foolscap log events to be written to `./flog.out.bz2` (instead of merely being recorded in the circular buffers for the use of remote subscribers or incident reports). The second will cause all log events to be written out, not just the higher-severity ones. The third will cause twisted log events (like the markers that indicate when each unit test is starting and stopping) to be copied into the flogfile, making it easier to correlate log events with unit tests.

Enabling this form of logging appears to roughly double the runtime of the unit tests. The `flog.out.bz2` file is approximately 2MB.

You can then use "`flogtool dump`" or "`flogtool web-viewer`" on the resulting `flog.out` file.

("`flogtool tail`" and the log-gatherer are not useful during unit tests, since there is no single Tub to which all the log messages are published).

It is possible for setting these environment variables to cause spurious test failures in tests with race condition bugs. All known instances of this have been fixed as of Tahoe-LAFS v1.7.1.

CHAPTER 28

Tahoe Statistics

Overview

Each Tahoe node collects and publishes statistics about its operations as it runs. These include counters of how many files have been uploaded and downloaded, CPU usage information, performance numbers like latency of storage server operations, and available disk space.

The easiest way to see the stats for any given node is use the web interface. From the main "Welcome Page", follow the "Operational Statistics" link inside the small "This Client" box. If the welcome page lives at http://localhost:3456/, then the statistics page will live at http://localhost:3456/statistics . This presents a summary of the stats block, along with a copy of the raw counters. To obtain just the raw counters (in JSON format), use /statistics?t=json instead.

Statistics Categories

The stats dictionary contains two keys: 'counters' and 'stats'. 'counters' are strictly counters: they are reset to zero when the node is started, and grow upwards. 'stats' are non-incrementing values, used to measure the current state of various systems. Some stats are actually booleans, expressed as '1' for true and '0' for false (internal restrictions require all stats values to be numbers).

Under both the 'counters' and 'stats' dictionaries, each individual stat has a key with a dot-separated name, breaking them up into groups like 'cpu_monitor' and 'storage_server'.

The currently available stats (as of release 1.6.0 or so) are described here:

counters.storage_server.*

this group counts inbound storage-server operations. They are not provided by client-only nodes which have been configured to not run a storage server (with [storage]enabled=false in tahoe.cfg)

allocate, write, close, abort these are for immutable file uploads. 'allocate' is incremented when a client asks if it can upload a share to the server. 'write' is incremented for each chunk of data written. 'close' is incremented when the share is finished. 'abort' is incremented if the client abandons the upload.

get, read these are for immutable file downloads. 'get' is incremented when a client asks if the server has a specific share. 'read' is incremented for each chunk of data read.

readv, writev these are for immutable file creation, publish, and retrieve. 'readv' is incremented each time a client reads part of a mutable share. 'writev' is incremented each time a client sends a modification request.

add-lease, renew, cancel these are for share lease modifications. 'add-lease' is incremented when an 'add-lease' operation is performed (which either adds a new lease or renews an existing lease). 'renew' is for the 'renew-lease' operation (which can only be used to renew an existing one). 'cancel' is used for the 'cancel-lease' operation.

bytes_freed this counts how many bytes were freed when a 'cancel-lease' operation removed the last lease from a share and the share was thus deleted.

bytes_added this counts how many bytes were consumed by immutable share uploads. It is incremented at the same time as the 'close' counter.

stats.storage_server.*

allocated this counts how many bytes are currently 'allocated', which tracks the space that will eventually be consumed by immutable share upload operations. The stat is increased as soon as the upload begins (at the same time the 'allocated' counter is incremented), and goes back to zero when the 'close' or 'abort' message is received (at which point the 'disk_used' stat should incremented by the same amount).

disk_total, disk_used, disk_free_for_root, disk_free_for_nonroot, disk_avail, reserved_space these all reflect disk-space usage policies and status. 'disk_total' is the total size of disk where the storage server's BASEDIR/storage/shares directory lives, as reported by /bin/df or equivalent. 'disk_used', 'disk_free_for_root', and 'disk_free_for_nonroot' show related information. 'reserved_space' reports the reservation configured by the tahoe.cfg [storage]reserved_space value. 'disk_avail' reports the remaining disk space available for the Tahoe server after subtracting reserved_space from disk_avail. All values are in bytes.

accepting_immutable_shares this is '1' if the storage server is currently accepting uploads of immutable shares. It may be '0' if a server is disabled by configuration, or if the disk is full (i.e. disk_avail is less than reserved_space).

total_bucket_count this counts the number of 'buckets' (i.e. unique storage-index values) currently managed by the storage server. It indicates roughly how many files are managed by the server.

latencies.*.* these stats keep track of local disk latencies for storage-server operations. A number of percentile values are tracked for many operations. For example, 'storage_server.latencies.readv.50_0_percentile' records the median response time for a 'readv' request. All values are in seconds. These are recorded by the storage server, starting from the time the request arrives (post-deserialization) and ending when the response begins serialization. As such, they are mostly useful for measuring disk speeds. The operations tracked are the same as the counters.storage_server.* counter values (allocate, write, close, get, read, add-lease, renew, cancel, readv, writev). The percentile values tracked are: mean, 01_0_percentile, 10_0_percentile, 50_0_percentile, 90_0_percentile, 95_0_percentile, 99_0_percentile, 99_9_percentile. (the last value, 99.9 percentile, means that 999 out of the last 1000 operations were faster than the given number, and is the same threshold used by Amazon's internal SLA, according to the Dynamo paper). Percentiles

are only reported in the case of a sufficient number of observations for unambiguous interpretation. For example, the 99.9th percentile is (at the level of thousandths precision) 9 thousandths greater than the 99th percentile for sample sizes greater than or equal to 1000, thus the 99.9th percentile is only reported for samples of 1000 or more observations.

counters.uploader.files_uploaded

counters.uploader.bytes_uploaded

counters.downloader.files_downloaded

counters.downloader.bytes_downloaded

These count client activity: a Tahoe client will increment these when it uploads or downloads an immutable file. 'files_uploaded' is incremented by one for each operation, while 'bytes_uploaded' is incremented by the size of the file.

counters.mutable.files_published

counters.mutable.bytes_published

counters.mutable.files_retrieved

counters.mutable.bytes_retrieved

These count client activity for mutable files. 'published' is the act of changing an existing mutable file (or creating a brand-new mutable file). 'retrieved' is the act of reading its current contents.

counters.chk_upload_helper.*

These count activity of the "Helper", which receives ciphertext from clients and performs erasure-coding and share upload for files that are not already in the grid. The code which implements these counters is in src/allmydata/immutable/offloaded.py .

upload_requests incremented each time a client asks to upload a file upload_already_present: incremented when the file is already in the grid

upload_need_upload incremented when the file is not already in the grid

resumes incremented when the helper already has partial ciphertext for the requested upload, indicating that the client is resuming an earlier upload

fetched_bytes this counts how many bytes of ciphertext have been fetched from uploading clients

encoded_bytes this counts how many bytes of ciphertext have been encoded and turned into successfully-uploaded shares. If no uploads have failed or been abandoned, encoded_bytes should eventually equal fetched_bytes.

stats.chk_upload_helper.*

These also track Helper activity:

active_uploads how many files are currently being uploaded. 0 when idle.

incoming_count how many cache files are present in the incoming/ directory, which holds ciphertext files that are still being fetched from the client

incoming_size total size of cache files in the incoming/ directory

incoming_size_old total size of 'old' cache files (more than 48 hours)

encoding_count how many cache files are present in the encoding/ directory, which holds ciphertext files that are being encoded and uploaded

encoding_size total size of cache files in the encoding/ directory

encoding_size_old total size of 'old' cache files (more than 48 hours)

stats.node.uptime how many seconds since the node process was started

stats.cpu_monitor.*

1min_avg, 5min_avg, 15min_avg estimate of what percentage of system CPU time was consumed by the node process, over the given time interval. Expressed as a float, 0.0 for 0%, 1.0 for 100%

total estimate of total number of CPU seconds consumed by node since the process was started. Ticket #472 indicates that .total may sometimes be negative due to wraparound of the kernel's counter.

stats.load_monitor.*

When enabled, the "load monitor" continually schedules a one-second callback, and measures how late the response is. This estimates system load (if the system is idle, the response should be on time). This is only enabled if a stats-gatherer is configured.

avg_load average "load" value (seconds late) over the last minute

max_load maximum "load" value over the last minute

Running a Tahoe Stats-Gatherer Service

The "stats-gatherer" is a simple daemon that periodically collects stats from several tahoe nodes. It could be useful, e.g., in a production environment, where you want to monitor dozens of storage servers from a central management host. It merely gatherers statistics from many nodes into a single place: it does not do any actual analysis.

The stats gatherer listens on a network port using the same Foolscap connection library that Tahoe clients use to connect to storage servers. Tahoe nodes can be configured to connect to the stats gatherer and publish their stats on a periodic basis. (In fact, what happens is that nodes connect to the gatherer and offer it a second FURL which points back to the node's "stats port", which the gatherer then uses to pull stats on a periodic basis. The initial connection is flipped to allow the nodes to live behind NAT boxes, as long as the stats-gatherer has a reachable IP address.)

The stats-gatherer is created in the same fashion as regular tahoe client nodes and introducer nodes. Choose a base directory for the gatherer to live in (but do not create the directory). Choose the hostname that should be advertised in the gatherer's FURL. Then run:

```
tahoe create-stats-gatherer --hostname=HOSTNAME $BASEDIR
```

and start it with "tahoe start $BASEDIR". Once running, the gatherer will write a FURL into $BASEDIR/stats_gatherer.furl .

To configure a Tahoe client/server node to contact the stats gatherer, copy this FURL into the node's tahoe.cfg file, in a section named "[client]", under a key named "stats_gatherer.furl", like so:

```
[client]
stats_gatherer.furl = pb://qbo4ktl667zmtiuou6lwbjryli2brv6t@HOSTNAME:PORTNUM/
↪wxycb4kaexzskubjnauxeoptympyf45y
```

or simply copy the stats_gatherer.furl file into the node's base directory (next to the tahoe.cfg file): it will be interpreted in the same way.

When the gatherer is created, it will allocate a random unused TCP port, so it should not conflict with anything else that you have running on that host at that time. To explicitly control which port it uses, run the creation command with `--location=` and `--port=` instead of `--hostname=`. If you use a hostname of `example.org` and a port number of `1234`, then run:

```
tahoe create-stats-gatherer --location=tcp:example.org:1234 --port=tcp:1234
```

`--location=` is a Foolscap FURL hints string (so it can be a comma-separated list of connection hints), and `--port=` is a Twisted "server endpoint specification string", as described in *Configuring a Tahoe-LAFS node*.

Once running, the stats gatherer will create a standard JSON file in `$BASEDIR/stats.json`. Once a minute, the gatherer will pull stats information from every connected node and write them into the file. The file will contain a dictionary, in which node identifiers (known as "tubid" strings) are the keys, and the values are a dict with 'timestamp', 'nickname', and 'stats' keys. d[tubid][stats] will contain the stats dictionary as made available at http://localhost:3456/statistics?t=json . The file will only contain the most recent update from each node.

Other tools can be built to examine these stats and render them into something useful. For example, a tool could sum the "storage_server.disk_avail' values from all servers to compute a total-disk-available number for the entire grid (however, the "disk watcher" daemon, in misc/operations_helpers/spacetime/, is better suited for this specific task).

Using Munin To Graph Stats Values

The misc/operations_helpers/munin/ directory contains various plugins to graph stats for Tahoe nodes. They are intended for use with the Munin system-management tool, which typically polls target systems every 5 minutes and produces a web page with graphs of various things over multiple time scales (last hour, last month, last year).

Most of the plugins are designed to pull stats from a single Tahoe node, and are configured with the e.g. http://localhost:3456/statistics?t=json URL. The "tahoe_stats" plugin is designed to read from the JSON file created by the stats-gatherer. Some plugins are to be used with the disk watcher, and a few (like tahoe_nodememory) are designed to watch the node processes directly (and must therefore run on the same host as the target node).

Please see the docstrings at the beginning of each plugin for details, and the "tahoe-conf" file for notes about configuration and installing these plugins into a Munin environment.

CHAPTER 29

How To Build Tahoe-LAFS On A Desert Island

(or an airplane, or anywhere else without internet connectivity)

Here's the story: you leave for the airport in an hour, you know you want to do some Tahoe hacking on the flight. What can you grab right now that will let you install the necessary dependencies later, when you are offline?

Pip can help, with a technique described in the pip documentation https://pip.pypa.io/en/stable/user_guide/#installing-from-local-packages .

First, do two setup steps:

- `mkdir ~/.pip/wheels`
- edit `~/.pip/pip.conf` to set `[global] find-links = ~/.pip/wheels`

(the filename may vary on non-unix platforms: check the pip documentation for details)

This instructs all `pip install` commands to look in your local directory for compiled wheels, in addition to asking PyPI and the normal wheel cache.

Before you get shipwrecked (or leave the internet for a while), do this from your tahoe source tree (or any python source tree that you want to hack on):

- `pip wheel -w ~/.pip/wheels .`

That command will require network and time: it will download and compile whatever is necessary right away. Schedule your shipwreck for *after* it completes.

Specifically, it will get wheels for everything that the current project (".", i.e. tahoe) needs, and write them to the `~/.pip/wheels` directory. It will query PyPI to learn the current version of every dependency, then acquire wheels from the first source that has one:

- copy from our `~/.pip/wheels` directory
- copy from the local wheel cache (see below for where this lives)
- download a wheel from PyPI
- build a wheel from a tarball (cached or downloaded)

Later, on the plane, do this:

- `virtualenv --no-download ve`
- `. ve/bin/activate`
- `pip install --no-index --editable .`

That tells virtualenv/pip to not try to contact PyPI, and your `pip.conf` "find-links" tells them to use the wheels in `~/.pip/wheels/` instead.

How This Works

The pip wheel cache

Modern versions of pip and setuptools will, by default, cache both their HTTP downloads and their generated wheels. When pip is asked to install a package, it will first check with PyPI. If the PyPI index says it needs to download a newer version, but it can find a copy of the tarball/zipball/wheel in the HTTP cache, it will not actually download anything. Then it tries to build a wheel: if it already has one in the wheel cache (downloaded or built earlier), it will not actually build anything.

If it cannot contact PyPI, it will fail. The `--no-index` above is to tell it to skip the PyPI step, but that leaves it with no source of packages. The `find-links` setting is what provides an alternate source of packages.

The HTTP and wheel caches are not single flat directories: they use a hierarchy of subdirectories, named after a hash of the URL or name of the object being stored (this is to avoid filesystem limitations on the size of a directory). As a result, the wheel cache is not suitable for use as a `find-links` target (but see below).

There is a command named `pip wheel` which only creates wheels (and stores them in `--wheel-dir=`, which defaults to the current directory). This command does not populate the wheel cache: it reads from (and writes to) the HTTP cache, and reads from the wheel cache, but will only save the generated wheels into the directory you specify with `--wheel-dir=`.

Where Does The Cache Live?

Pip's cache location depends upon the platform. On linux, it defaults to ~/.cache/pip/ (both http/ and wheels/). On OS-X (homebrew), it uses ~/Library/Caches/pip/ . On Windows, try ~AppDataLocalpipcache .

The location can be overridden by `pip.conf`. Look for the "wheel-dir", "cache-dir", and "find-links" options.

How Can I Tell If It's Using The Cache?

When "pip install" has to download a source tarball (and build a wheel), it will say things like:

```
Collecting zfec
 Downloading zfec-1.4.24.tar.gz (175kB)
Building wheels for collected packages: zfec
 Running setup.py bdist_wheel for zfec ... done
 Stored in directory: $CACHEDIR
Successfully built zfec
Installing collected packages: zfec
Successfully installed zfec-1.4.24
```

When "pip install" can use a cached downloaded tarball, but does not have a cached wheel, it will say:

```
Collecting zfec
 Using cached zfec-1.4.24.tar.gz
Building wheels for collected packages: zfec
 Running setup.py bdist_wheel for zfec ... done
 Stored in directory: $CACHEDIR
Successfully built zfec
Installing collected packages: zfec
Successfully installed zfec-1.4.24
```

When "pip install" can use a cached wheel, it will just say:

```
Collecting zfec
Installed collected packages: zfec
Successfully installed zfec-1.4.24
```

Many packages publish pre-built wheels next to their source tarballs. This is common for non-platform-specific (pure-python) packages. It is also common for them to provide pre-compiled windows and OS-X wheel, so users do not have to have a compiler installed (pre-compiled Linux wheels are not common, because there are too many platform variations). When "pip install" can use a downloaded wheel like this, it will say:

```
Collecting six
 Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: six
Successfully installed six-1.10.0
```

Note that older versions of pip do not always use wheels, or the cache. Pip 8.0.0 or newer should be ok. The version of setuptools may also be significant.

CHAPTER 30

Debian and Ubuntu Support

Overview

Tahoe-LAFS is provided as a `.deb` package in current Debian (>= wheezy) and Ubuntu (>= lucid) releases. Before official packages were added, the Tahoe source tree provided support for building unofficial packages for a variety of popular Debian/Ubuntu versions. The project also ran buildbots to create `.debs` of current trunk for ease of testing.

As of version 1.9, the source tree no longer provides these tools. To construct a `.deb` from current trunk, your best bet is to apply the current Debian diff from the latest upstream package and invoke the `debian/rules` as usual. Debian's standard `apt-get` tool can be used to fetch the current source package (including the Debian-specific diff): run "`apt-get source tahoe-lafs`". That will fetch three files: the `.dsc` control file, the main Tahoe tarball, and the Debian-specific `.debian.tar.gz` file. Just unpack the `.debian.tar.gz` file inside your Tahoe source tree, modify the version number in `debian/changelog`, then run "`fakeroot ./debian/rules binary`", and a new `.deb` will be placed in the parent directory.

Dependency Packages

Tahoe depends upon a number of additional libraries. When building Tahoe from source, any dependencies that are not already present in the environment will be downloaded (via `pip` and `easy_install`) and installed in the virtualenv.

The `.deb` packages, of course, rely solely upon other `.deb` packages. For reference, here is a list of the debian package names that provide Tahoe's dependencies as of the 1.9 release:

- python
- python-zfec
- python-pycryptopp

- python-foolscap
- python-openssl (needed by foolscap)
- python-twisted
- python-nevow
- python-mock
- python-simplejson
- python-setuptools
- python-support (for Debian-specific install-time tools)

When building your own Debian packages, a convenient way to get all these dependencies installed is to first install the official "tahoe-lafs" package, then uninstall it, leaving the dependencies behind. You may also find it useful to run "`apt-get build-dep tahoe-lafs`" to make sure all the usual build-essential tools are installed.

CHAPTER 31

Building Tahoe-LAFS on Windows

You'll need `python`, `pip`, and `virtualenv`. But you won't need a compiler.

Preliminaries

1: Install Python-2.7.11 . Use the "Windows x86-64 MSI installer" at https://www.python.org/downloads/release/python-2711/

2: That should install `pip`, but if it doesn't, look at https://pip.pypa.io/en/stable/installing/ for installation instructions.

3: Install `virtualenv` with https://virtualenv.pypa.io/en/latest/installation.html

Installation

1: Start a CLI shell (e.g. PowerShell)

2: Create a new virtualenv. Everything specific to Tahoe will go into this. You can use whatever name you like for the virtualenv, but example uses "venv":

```
PS C:\Users\me> virtualenv venv
New python executable in C:\Users\me\venv\Scripts\python.exe
Installing setuptools, pip, wheel...done.
>
```

3: Use the virtualenv's `pip` to install the latest release of Tahoe-LAFS into this virtualenv:

```
PS C:\Users\me> venv\Scripts\pip install --find-links=https://tahoe-lafs.org/deps/␣
↪tahoe-lafs
Collecting tahoe-lafs
...
Installing collected packages: ...
Successfully installed ...
>
```

4: Verify that Tahoe was installed correctly by running `tahoe --version`, using the `tahoe` from the virtualenv's Scripts directory:

```
PS C:\Users\me> venv\Scripts\tahoe --version
tahoe-lafs: 1.11
foolscap: ...
```

Running Tahoe-LAFS

The rest of the documentation assumes you can run the `tahoe` executable just as you did in step 4 above. If you want to type just `tahoe` instead of `venv\Scripts\tahoe`, you can either "activate" the virtualenv (by running `venv\Scripts\activate`, or you can add the Scripts directory to your `%PATH%` environment variable.

Now use the docs in *How To Run Tahoe-LAFS* to learn how to configure your first Tahoe node.

Installing A Different Version

The `pip install tahoe-lafs` command above will install the latest release (from PyPI). If instead, you want to install from a git checkout, then run the following command (using pip from the virtualenv, from the root of your git checkout):

```
$ venv\Scripts\pip install --find-links=https://tahoe-lafs.org/deps/ .
```

If you're planning to hack on the source code, you might want to add `--editable` so you won't have to re-install each time you make a change.

Dependencies

Tahoe-LAFS depends upon several packages that use compiled C code, such as zfec, pycryptopp, and others. This code must be built separately for each platform (Windows, OS-X, and different flavors of Linux).

Pre-compiled "wheels" of all Tahoe's dependencies are hosted on the tahoe-lafs.org website in the `deps/` directory. The `--find-links=` argument (used in the examples above) instructs `pip` to look at that URL for dependencies. This should avoid the need for anything to be compiled during the install.

CHAPTER 32

OS-X Packaging

Pre-built Tahoe-LAFS ".pkg" installers for OS-X are generated with each source-code commit. These installers offer an easy way to get Tahoe and all its dependencies installed on your Mac. They do not yet provide a double-clickable application: after installation, you will have a "tahoe" command-line tool, which you can use from a shell (a Terminal window) just as if you'd installed from source.

Installers are available from this directory:

https://tahoe-lafs.org/source/tahoe-lafs/tarballs/OS-X-packages/

Download the latest .pkg file to your computer and double-click on it. This will install to /Applications/tahoe.app, however the app icon there is not how you use Tahoe (launching it will get you a dialog box with a reminder to use Terminal). `/Applications/tahoe.app/bin/tahoe` is the executable. The next shell you start ought to have that directory in your $PATH (thanks to a file in `/etc/paths.d/`), unless your `.profile` overrides it.

Tahoe-LAFS is also easy to install with pip, as described in the README.

CHAPTER 33

Building pyOpenSSL on Windows

This document details the steps to build an pyOpenSSL egg with embedded OpenSSL library, for use by Tahoe-LAFS on Windows.

The instructions were tried on Windows 7 64-bit and Windows XP 32-bit. They should work on other versions of Windows, maybe with minor variations.

Download and install Microsoft Visual C++ compiler for Python 2.7

For reasons detailed in the Python documentation, Python extension modules need to be built using a compiler compatible with the same version of Visual C++ that was used to build Python itself. Until recently, this meant downloading Microsoft Visual Studio 2008 Express Edition and Windows SDK 3.5. The recent release of the Microsoft Visual C++ compiler for Python 2.7 made things a lot simpler.

So, the first step is to download and install the C++ compiler from Microsoft from this link.

Find the location where it installed the `vcvarsall.bat` file; depending on the version of Windows it could be either `"%USERPROFILE%\AppData\Local\Programs\Common\Microsoft\Visual C++ for Python\9.0"` or `"%CommonProgramFiles%\Microsoft\Visual C++ for Python\9.0"`, for example. We'll call this `%VCDIR%` below.

Download and install Perl

Download and install ActiveState Perl:

- go to the ActiveState Perl download page.
- identify the correct link and manually change it from http to https.

Download and install the latest OpenSSL version

- Download the latest OpenSSL from the OpenSSL source download page and untar it. At the time of writing, the latest version was OpenSSL 1.0.1m.
- Set up the build environment. For 64-bit Windows:

```
"%VCDIR%\vcvarsall.bat" amd64
```

  or for 32-bit Windows:

```
"%VCDIR%\vcvarsall.bat" x86
```

- Go to the untar'ed OpenSSL source base directory. For 64-bit Windows, run:

```
mkdir c:\dist
perl Configure VC-WIN64A --prefix=c:\dist\openssl no-asm enable-tlsext
ms\do_win64a.bat
nmake -f ms\ntdll.mak
nmake -f ms\ntdll.mak install
```

  or for 32-bit Windows, run:

```
mkdir c:\dist
perl Configure VC-WIN32 --prefix=c:\dist\openssl no-asm enable-tlsext
ms\do_ms.bat
nmake -f ms\ntdll.mak
nmake -f ms\ntdll.mak install
```

To check that it is working, run `c:\dist\openssl\bin\openssl version`.

Building PyOpenSSL

- Download and untar pyOpenSSL 0.13.1 (see ticket #2221 for why we currently use this version). The MD5 hash of pyOpenSSL-0.13.1.tar.gz is e27a3b76734c39ea03952ca94cc56715.
- Set up the build environment by running `vcvarsall.bat` as for building OpenSSL above.
- Set OpenSSL `LIB`, `INCLUDE` and `PATH`:

```
set LIB=c:\dist\openssl\lib;%LIB%
set INCLUDE=c:\dist\openssl\include;%INCLUDE%
set PATH=c:\dist\openssl\bin;%PATH%
```

- A workaround is needed to ensure that the setuptools `bdist_egg` command is available. Edit pyOpenSSL's `setup.py` around line 13 as follows:

```
< from distutils.core import Extension, setup
---
> from setuptools import setup
> from distutils.core import Extension
```

- Run `python setup.py bdist_egg`

The generated egg will be in the `dist` directory. It is a good idea to check that Tahoe-LAFS is able to use it before uploading the egg to tahoe-lafs.org. This can be done by putting it in the `tahoe-deps` directory of a Tahoe-LAFS checkout or release, then running `python setup.py test`.

CHAPTER 34

Specifications

This section contains various attempts at writing detailed specifications of the data formats used by Tahoe.

Specification Document Outline

While we do not yet have a clear set of specification documents for Tahoe (explaining the file formats, so that others can write interoperable implementations), this document is intended to lay out an outline for what these specs ought to contain. Think of this as the ISO 7-Layer Model for Tahoe.

We currently imagine 4 documents.

1. *#1: Share Format, Encoding Algorithm*
2. *#2: Share Exchange Protocol*
3. *#3: Server Selection Algorithm, filecap format*
4. *#4: Directory Format*

#1: Share Format, Encoding Algorithm

This document will describe the way that files are encrypted and encoded into shares. It will include a specification of the share format, and explain both the encoding and decoding algorithms. It will cover both mutable and immutable files.

The immutable encoding algorithm, as described by this document, will start with a plaintext series of bytes, encoding parameters "k" and "N", and either an encryption key or a mechanism for deterministically deriving the key from the plaintext (the CHK specification). The algorithm will end with a set of N shares, and a set of values that must be included in the filecap to provide confidentiality (the encryption key) and integrity (the UEB hash).

The immutable decoding algorithm will start with the filecap values (key and UEB hash) and "k" shares. It will explain how to validate the shares against the integrity information, how to reverse the erasure-coding, and how to decrypt the resulting ciphertext. It will result in the original plaintext bytes (or some subrange thereof).

The sections on mutable files will contain similar information.

This document is *not* responsible for explaining the filecap format, since full filecaps may need to contain additional information as described in document #3. Likewise it it not responsible for explaining where to put the generated shares or where to find them again later.

It is also not responsible for explaining the access control mechanisms surrounding share upload, download, or modification ("Accounting" is the business of controlling share upload to conserve space, and mutable file shares require some sort of access control to prevent non-writecap holders from destroying shares). We don't yet have a document dedicated to explaining these, but let's call it "Access Control" for now.

#2: Share Exchange Protocol

This document explains the wire-protocol used to upload, download, and modify shares on the various storage servers.

Given the N shares created by the algorithm described in document #1, and a set of servers who are willing to accept those shares, the protocols in this document will be sufficient to get the shares onto the servers. Likewise, given a set of servers who hold at least k shares, these protocols will be enough to retrieve the shares necessary to begin the decoding process described in document #1. The notion of a "storage index" is used to reference a particular share: the storage index is generated by the encoding process described in document #1.

This document does *not* describe how to identify or choose those servers, rather it explains what to do once they have been selected (by the mechanisms in document #3).

This document also explains the protocols that a client uses to ask a server whether or not it is willing to accept an uploaded share, and whether it has a share available for download. These protocols will be used by the mechanisms in document #3 to help decide where the shares should be placed.

Where cryptographic mechanisms are necessary to implement access-control policy, this document will explain those mechanisms.

In the future, Tahoe will be able to use multiple protocols to speak to storage servers. There will be alternative forms of this document, one for each protocol. The first one to be written will describe the Foolscap-based protocol that tahoe currently uses, but we anticipate a subsequent one to describe a more HTTP-based protocol.

#3: Server Selection Algorithm, filecap format

This document has two interrelated purposes. With a deeper understanding of the issues, we may be able to separate these more cleanly in the future.

The first purpose is to explain the server selection algorithm. Given a set of N shares, where should those shares be uploaded? Given some information stored about a previously-uploaded file, how should a downloader locate and recover at least k shares? Given a previously-uploaded mutable file, how should a modifier locate all (or most of) the shares with a reasonable amount of work?

This question implies many things, all of which should be explained in this document:

- the notion of a "grid", nominally a set of servers who could potentially hold shares, which might change over time
- a way to configure which grid should be used
- a way to discover which servers are a part of that grid
- a way to decide which servers are reliable enough to be worth sending shares
- an algorithm to handle servers which refuse shares
- a way for a downloader to locate which servers have shares
- a way to choose which shares should be used for download

The server-selection algorithm has several obviously competing goals:

- minimize the amount of work that must be done during upload
- minimize the total storage resources used
- avoid "hot spots", balance load among multiple servers
- maximize the chance that enough shares will be downloadable later, by uploading lots of shares, and by placing them on reliable servers
- minimize the work that the future downloader must do
- tolerate temporary server failures, permanent server departure, and new server insertions
- minimize the amount of information that must be added to the filecap

The server-selection algorithm is defined in some context: some set of expectations about the servers or grid with which it is expected to operate. Different algorithms are appropriate for different situtations, so there will be multiple alternatives of this document.

The first version of this document will describe the algorithm that the current (1.3.0) release uses, which is heavily weighted towards the two main use case scenarios for which Tahoe has been designed: the small, stable friendnet, and the allmydata.com managed grid. In both cases, we assume that the storage servers are online most of the time, they are uniformly highly reliable, and that the set of servers does not change very rapidly. The server-selection algorithm for this environment uses a permuted server list to achieve load-balancing, uses all servers identically, and derives the permutation key from the storage index to avoid adding a new field to the filecap.

An alternative algorithm could give clients more precise control over share placement, for example by a user who wished to make sure that k+1 shares are located in each datacenter (to allow downloads to take place using only local bandwidth). This algorithm could skip the permuted list and use other mechanisms to accomplish load-balancing (or ignore the issue altogether). It could add additional information to the filecap (like a list of which servers received the shares) in lieu of performing a search at download time, perhaps at the expense of allowing a repairer to move shares to a new server after the initial upload. It might make up for this by storing "location hints" next to each share, to indicate where other shares are likely to be found, and obligating the repairer to update these hints.

The second purpose of this document is to explain the format of the file capability string (or "filecap" for short). There are multiple kinds of capabilties (read-write, read-only, verify-only, repaircap, lease-renewal cap, traverse-only, etc). There are multiple ways to represent the filecap (compressed binary, human-readable, clickable-HTTP-URL, "tahoe:" URL, etc), but they must all contain enough information to reliably retrieve a file (given some context, of course). It must at least contain the confidentiality and integrity information from document #1 (i.e. the encryption key and the UEB hash). It must also contain whatever additional information the upload-time server-selection algorithm generated that will be required by the downloader.

For some server-selection algorithms, the additional information will be minimal. For example, the 1.3.0 release uses the hash of the encryption key as a storage index, and uses the storage index to permute the server list, and uses an Introducer to learn the current list of servers. This allows a "close-enough" list of servers to be compressed into a filecap field that is already required anyways (the encryption key). It also adds k and N to the filecap, to speed up the downloader's search (the downloader knows how many shares it needs, so it can send out multiple queries in parallel).

But other server-selection algorithms might require more information. Each variant of this document will explain how to encode that additional information into the filecap, and how to extract and use that information at download time.

These two purposes are interrelated. A filecap that is interpreted in the context of the allmydata.com commercial grid, which uses tahoe-1.3.0, implies a specific peer-selection algorithm, a specific Introducer, and therefore a fairly-specific set of servers to query for shares. A filecap which is meant to be interpreted on a different sort of grid would need different information.

Some filecap formats can be designed to contain more information (and depend less upon context), such as the way an HTTP URL implies the existence of a single global DNS system. Ideally a tahoe filecap should be able to specify

which "grid" it lives in, with enough information to allow a compatible implementation of Tahoe to locate that grid and retrieve the file (regardless of which server-selection algorithm was used for upload).

This more-universal format might come at the expense of reliability, however. Tahoe-1.3.0 filecaps do not contain hostnames, because the failure of DNS or an individual host might then impact file availability (however the Introducer contains DNS names or IP addresses).

#4: Directory Format

Tahoe directories are a special way of interpreting and managing the contents of a file (either mutable or immutable). These "dirnode" files are basically serialized tables that map child name to filecap/dircap. This document describes the format of these files.

Tahoe-1.3.0 directories are "transitively readonly", which is accomplished by applying an additional layer of encryption to the list of child writecaps. The key for this encryption is derived from the containing file's writecap. This document must explain how to derive this key and apply it to the appropriate portion of the table.

Future versions of the directory format are expected to contain "deep-traversal caps", which allow verification/repair of files without exposing their plaintext to the repair agent. This document wil be responsible for explaining traversal caps too.

Future versions of the directory format will probably contain an index and more advanced data structures (for efficiency and fast lookups), instead of a simple flat list of (childname, childcap). This document will also need to describe metadata formats, including what access-control policies are defined for the metadata.

Tahoe URIs

Each file and directory in a Tahoe-LAFS file store is described by a "URI". There are different kinds of URIs for different kinds of objects, and there are different kinds of URIs to provide different kinds of access to those objects. Each URI is a string representation of a "capability" or "cap", and there are read-caps, write-caps, verify-caps, and others.

Each URI provides both `location` and `identification` properties. `location` means that holding the URI is sufficient to locate the data it represents (this means it contains a storage index or a lookup key, whatever is necessary to find the place or places where the data is being kept). `identification` means that the URI also serves to validate the data: an attacker who wants to trick you into into using the wrong data will be limited in their abilities by the identification properties of the URI.

Some URIs are subsets of others. In particular, if you know a URI which allows you to modify some object, you can produce a weaker read-only URI and give it to someone else, and they will be able to read that object but not modify it. Directories, for example, have a read-cap which is derived from the write-cap: anyone with read/write access to the directory can produce a limited URI that grants read-only access, but not the other way around.

src/allmydata/uri.py is the main place where URIs are processed. It is the authoritative definition point for all the the URI types described herein.

File URIs

The lowest layer of the Tahoe architecture (the "key-value store") is reponsible for mapping URIs to data. This is basically a distributed hash table, in which the URI is the key, and some sequence of bytes is the value.

There are two kinds of entries in this table: immutable and mutable. For immutable entries, the URI represents a fixed chunk of data. The URI itself is derived from the data when it is uploaded into the grid, and can be used to locate and download that data from the grid at some time in the future.

For mutable entries, the URI identifies a "slot" or "container", which can be filled with different pieces of data at different times.

It is important to note that the values referenced by these URIs are just sequences of bytes, and that **no** filenames or other metadata is retained at this layer. The file store layer (which sits above the key-value store layer) is entirely responsible for directories and filenames and the like.

CHK URIs

CHK (Content Hash Keyed) files are immutable sequences of bytes. They are uploaded in a distributed fashion using a "storage index" (for the "location" property), and encrypted using a "read key". A secure hash of the data is computed to help validate the data afterwards (providing the "identification" property). All of these pieces, plus information about the file's size and the number of shares into which it has been distributed, are put into the "CHK" uri. The storage index is derived by hashing the read key (using a tagged SHA-256d hash, then truncated to 128 bits), so it does not need to be physically present in the URI.

The current format for CHK URIs is the concatenation of the following strings:

```
URI:CHK:(key):(hash):(needed-shares):(total-shares):(size)
```

Where (key) is the base32 encoding of the 16-byte AES read key, (hash) is the base32 encoding of the SHA-256 hash of the URI Extension Block, (needed-shares) is an ascii decimal representation of the number of shares required to reconstruct this file, (total-shares) is the same representation of the total number of shares created, and (size) is an ascii decimal representation of the size of the data represented by this URI. All base32 encodings are expressed in lower-case, with the trailing '=' signs removed.

For example, the following is a CHK URI, generated from a previous version of the contents of *architecture.rst*:

```
URI:CHK:ihrbeov7lbvoduupd4qblysj7a:bg5agsdt62jb34hxvxmdsbza6do64f4fg5anxxod2buttbo6udzq:3:10:28733
```

Historical note: The name "CHK" is somewhat inaccurate and continues to be used for historical reasons. "Content Hash Key" means that the encryption key is derived by hashing the contents, which gives the useful property that encoding the same file twice will result in the same URI. However, this is an optional step: by passing a different flag to the appropriate API call, Tahoe will generate a random encryption key instead of hashing the file: this gives the useful property that the URI or storage index does not reveal anything about the file's contents (except filesize), which improves privacy. The URI:CHK: prefix really indicates that an immutable file is in use, without saying anything about how the key was derived.

LIT URIs

LITeral files are also an immutable sequence of bytes, but they are so short that the data is stored inside the URI itself. These are used for files of 55 bytes or shorter, which is the point at which the LIT URI is the same length as a CHK URI would be.

LIT URIs do not require an upload or download phase, as their data is stored directly in the URI.

The format of a LIT URI is simply a fixed prefix concatenated with the base32 encoding of the file's data:

```
URI:LIT:bjuw4y3movsgkidbnrwg26lemf2gcl3xmvrc6kropbuhi3lmbi
```

The LIT URI for an empty file is "URI:LIT:", and the LIT URI for a 5-byte file that contains the string "hello" is "URI:LIT:nbswy3dp".

Mutable File URIs

The other kind of DHT entry is the "mutable slot", in which the URI names a container to which data can be placed and retrieved without changing the identity of the container.

These slots have write-caps (which allow read/write access), read-caps (which only allow read-access), and verify-caps (which allow a file checker/repairer to confirm that the contents exist, but does not let it decrypt the contents).

Mutable slots use public key technology to provide data integrity, and put a hash of the public key in the URI. As a result, the data validation is limited to confirming that the data retrieved matches *some* data that was uploaded in the past, but not _which_ version of that data.

The format of the write-cap for mutable files is:

```
URI:SSK:(writekey):(fingerprint)
```

Where (writekey) is the base32 encoding of the 16-byte AES encryption key that is used to encrypt the RSA private key, and (fingerprint) is the base32 encoded 32-byte SHA-256 hash of the RSA public key. For more details about the way these keys are used, please see *Mutable Files*.

The format for mutable read-caps is:

```
URI:SSK-RO:(readkey):(fingerprint)
```

The read-cap is just like the write-cap except it contains the other AES encryption key: the one used for encrypting the mutable file's contents. This second key is derived by hashing the writekey, which allows the holder of a write-cap to produce a read-cap, but not the other way around. The fingerprint is the same in both caps.

Historical note: the "SSK" prefix is a perhaps-inaccurate reference to "Sub-Space Keys" from the Freenet project, which uses a vaguely similar structure to provide mutable file access.

Directory URIs

The key-value store layer provides a mapping from URI to data. To turn this into a graph of directories and files, the "file store" layer (which sits on top of the key-value store layer) needs to keep track of "directory nodes", or "dirnodes" for short. *Tahoe-LAFS Directory Nodes* describes how these work.

Dirnodes are contained inside mutable files, and are thus simply a particular way to interpret the contents of these files. As a result, a directory write-cap looks a lot like a mutable-file write-cap:

```
URI:DIR2:(writekey):(fingerprint)
```

Likewise directory read-caps (which provide read-only access to the directory) look much like mutable-file read-caps:

```
URI:DIR2-RO:(readkey):(fingerprint)
```

Historical note: the "DIR2" prefix is used because the non-distributed dirnodes in earlier Tahoe releases had already claimed the "DIR" prefix.

Internal Usage of URIs

The classes in source:src/allmydata/uri.py are used to pack and unpack these various kinds of URIs. Three Interfaces are defined (IURI, IFileURI, and IDirnodeURI) which are implemented by these classes, and string-to-URI-class conversion routines have been registered as adapters, so that code which wants to extract e.g. the size of a CHK or LIT uri can do:

```
print IFileURI(uri).get_size()
```

If the URI does not represent a CHK or LIT uri (for example, if it was for a directory instead), the adaptation will fail, raising a TypeError inside the IFileURI() call.

Several utility methods are provided on these objects. The most important is `to_string()`, which returns the string form of the URI. Therefore `IURI(uri).to_string == uri` is true for any valid URI. See the IURI class in source:src/allmydata/interfaces.py for more details.

File Encoding

When the client wishes to upload an immutable file, the first step is to decide upon an encryption key. There are two methods: convergent or random. The goal of the convergent-key method is to make sure that multiple uploads of the same file will result in only one copy on the grid, whereas the random-key method does not provide this "convergence" feature.

The convergent-key method computes the SHA-256d hash of a single-purpose tag, the encoding parameters, a "convergence secret", and the contents of the file. It uses a portion of the resulting hash as the AES encryption key. There are security concerns with using convergence this approach (the "partial-information guessing attack", please see ticket #365 for some references), so Tahoe uses a separate (randomly-generated) "convergence secret" for each node, stored in NODEDIR/private/convergence . The encoding parameters (k, N, and the segment size) are included in the hash to make sure that two different encodings of the same file will get different keys. This method requires an extra IO pass over the file, to compute this key, and encryption cannot be started until the pass is complete. This means that the convergent-key method will require at least two total passes over the file.

The random-key method simply chooses a random encryption key. Convergence is disabled, however this method does not require a separate IO pass, so upload can be done with a single pass. This mode makes it easier to perform streaming upload.

Regardless of which method is used to generate the key, the plaintext file is encrypted (using AES in CTR mode) to produce a ciphertext. This ciphertext is then erasure-coded and uploaded to the servers. Two hashes of the ciphertext are generated as the encryption proceeds: a flat hash of the whole ciphertext, and a Merkle tree. These are used to verify the correctness of the erasure decoding step, and can be used by a "verifier" process to make sure the file is intact without requiring the decryption key.

The encryption key is hashed (with SHA-256d and a single-purpose tag) to produce the "Storage Index". This Storage Index (or SI) is used to identify the shares produced by the method described below. The grid can be thought of as a large table that maps Storage Index to a ciphertext. Since the ciphertext is stored as erasure-coded shares, it can also be thought of as a table that maps SI to shares.

Anybody who knows a Storage Index can retrieve the associated ciphertext: ciphertexts are not secret.

The ciphertext file is then broken up into segments. The last segment is likely to be shorter than the rest. Each segment is erasure-coded into a number of "blocks". This takes place one segment at a time. (In fact, encryption and erasure-coding take place at the same time, once per plaintext segment). Larger segment sizes result in less overhead overall, but increase both the memory footprint and the "alacrity" (the number of bytes we have to receive before we can deliver validated plaintext to the user). The current default segment size is 128KiB.

One block from each segment is sent to each shareholder (aka leaseholder, aka landlord, aka storage node, aka peer). The "share" held by each remote shareholder is nominally just a collection of these blocks. The file will be recoverable when a certain number of shares have been retrieved.

The blocks are hashed as they are generated and transmitted. These block hashes are put into a Merkle hash tree. When the last share has been created, the merkle tree is completed and delivered to the peer. Later, when we retrieve these blocks, the peer will send many of the merkle hash tree nodes ahead of time, so we can validate each block independently.

The root of this block hash tree is called the "block root hash" and used in the next step.

There is a higher-level Merkle tree called the "share hash tree". Its leaves are the block root hashes from each share. The root of this tree is called the "share root hash" and is included in the "URI Extension Block", aka UEB. The ciphertext hash and Merkle tree are also put here, along with the original file size, and the encoding parameters. The UEB contains all the non-secret values that could be put in the URI, but would have made the URI too big. So instead, the UEB is stored with the share, and the hash of the UEB is put in the URI.

The URI then contains the secret encryption key and the UEB hash. It also contains the basic encoding parameters (k and N) and the file size, to make download more efficient (by knowing the number of required shares ahead of time, sufficient download queries can be generated in parallel).

The URI (also known as the immutable-file read-cap, since possessing it grants the holder the capability to read the file's plaintext) is then represented as a (relatively) short printable string like so:

```
URI:CHK:auxet66ynq55naiy2ay7cgrshm:6rudoctmbxsmbg7gwtjlimd6umtwrrsxkjzthuldsmo4nnfoc6fa:3:10:1000000
```

During download, when a peer begins to transmit a share, it first transmits all of the parts of the share hash tree that are necessary to validate its block root hash. Then it transmits the portions of the block hash tree that are necessary to validate the first block. Then it transmits the first block. It then continues this loop: transmitting any portions of the block hash tree to validate block#N, then sending block#N.

So the "share" that is sent to the remote peer actually consists of three pieces, sent in a specific order as they become available, and retrieved during download in a different order according to when they are needed.

The first piece is the blocks themselves, one per segment. The last block will likely be shorter than the rest, because the last segment is probably shorter than the rest. The second piece is the block hash tree, consisting of a total of two SHA-1 hashes per block. The third piece is a hash chain from the share hash tree, consisting of log2(numshares) hashes.

During upload, all blocks are sent first, followed by the block hash tree, followed by the share hash chain. During download, the share hash chain is delivered first, followed by the block root hash. The client then uses the hash chain to validate the block root hash. Then the peer delivers enough of the block hash tree to validate the first block, followed by the first block itself. The block hash chain is used to validate the block, then it is passed (along with the first block from several other peers) into decoding, to produce the first segment of crypttext, which is then decrypted to produce the first segment of plaintext, which is finally delivered to the user.

Hashes

All hashes use SHA-256d, as defined in Practical Cryptography (by Ferguson and Schneier). All hashes use a single-purpose tag, e.g. the hash that converts an encryption key into a storage index is defined as follows:

```
SI = SHA256d(netstring("allmydata_immutable_key_to_storage_index_v1") + key)
```

When two separate values need to be combined together in a hash, we wrap each in a netstring.

Using SHA-256d (instead of plain SHA-256) guards against length-extension attacks. Using the tag protects our Merkle trees against attacks in which the hash of a leaf is confused with a hash of two children (allowing an attacker to generate corrupted data that nevertheless appears to be valid), and is simply good "cryptograhic hygiene". The "Chosen Protocol Attack" by Kelsey, Schneier, and Wagner is relevant. Putting the tag in a netstring guards against attacks that seek to confuse the end of the tag with the beginning of the subsequent value.

URI Extension Block

This block is a serialized dictionary with string keys and string values (some of which represent numbers, some of which are SHA-256 hashes). All buckets hold an identical copy. The hash of the serialized data is kept in the URI.

The download process must obtain a valid copy of this data before any decoding can take place. The download process must also obtain other data before incremental validation can be performed. Full-file validation (for clients who do not wish to do incremental validation) can be performed solely with the data from this block.

At the moment, this data block contains the following keys (and an estimate on their sizes):

```
size                5
segment_size        7
num_segments        2
needed_shares       2
total_shares        3

codec_name          3
codec_params        5+1+2+1+3=12
tail_codec_params   12

share_root_hash     32 (binary) or 52 (base32-encoded) each
plaintext_hash
plaintext_root_hash
crypttext_hash
crypttext_root_hash
```

Some pieces are needed elsewhere (size should be visible without pulling the block, the Tahoe3 algorithm needs total_shares to find the right peers, all peer selection algorithms need needed_shares to ask a minimal set of peers). Some pieces are arguably redundant but are convenient to have present (test_encode.py makes use of num_segments).

The rule for this data block is that it should be a constant size for all files, regardless of file size. Therefore hash trees (which have a size that depends linearly upon the number of segments) are stored elsewhere in the bucket, with only the hash tree root stored in this data block.

This block will be serialized as follows:

```
assert that all keys match ^[a-zA-z_\-]+$
sort all the keys lexicographically
for k in keys:
 write("%s:" % k)
 write(netstring(data[k]))
```

Serialized size:

```
dense binary (but decimal) packing: 160+46=206
including 'key:' (185) and netstring (6*3+7*4=46) on values: 231
including 'key:%d\n' (185+13=198) and printable values (46+5*52=306)=504
```

We'll go with the 231-sized block, and provide a tool to dump it as text if we really want one.

Mutable Files

Mutable files are places with a stable identifier that can hold data that changes over time. In contrast to immutable slots, for which the identifier/capability is derived from the contents themselves, the mutable file identifier remains fixed for the life of the slot, regardless of what data is placed inside it.

Each mutable file is referenced by two different caps. The "read-write" cap grants read-write access to its holder, allowing them to put whatever contents they like into the slot. The "read-only" cap is less powerful, only granting read access, and not enabling modification of the data. The read-write cap can be turned into the read-only cap, but not the other way around.

The data in these files is distributed over a number of servers, using the same erasure coding that immutable files use, with 3-of-10 being a typical choice of encoding parameters. The data is encrypted and signed in such a way that only the holders of the read-write cap will be able to set the contents of the slot, and only the holders of the read-only cap will be able to read those contents. Holders of either cap will be able to validate the contents as being written by someone with the read-write cap. The servers who hold the shares are not automatically given the ability read or modify them: the worst they can do is deny service (by deleting or corrupting the shares), or attempt a rollback attack (which can only succeed with the cooperation of at least k servers).

Mutable Formats

History

When mutable files first shipped in Tahoe-0.8.0 (15-Feb-2008), the only version available was "SDMF", described below. This was a limited-functionality placeholder, intended to be replaced with improved-efficiency "MDMF" files shortly afterwards. The development process took longer than expected, and MDMF didn't ship until Tahoe-1.9.0 (31-Oct-2011), and even then it was opt-in (not used by default).

SDMF was intended for relatively small mutable files, up to a few megabytes. It uses only one segment, so alacrity (the measure of how quickly the first byte of plaintext is returned to the client) suffers, as the whole file must be downloaded even if you only want to get a single byte. The memory used by both clients and servers also scales with the size of the file, instead of being limited to the half-a-MB-or-so that immutable file operations use, so large files cause significant memory usage. To discourage the use of SDMF outside it's design parameters, the early versions of Tahoe enforced a maximum size on mutable files (maybe 10MB). Since most directories are built out of mutable files, this imposed a limit of about 30k entries per directory. In subsequent releases, this limit was removed, but the performance problems inherent in the SDMF implementation remained.

In the summer of 2010, Google-Summer-of-Code student Kevan Carstensen took on the project of finally implementing MDMF. Because of my (Brian) design mistake in SDMF (not including a separate encryption seed in each segment), the share format for SDMF could not be used for MDMF, resulting in a larger gap between the two implementations (my original intention had been to make SDMF a clean subset of MDMF, where any single-segment MDMF file could be handled by the old SDMF code). In the fall of 2011, Kevan's code was finally integrated, and first made available in the Tahoe-1.9.0 release.

SDMF vs. MDMF

The improvement of MDMF is the use of multiple segments: individual 128-KiB sections of the file can be retrieved or modified independently. The improvement can be seen when fetching just a portion of the file (using a Range: header on the webapi), or when modifying a portion (again with a Range: header). It can also be seen indirectly when fetching the whole file: the first segment of data should be delivered faster from a large MDMF file than from an SDMF file, although the overall download will then proceed at the same rate.

We've decided to make it opt-in for now: mutable files default to SDMF format unless explicitly configured to use MDMF, either in `tahoe.cfg` (see *Configuring a Tahoe-LAFS node*) or in the WUI or CLI command that created a new mutable file.

The code can read and modify existing files of either format without user intervention. We expect to make MDMF the default in a subsequent release, perhaps 2.0.

Which format should you use? SDMF works well for files up to a few MB, and can be handled by older versions (Tahoe-1.8.3 and earlier). If you do not need to support older clients, want to efficiently work with mutable files, and have code which will use Range: headers that make partial reads and writes, then MDMF is for you.

Consistency vs. Availability

There is an age-old battle between consistency and availability. Epic papers have been written, elaborate proofs have been established, and generations of theorists have learned that you cannot simultaneously achieve guaranteed consistency with guaranteed reliability. In addition, the closer to 0 you get on either axis, the cost and complexity of the design goes up.

Tahoe's design goals are to largely favor design simplicity, then slightly favor read availability, over the other criteria.

As we develop more sophisticated mutable slots, the API may expose multiple read versions to the application layer. The tahoe philosophy is to defer most consistency recovery logic to the higher layers. Some applications have effective ways to merge multiple versions, so inconsistency is not necessarily a problem (i.e. directory nodes can usually merge multiple "add child" operations).

The Prime Coordination Directive: "Don't Do That"

The current rule for applications which run on top of Tahoe is "do not perform simultaneous uncoordinated writes". That means you need non-tahoe means to make sure that two parties are not trying to modify the same mutable slot at the same time. For example:

- don't give the read-write URI to anyone else. Dirnodes in a private directory generally satisfy this case, as long as you don't use two clients on the same account at the same time
- if you give a read-write URI to someone else, stop using it yourself. An inbox would be a good example of this.
- if you give a read-write URI to someone else, call them on the phone before you write into it
- build an automated mechanism to have your agents coordinate writes. For example, we expect a future release to include a FURL for a "coordination server" in the dirnodes. The rule can be that you must contact the coordination server and obtain a lock/lease on the file before you're allowed to modify it.

If you do not follow this rule, Bad Things will happen. The worst-case Bad Thing is that the entire file will be lost. A less-bad Bad Thing is that one or more of the simultaneous writers will lose their changes. An observer of the file may not see monotonically-increasing changes to the file, i.e. they may see version 1, then version 2, then 3, then 2 again.

Tahoe takes some amount of care to reduce the badness of these Bad Things. One way you can help nudge it from the "lose your file" case into the "lose some changes" case is to reduce the number of competing versions: multiple versions of the file that different parties are trying to establish as the one true current contents. Each simultaneous writer counts as a "competing version", as does the previous version of the file. If the count "S" of these competing versions is larger than N/k, then the file runs the risk of being lost completely. [TODO] If at least one of the writers remains running after the collision is detected, it will attempt to recover, but if S>(N/k) and all writers crash after writing a few shares, the file will be lost.

Note that Tahoe uses serialization internally to make sure that a single Tahoe node will not perform simultaneous modifications to a mutable file. It accomplishes this by using a weakref cache of the MutableFileNode (so that there will never be two distinct MutableFileNodes for the same file), and by forcing all mutable file operations to obtain a per-node lock before they run. The Prime Coordination Directive therefore applies to inter-node conflicts, not intra-node ones.

Small Distributed Mutable Files

SDMF slots are suitable for small (<1MB) files that are editing by rewriting the entire file. The three operations are:

- allocate (with initial contents)
- set (with new contents)
- get (old contents)

The first use of SDMF slots will be to hold directories (dirnodes), which map encrypted child names to rw-URI/ro-URI pairs.

SDMF slots overview

Each SDMF slot is created with a public/private key pair. The public key is known as the "verification key", while the private key is called the "signature key". The private key is hashed and truncated to 16 bytes to form the "write key" (an AES symmetric key). The write key is then hashed and truncated to form the "read key". The read key is hashed and truncated to form the 16-byte "storage index" (a unique string used as an index to locate stored data).

The public key is hashed by itself to form the "verification key hash".

The write key is hashed a different way to form the "write enabler master". For each storage server on which a share is kept, the write enabler master is concatenated with the server's nodeid and hashed, and the result is called the "write enabler" for that particular server. Note that multiple shares of the same slot stored on the same server will all get the same write enabler, i.e. the write enabler is associated with the "bucket", rather than the individual shares.

The private key is encrypted (using AES in counter mode) by the write key, and the resulting crypttext is stored on the servers. so it will be retrievable by anyone who knows the write key. The write key is not used to encrypt anything else, and the private key never changes, so we do not need an IV for this purpose.

The actual data is encrypted (using AES in counter mode) with a key derived by concatenating the readkey with the IV, the hashing the results and truncating to 16 bytes. The IV is randomly generated each time the slot is updated, and stored next to the encrypted data.

The read-write URI consists of the write key and the verification key hash. The read-only URI contains the read key and the verification key hash. The verify-only URI contains the storage index and the verification key hash.

```
URI:SSK-RW:b2a(writekey):b2a(verification_key_hash)
URI:SSK-RO:b2a(readkey):b2a(verification_key_hash)
URI:SSK-Verify:b2a(storage_index):b2a(verification_key_hash)
```

Note that this allows the read-only and verify-only URIs to be derived from the read-write URI without actually retrieving the public keys. Also note that it means the read-write agent must validate both the private key and the public key when they are first fetched. All users validate the public key in exactly the same way.

The SDMF slot is allocated by sending a request to the storage server with a desired size, the storage index, and the write enabler for that server's nodeid. If granted, the write enabler is stashed inside the slot's backing store file. All further write requests must be accompanied by the write enabler or they will not be honored. The storage server does not share the write enabler with anyone else.

The SDMF slot structure will be described in more detail below. The important pieces are:

- a sequence number
- a root hash "R"
- the encoding parameters (including k, N, file size, segment size)
- a signed copy of [seqnum,R,encoding_params], using the signature key
- the verification key (not encrypted)
- the share hash chain (part of a Merkle tree over the share hashes)
- the block hash tree (Merkle tree over blocks of share data)
- the share data itself (erasure-coding of read-key-encrypted file data)
- the signature key, encrypted with the write key

The access pattern for read is:

- hash read-key to get storage index
- use storage index to locate 'k' shares with identical 'R' values
  - either get one share, read 'k' from it, then read k-1 shares
  - or read, say, 5 shares, discover k, either get more or be finished
  - or copy k into the URIs
- read verification key
- hash verification key, compare against verification key hash
- read seqnum, R, encoding parameters, signature
- verify signature against verification key
- read share data, compute block-hash Merkle tree and root "r"
- read share hash chain (leading from "r" to "R")
- validate share hash chain up to the root "R"
- submit share data to erasure decoding

- decrypt decoded data with read-key
- submit plaintext to application

The access pattern for write is:

- hash write-key to get read-key, hash read-key to get storage index
- use the storage index to locate at least one share
- read verification key and encrypted signature key
- decrypt signature key using write-key
- hash signature key, compare against write-key
- hash verification key, compare against verification key hash
- encrypt plaintext from application with read-key
  - application can encrypt some data with the write-key to make it only available to writers (use this for transitive read-onlyness of dirnodes)
- erasure-code crypttext to form shares
- split shares into blocks
- compute Merkle tree of blocks, giving root "r" for each share
- compute Merkle tree of shares, find root "R" for the file as a whole
- create share data structures, one per server:
  - use seqnum which is one higher than the old version
  - share hash chain has log(N) hashes, different for each server
  - signed data is the same for each server
- now we have N shares and need homes for them
- walk through peers
  - if share is not already present, allocate-and-set
  - otherwise, try to modify existing share:
  - send testv_and_writev operation to each one
  - testv says to accept share if their(seqnum+R) <= our(seqnum+R)
  - count how many servers wind up with which versions (histogram over R)
  - keep going until N servers have the same version, or we run out of servers
    - if any servers wound up with a different version, report error to application
    - if we ran out of servers, initiate recovery process (described below)

Server Storage Protocol

The storage servers will provide a mutable slot container which is oblivious to the details of the data being contained inside it. Each storage index refers to a "bucket", and each bucket has one or more shares inside it. (In a well-provisioned network, each bucket will have only one share). The bucket is stored as a directory, using the base32-encoded storage index as the directory name. Each share is stored in a single file, using the share number as the filename.

The container holds space for a container magic number (for versioning), the write enabler, the nodeid which accepted the write enabler (used for share migration, described below), a small number of lease structures, the embedded data itself, and expansion space for additional lease structures:

```
#   offset    size    name
1   0         32      magic verstr "Tahoe mutable container v1\n\x75\x09\x44\x03\x8e"
2   32        20      write enabler's nodeid
3   52        32      write enabler
4   84        8       data size (actual share data present) (a)
5   92        8       offset of (8) count of extra leases (after data)
6   100       368     four leases, 92 bytes each
                       0    4   ownerid (0 means "no lease here")
                       4    4   expiration timestamp
                       8   32   renewal token
                       40  32   cancel token
                       72  20   nodeid which accepted the tokens
7   468       (a)     data
8   ??        4       count of extra leases
9   ??        n*92    extra leases
```

The "extra leases" field must be copied and rewritten each time the size of the enclosed data changes. The hope is that most buckets will have four or fewer leases and this extra copying will not usually be necessary.

The (4) "data size" field contains the actual number of bytes of data present in field (7), such that a client request to read beyond 504+(a) will result in an error. This allows the client to (one day) read relative to the end of the file. The container size (that is, (8)-(7)) might be larger, especially if extra size was pre-allocated in anticipation of filling the container with a lot of data.

The offset in (5) points at the *count* of extra leases, at (8). The actual leases (at (9)) begin 4 bytes later. If the container size changes, both (8) and (9) must be relocated by copying.

The server will honor any write commands that provide the write token and do not exceed the server-wide storage size limitations. Read and write commands MUST be restricted to the 'data' portion of the container: the implementation of those commands MUST perform correct bounds-checking to make sure other portions of the container are inaccessible to the clients.

The two methods provided by the storage server on these "MutableSlot" share objects are:

- readv(ListOf(offset=int, length=int))
  - returns a list of bytestrings, of the various requested lengths
  - offset < 0 is interpreted relative to the end of the data
  - spans which hit the end of the data will return truncated data
- testv_and_writev(write_enabler, test_vector, write_vector)
  - this is a test-and-set operation which performs the given tests and only applies the desired writes if all tests succeed. This is used to detect simultaneous writers, and to reduce the chance that an update will lose data recently written by some other party (written after the last time this slot was read).
  - test_vector=ListOf(TupleOf(offset, length, opcode, specimen))
  - the opcode is a string, from the set [gt, ge, eq, le, lt, ne]
  - each element of the test vector is read from the slot's data and compared against the specimen using the desired (in)equality. If all tests evaluate True, the write is performed
  - write_vector=ListOf(TupleOf(offset, newdata))
    - offset < 0 is not yet defined, it probably means relative to the end of the data, which probably means append, but we haven't nailed it down quite yet

        * write vectors are executed in order, which specifies the results of overlapping writes
    - return value:
        * error: OutOfSpace
        * error: something else (io error, out of memory, whatever)
        * (True, old_test_data): the write was accepted (test_vector passed)
        * (False, old_test_data): the write was rejected (test_vector failed)
            · both 'accepted' and 'rejected' return the old data that was used for the test_vector comparison. This can be used by the client to detect write collisions, including collisions for which the desired behavior was to overwrite the old version.

In addition, the storage server provides several methods to access these share objects:

- allocate_mutable_slot(storage_index, sharenums=SetOf(int))
    - returns DictOf(int, MutableSlot)
- get_mutable_slot(storage_index)
    - returns DictOf(int, MutableSlot)
    - or raises KeyError

We intend to add an interface which allows small slots to allocate-and-write in a single call, as well as do update or read in a single call. The goal is to allow a reasonably-sized dirnode to be created (or updated, or read) in just one round trip (to all N shareholders in parallel).

migrating shares

If a share must be migrated from one server to another, two values become invalid: the write enabler (since it was computed for the old server), and the lease renew/cancel tokens.

Suppose that a slot was first created on nodeA, and was thus initialized with WE(nodeA) (= H(WEM+nodeA)). Later, for provisioning reasons, the share is moved from nodeA to nodeB.

Readers may still be able to find the share in its new home, depending upon how many servers are present in the grid, where the new nodeid lands in the permuted index for this particular storage index, and how many servers the reading client is willing to contact.

When a client attempts to write to this migrated share, it will get a "bad write enabler" error, since the WE it computes for nodeB will not match the WE(nodeA) that was embedded in the share. When this occurs, the "bad write enabler" message must include the old nodeid (e.g. nodeA) that was in the share.

The client then computes H(nodeB+H(WEM+nodeA)), which is the same as H(nodeB+WE(nodeA)). The client sends this along with the new WE(nodeB), which is H(WEM+nodeB). Note that the client only sends WE(nodeB) to nodeB, never to anyone else. Also note that the client does not send a value to nodeB that would allow the node to impersonate the client to a third node: everything sent to nodeB will include something specific to nodeB in it.

The server locally computes H(nodeB+WE(nodeA)), using its own node id and the old write enabler from the share. It compares this against the value supplied by the client. If they match, this serves as proof that the client was able to compute the old write enabler. The server then accepts the client's new WE(nodeB) and writes it into the container.

This WE-fixup process requires an extra round trip, and requires the error message to include the old nodeid, but does not require any public key operations on either client or server.

Migrating the leases will require a similar protocol. This protocol will be defined concretely at a later date.

Code Details

The MutableFileNode class is used to manipulate mutable files (as opposed to ImmutableFileNodes). These are initially generated with client.create_mutable_file(), and later recreated from URIs with client.create_node_from_uri(). Instances of this class will contain a URI and a reference to the client (for peer selection and connection).

NOTE: this section is out of date. Please see src/allmydata/interfaces.py (the section on IMutableFilesystemNode) for more accurate information.

The methods of MutableFileNode are:

- download_to_data() -> [deferred] newdata, NotEnoughSharesError
  - if there are multiple retrieveable versions in the grid, get() returns the first version it can reconstruct, and silently ignores the others. In the future, a more advanced API will signal and provide access to the multiple heads.
- update(newdata) -> OK, UncoordinatedWriteError, NotEnoughSharesError
- overwrite(newdata) -> OK, UncoordinatedWriteError, NotEnoughSharesError

download_to_data() causes a new retrieval to occur, pulling the current contents from the grid and returning them to the caller. At the same time, this call caches information about the current version of the file. This information will be used in a subsequent call to update(), and if another change has occured between the two, this information will be out of date, triggering the UncoordinatedWriteError.

update() is therefore intended to be used just after a download_to_data(), in the following pattern:

```
d = mfn.download_to_data()
d.addCallback(apply_delta)
d.addCallback(mfn.update)
```

If the update() call raises UCW, then the application can simply return an error to the user ("you violated the Prime Coordination Directive"), and they can try again later. Alternatively, the application can attempt to retry on its own. To accomplish this, the app needs to pause, download the new (post-collision and post-recovery) form of the file, reapply their delta, then submit the update request again. A randomized pause is necessary to reduce the chances of colliding a second time with another client that is doing exactly the same thing:

```
d = mfn.download_to_data()
d.addCallback(apply_delta)
d.addCallback(mfn.update)
def _retry(f):
  f.trap(UncoordinatedWriteError)
  d1 = pause(random.uniform(5, 20))
  d1.addCallback(lambda res: mfn.download_to_data())
  d1.addCallback(apply_delta)
  d1.addCallback(mfn.update)
  return d1
d.addErrback(_retry)
```

Enthusiastic applications can retry multiple times, using a randomized exponential backoff between each. A particularly enthusiastic application can retry forever, but such apps are encouraged to provide a means to the user of giving up after a while.

UCW does not mean that the update was not applied, so it is also a good idea to skip the retry-update step if the delta was already applied:

```
d = mfn.download_to_data()
d.addCallback(apply_delta)
d.addCallback(mfn.update)
```

```
def _retry(f):
  f.trap(UncoordinatedWriteError)
  d1 = pause(random.uniform(5, 20))
  d1.addCallback(lambda res: mfn.download_to_data())
  def _maybe_apply_delta(contents):
    new_contents = apply_delta(contents)
    if new_contents != contents:
      return mfn.update(new_contents)
  d1.addCallback(_maybe_apply_delta)
  return d1
d.addErrback(_retry)
```

update() is the right interface to use for delta-application situations, like directory nodes (in which apply_delta might be adding or removing child entries from a serialized table).

Note that any uncoordinated write has the potential to lose data. We must do more analysis to be sure, but it appears that two clients who write to the same mutable file at the same time (even if both eventually retry) will, with high probability, result in one client observing UCW and the other silently losing their changes. It is also possible for both clients to observe UCW. The moral of the story is that the Prime Coordination Directive is there for a reason, and that recovery/UCW/retry is not a subsitute for write coordination.

overwrite() tells the client to ignore this cached version information, and to unconditionally replace the mutable file's contents with the new data. This should not be used in delta application, but rather in situations where you want to replace the file's contents with completely unrelated ones. When raw files are uploaded into a mutable slot through the Tahoe-LAFS web-API (using POST and the ?mutable=true argument), they are put in place with overwrite().

The peer-selection and data-structure manipulation (and signing/verification) steps will be implemented in a separate class in allmydata/mutable.py .

SMDF Slot Format

This SMDF data lives inside a server-side MutableSlot container. The server is oblivious to this format.

This data is tightly packed. In particular, the share data is defined to run all the way to the beginning of the encrypted private key (the encprivkey offset is used both to terminate the share data and to begin the encprivkey).

```
 #    offset   size    name
 1    0        1       version byte, \x00 for this format
 2    1        8       sequence number. 2^64-1 must be handled specially, TBD
 3    9        32      "R" (root of share hash Merkle tree)
 4    41       16      IV (share data is AES(H(readkey+IV)) )
 5    57       18      encoding parameters:
       57       1        k
       58       1        N
       59       8        segment size
       67       8        data length (of original plaintext)
 6    75       32      offset table:
       75       4        (8) signature
       79       4        (9) share hash chain
       83       4        (10) block hash tree
       87       4        (11) share data
       91       8        (12) encrypted private key
       99       8        (13) EOF
 7    107      436ish  verification key (2048 RSA key)
 8    543ish   256ish  signature=RSAsign(sigkey, H(version+seqnum+r+IV+encparm))
 9    799ish   (a)     share hash chain, encoded as:
                       "".join([pack(">H32s", shnum, hash)
```

```
                                  for (shnum,hash) in needed_hashes])
10    (927ish) (b)     block hash tree, encoded as:
                        "".join([pack(">32s",hash) for hash in block_hash_tree])
11    (935ish) LEN     share data (no gap between this and encprivkey)
12    ??       1216ish encrypted private key= AESenc(write-key, RSA-key)
13    ??       --      EOF

(a) The share hash chain contains ceil(log(N)) hashes, each 32 bytes long.
   This is the set of hashes necessary to validate this share's leaf in the
   share Merkle tree. For N=10, this is 4 hashes, i.e. 128 bytes.
(b) The block hash tree contains ceil(length/segsize) hashes, each 32 bytes
   long. This is the set of hashes necessary to validate any given block of
   share data up to the per-share root "r". Each "r" is a leaf of the share
   has tree (with root "R"), from which a minimal subset of hashes is put in
   the share hash chain in (8).
```

Recovery

The first line of defense against damage caused by colliding writes is the Prime Coordination Directive: "Don't Do That".

The second line of defense is to keep "S" (the number of competing versions) lower than N/k. If this holds true, at least one competing version will have k shares and thus be recoverable. Note that server unavailability counts against us here: the old version stored on the unavailable server must be included in the value of S.

The third line of defense is our use of testv_and_writev() (described below), which increases the convergence of simultaneous writes: one of the writers will be favored (the one with the highest "R"), and that version is more likely to be accepted than the others. This defense is least effective in the pathological situation where S simultaneous writers are active, the one with the lowest "R" writes to N-k+1 of the shares and then dies, then the one with the next-lowest "R" writes to N-2k+1 of the shares and dies, etc, until the one with the highest "R" writes to k-1 shares and dies. Any other sequencing will allow the highest "R" to write to at least k shares and establish a new revision.

The fourth line of defense is the fact that each client keeps writing until at least one version has N shares. This uses additional servers, if necessary, to make sure that either the client's version or some newer/overriding version is highly available.

The fifth line of defense is the recovery algorithm, which seeks to make sure that at least *one* version is highly available, even if that version is somebody else's.

The write-shares-to-peers algorithm is as follows:

- permute peers according to storage index
- walk through peers, trying to assign one share per peer
- for each peer:
  - send testv_and_writev, using "old(seqnum+R) <= our(seqnum+R)" as the test
    - this means that we will overwrite any old versions, and we will overwrite simultaenous writers of the same version if our R is higher. We will not overwrite writers using a higher seqnum.
  - record the version that each share winds up with. If the write was accepted, this is our own version. If it was rejected, read the old_test_data to find out what version was retained.
  - if old_test_data indicates the seqnum was equal or greater than our own, mark the "Simultanous Writes Detected" flag, which will eventually result in an error being reported to the writer (in their close() call).
  - build a histogram of "R" values

  - repeat until the histogram indicate that some version (possibly ours) has N shares. Use new servers if necessary.
  - If we run out of servers:
    * if there are at least shares-of-happiness of any one version, we're happy, so return. (the close() might still get an error)
    * not happy, need to reinforce something, goto RECOVERY

Recovery:

- read all shares, count the versions, identify the recoverable ones, discard the unrecoverable ones.
- sort versions: locate max(seqnums), put all versions with that seqnum in the list, sort by number of outstanding shares. Then put our own version. (TODO: put versions with seqnum <max but >us ahead of us?).
- for each version:
  - attempt to recover that version
  - if not possible, remove it from the list, go to next one
  - if recovered, start at beginning of peer list, push that version, continue until N shares are placed
  - if pushing our own version, bump up the seqnum to one higher than the max seqnum we saw
  - if we run out of servers:
    * schedule retry and exponential backoff to repeat RECOVERY
  - admit defeat after some period? presumeably the client will be shut down eventually, maybe keep trying (once per hour?) until then.

Medium Distributed Mutable Files

These are just like the SDMF case, but:

- We actually take advantage of the Merkle hash tree over the blocks, by reading a single segment of data at a time (and its necessary hashes), to reduce the read-time alacrity.
- We allow arbitrary writes to any range of the file.
- We add more code to first read each segment that a write must modify. This looks exactly like the way a normal filesystem uses a block device, or how a CPU must perform a cache-line fill before modifying a single word.
- We might implement some sort of copy-based atomic update server call, to allow multiple writev() calls to appear atomic to any readers.

MDMF slots provide fairly efficient in-place edits of very large files (a few GB). Appending data is also fairly efficient.

Large Distributed Mutable Files

LDMF slots (not implemented) would use a fundamentally different way to store the file, inspired by Mercurial's "revlog" format. This would enable very efficient insert/remove/replace editing of arbitrary spans. Multiple versions of the file can be retained, in a revision graph that can have multiple heads. Each revision can be referenced by a cryptographic identifier. There are two forms of the URI, one that means "most recent version", and a longer one that points to a specific revision.

Metadata can be attached to the revisions, like timestamps, to enable rolling back an entire tree to a specific point in history.

LDMF1 provides deltas but tries to avoid dealing with multiple heads. LDMF2 provides explicit support for revision identifiers and branching.

TODO

improve allocate-and-write or get-writer-buckets API to allow one-call (or maybe two-call) updates. The challenge is in figuring out which shares are on which machines. First cut will have lots of round trips.

(eventually) define behavior when seqnum wraps. At the very least make sure it can't cause a security problem. "the slot is worn out" is acceptable.

(eventually) define share-migration lease update protocol. Including the nodeid who accepted the lease is useful, we can use the same protocol as we do for updating the write enabler. However we need to know which lease to update.. maybe send back a list of all old nodeids that we find, then try all of them when we accept the update?

We now do this in a specially-formatted IndexError exception: "UNABLE to renew non-existent lease. I have leases accepted by " + "nodeids: '12345','abcde','44221' ."

confirm that a repairer can regenerate shares without the private key. Hmm, without the write-enabler they won't be able to write those shares to the servers.. although they could add immutable new shares to new servers.

Tahoe-LAFS Directory Nodes

As explained in the architecture docs, Tahoe-LAFS can be roughly viewed as a collection of three layers. The lowest layer is the key-value store: it provides operations that accept files and upload them to the grid, creating a URI in the process which securely references the file's contents. The middle layer is the file store, creating a structure of directories and filenames resembling the traditional Unix or Windows filesystems. The top layer is the application layer, which uses the lower layers to provide useful services to users, like a backup application, or a way to share files with friends.

This document examines the middle layer, the "file store".

Key-value Store Primitives

In the lowest layer (key-value store), there are two operations that reference immutable data (which we refer to as "CHK URIs" or "CHK read-capabilities" or "CHK read-caps"). One puts data into the grid (but only if it doesn't exist already), the other retrieves it:

```
chk_uri = put(data)
data = get(chk_uri)
```

We also have three operations which reference mutable data (which we refer to as "mutable slots", or "mutable write-caps and read-caps", or sometimes "SSK slots"). One creates a slot with some initial contents, a second replaces the contents of a pre-existing slot, and the third retrieves the contents:

```
mutable_uri = create(initial_data)
replace(mutable_uri, new_data)
data = get(mutable_uri)
```

File Store Goals

The main goal for the middle (file store) layer is to give users a way to organize the data that they have uploaded into the grid. The traditional way to do this in computer filesystems is to put this data into files, give those files names, and collect these names into directories.

Each directory is a set of name-entry pairs, each of which maps a "child name" to a directory entry pointing to an object of some kind. Those child objects might be files, or they might be other directories. Each directory entry also contains metadata.

The directory structure is therefore a directed graph of nodes, in which each node might be a directory node or a file node. All file nodes are terminal nodes.

Dirnode Goals

What properties might be desirable for these directory nodes? In no particular order:

1. functional. Code which does not work doesn't count.
2. easy to document, explain, and understand
3. confidential: it should not be possible for others to see the contents of a directory
4. integrity: it should not be possible for others to modify the contents of a directory
5. available: directories should survive host failure, just like files do
6. efficient: in storage, communication bandwidth, number of round-trips
7. easy to delegate individual directories in a flexible way
8. updateness: everybody looking at a directory should see the same contents
9. monotonicity: everybody looking at a directory should see the same sequence of updates

Some of these goals are mutually exclusive. For example, availability and consistency are opposing, so it is not possible to achieve #5 and #8 at the same time. Moreover, it takes a more complex architecture to get close to the available-and-consistent ideal, so #2/#6 is in opposition to #5/#8.

Tahoe-LAFS v0.7.0 introduced distributed mutable files, which use public-key cryptography for integrity, and erasure coding for availability. These achieve roughly the same properties as immutable CHK files, but their contents can be

replaced without changing their identity. Dirnodes are then just a special way of interpreting the contents of a specific mutable file. Earlier releases used a "vdrive server": this server was abolished in the v0.7.0 release.

For details of how mutable files work, please see *Mutable Files*.

For releases since v0.7.0, we achieve most of our desired properties. The integrity and availability of dirnodes is equivalent to that of regular (immutable) files, with the exception that there are more simultaneous-update failure modes for mutable slots. Delegation is quite strong: you can give read-write or read-only access to any subtree, and the data format used for dirnodes is such that read-only access is transitive: i.e. if you grant Bob read-only access to a parent directory, then Bob will get read-only access (and *not* read-write access) to its children.

Relative to the previous "vdrive server"-based scheme, the current distributed dirnode approach gives better availability, but cannot guarantee updateness quite as well, and requires far more network traffic for each retrieval and update. Mutable files are somewhat less available than immutable files, simply because of the increased number of combinations (shares of an immutable file are either present or not, whereas there are multiple versions of each mutable file, and you might have some shares of version 1 and other shares of version 2). In extreme cases of simultaneous update, mutable files might suffer from non-monotonicity.

Dirnode secret values

As mentioned before, dirnodes are simply a special way to interpret the contents of a mutable file, so the secret keys and capability strings described in *Mutable Files* are all the same. Each dirnode contains an RSA public/private keypair, and the holder of the "write capability" will be able to retrieve the private key (as well as the AES encryption key used for the data itself). The holder of the "read capability" will be able to obtain the public key and the AES data key, but not the RSA private key needed to modify the data.

The "write capability" for a dirnode grants read-write access to its contents. This is expressed on concrete form as the "dirnode write cap": a printable string which contains the necessary secrets to grant this access. Likewise, the "read capability" grants read-only access to a dirnode, and can be represented by a "dirnode read cap" string.

For example, URI:DIR2:swdi8ge1s7qko45d3ckkyw1aac%3Aar8r5j99a4mezdojejmsfp4fj1zeky9gjigyrid4urxdimego68o
is a write-capability URI, while URI:DIR2-RO:buxjqykt637u61nnmjg7s8zkny:ar8r5j99a4mezdojejmsfp4fj1zeky9gjigyrid4urxdimego68
is a read-capability URI, both for the same dirnode.

Dirnode storage format

Each dirnode is stored in a single mutable file, distributed in the Tahoe-LAFS grid. The contents of this file are a serialized list of netstrings, one per child. Each child is a list of four netstrings: (name, rocap, rwcap, metadata). (Remember that the contents of the mutable file are encrypted by the read-cap, so this section describes the plaintext contents of the mutable file, *after* it has been decrypted by the read-cap.)

The name is simple a UTF-8 -encoded child name. The 'rocap' is a read-only capability URI to that child, either an immutable (CHK) file, a mutable file, or a directory. It is also possible to store 'unknown' URIs that are not recognized by the current version of Tahoe-LAFS. The 'rwcap' is a read-write capability URI for that child, encrypted with the dirnode's write-cap: this enables the "transitive readonlyness" property, described further below. The 'metadata' is a JSON-encoded dictionary of type,value metadata pairs. Some metadata keys are pre-defined, the rest are left up to the application.

Each rwcap is stored as IV + ciphertext + MAC. The IV is a 16-byte random value. The ciphertext is obtained by using AES in CTR mode on the rwcap URI string, using a key that is formed from a tagged hash of the IV and the dirnode's writekey. The MAC is written only for compatibility with older Tahoe-LAFS versions and is no longer verified.

If Bob has read-only access to the 'bar' directory, and he adds it as a child to the 'foo' directory, then he will put the read-only cap for 'bar' in both the rwcap and rocap slots (encrypting the rwcap contents as described above). If he has full read-write access to 'bar', then he will put the read-write cap in the 'rwcap' slot, and the read-only cap in the

'rocap' slot. Since other users who have read-only access to 'foo' will be unable to decrypt its rwcap slot, this limits those users to read-only access to 'bar' as well, thus providing the transitive readonlyness that we desire.

Dirnode sizes, mutable-file initial read sizes

How big are dirnodes? When reading dirnode data out of mutable files, how large should our initial read be? If we guess exactly, we can read a dirnode in a single round-trip, and update one in two RTT. If we guess too high, we'll waste some amount of bandwidth. If we guess low, we need to make a second pass to get the data (or the encrypted privkey, for writes), which will cost us at least another RTT.

Assuming child names are between 10 and 99 characters long, how long are the various pieces of a dirnode?

```
netstring(name) ~= 4+len(name)
chk-cap = 97 (for 4-char filesizes)
dir-rw-cap = 88
dir-ro-cap = 91
netstring(cap) = 4+len(cap)
encrypted(cap) = 16+cap+32
JSON({}) = 2
JSON({ctime=float,mtime=float,'tahoe':{linkcrtime=float,linkmotime=float}}): 137
netstring(metadata) = 4+137 = 141
```

so a CHK entry is:

```
5+ 4+len(name) + 4+97 + 5+16+97+32 + 4+137
```

And a 15-byte filename gives a 416-byte entry. When the entry points at a subdirectory instead of a file, the entry is a little bit smaller. So an empty directory uses 0 bytes, a directory with one child uses about 416 bytes, a directory with two children uses about 832, etc.

When the dirnode data is encoding using our default 3-of-10, that means we get 139ish bytes of data in each share per child.

The pubkey, signature, and hashes form the first 935ish bytes of the container, then comes our data, then about 1216 bytes of encprivkey. So if we read the first:

```
1kB: we get 65bytes of dirnode data : only empty directories
2kB: 1065bytes: about 8
3kB: 2065bytes: about 15 entries, or 6 entries plus the encprivkey
4kB: 3065bytes: about 22 entries, or about 13 plus the encprivkey
```

So we've written the code to do an initial read of 4kB from each share when we read the mutable file, which should give good performance (one RTT) for small directories.

Design Goals, redux

How well does this design meet the goals?

1. functional: YES: the code works and has extensive unit tests
2. documentable: YES: this document is the existence proof
3. confidential: YES: see below
4. integrity: MOSTLY: a coalition of storage servers can rollback individual mutable files, but not a single one. No server can substitute fake data as genuine.

5. availability: YES: as long as 'k' storage servers are present and have the same version of the mutable file, the dirnode will be available.
6. **efficient: MOSTLY:**

   network: single dirnode lookup is very efficient, since clients can fetch specific keys rather than being required to get or set the entire dirnode each time. Traversing many directories takes a lot of roundtrips, and these can't be collapsed with promise-pipelining because the intermediate values must only be visible to the client. Modifying many dirnodes at once (e.g. importing a large pre-existing directory tree) is pretty slow, since each graph edge must be created independently.

   storage: each child has a separate IV, which makes them larger than if all children were aggregated into a single encrypted string
7. delegation: VERY: each dirnode is a completely independent object, to which clients can be granted separate read-write or read-only access
8. updateness: VERY: with only a single point of access, and no caching, each client operation starts by fetching the current value, so there are no opportunities for staleness
9. monotonicity: VERY: the single point of access also protects against retrograde motion

Confidentiality leaks in the storage servers

Dirnode (and the mutable files upon which they are based) are very private against other clients: traffic between the client and the storage servers is protected by the Foolscap SSL connection, so they can observe very little. Storage index values are hashes of secrets and thus unguessable, and they are not made public, so other clients cannot snoop through encrypted dirnodes that they have not been told about.

Storage servers can observe access patterns and see ciphertext, but they cannot see the plaintext (of child names, metadata, or URIs). If an attacker operates a significant number of storage servers, they can infer the shape of the directory structure by assuming that directories are usually accessed from root to leaf in rapid succession. Since filenames are usually much shorter than read-caps and write-caps, the attacker can use the length of the ciphertext to guess the number of children of each node, and might be able to guess the length of the child names (or at least their sum). From this, the attacker may be able to build up a graph with the same shape as the plaintext file store, but with unlabeled edges and unknown file contents.

Integrity failures in the storage servers

The mutable file's integrity mechanism (RSA signature on the hash of the file contents) prevents the storage server from modifying the dirnode's contents without detection. Therefore the storage servers can make the dirnode unavailable, but not corrupt it.

A sufficient number of colluding storage servers can perform a rollback attack: replace all shares of the whole mutable file with an earlier version. To prevent this, when retrieving the contents of a mutable file, the client queries more servers than necessary and uses the highest available version number. This insures that one or two misbehaving storage servers cannot cause this rollback on their own.

Improving the efficiency of dirnodes

The current mutable-file -based dirnode scheme suffers from certain inefficiencies. A very large directory (with thousands or millions of children) will take a significant time to extract any single entry, because the whole file must be downloaded first, then parsed and searched to find the desired child entry. Likewise, modifying a single child will require the whole file to be re-uploaded.

The current design assumes (and in some cases, requires) that dirnodes remain small. The mutable files on which dirnodes are based are currently using "SDMF" ("Small Distributed Mutable File") design rules, which state that the size of the data shall remain below one megabyte. More advanced forms of mutable files (MDMF and LDMF) are in the design phase to allow efficient manipulation of larger mutable files. This would reduce the work needed to modify a single entry in a large directory.

Judicious caching may help improve the reading-large-directory case. Some form of mutable index at the beginning of the dirnode might help as well. The MDMF design rules allow for efficient random-access reads from the middle of the file, which would give the index something useful to point at.

The current SDMF design generates a new RSA public/private keypair for each directory. This takes considerable time and CPU effort, generally one or two seconds per directory. We have designed (but not yet built) a DSA-based mutable file scheme which will use shared parameters to reduce the directory-creation effort to a bare minimum (picking a random number instead of generating two random primes).

When a backup program is run for the first time, it needs to copy a large amount of data from a pre-existing local filesystem into reliable storage. This means that a large and complex directory structure needs to be duplicated in the dirnode layer. With the one-object-per-dirnode approach described here, this requires as many operations as there are edges in the imported filesystem graph.

Another approach would be to aggregate multiple directories into a single storage object. This object would contain a serialized graph rather than a single name-to-child dictionary. Most directory operations would fetch the whole block of data (and presumeably cache it for a while to avoid lots of re-fetches), and modification operations would need to replace the whole thing at once. This "realm" approach would have the added benefit of combining more data into a single encrypted bundle (perhaps hiding the shape of the graph from a determined attacker), and would reduce round-trips when performing deep directory traversals (assuming the realm was already cached). It would also prevent fine-grained rollback attacks from working: a coalition of storage servers could change the entire realm to look like an earlier state, but it could not independently roll back individual directories.

The drawbacks of this aggregation would be that small accesses (adding a single child, looking up a single child) would require pulling or pushing a lot of unrelated data, increasing network overhead (and necessitating test-and-set semantics for the modification side, which increases the chances that a user operation will fail, making it more challenging to provide promises of atomicity to the user).

It would also make it much more difficult to enable the delegation ("sharing") of specific directories. Since each aggregate "realm" provides all-or-nothing access control, the act of delegating any directory from the middle of the realm would require the realm first be split into the upper piece that isn't being shared and the lower piece that is. This splitting would have to be done in response to what is essentially a read operation, which is not traditionally supposed to be a high-effort action. On the other hand, it may be possible to aggregate the ciphertext, but use distinct encryption keys for each component directory, to get the benefits of both schemes at once.

Dirnode expiration and leases

Dirnodes are created any time a client wishes to add a new directory. How long do they live? What's to keep them from sticking around forever, taking up space that nobody can reach any longer?

Mutable files are created with limited-time "leases", which keep the shares alive until the last lease has expired or been cancelled. Clients which know and care about specific dirnodes can ask to keep them alive for a while, by renewing a lease on them (with a typical period of one month). Clients are expected to assist in the deletion of dirnodes by canceling their leases as soon as they are done with them. This means that when a client unlinks a directory, it should also cancel its lease on that directory. When the lease count on a given share goes to zero, the storage server can delete the related storage. Multiple clients may all have leases on the same dirnode: the server may delete the shares only after all of the leases have gone away.

We expect that clients will periodically create a "manifest": a list of so-called "refresh capabilities" for all of the dirnodes and files that they can reach. They will give this manifest to the "repairer", which is a service that keeps files (and dirnodes) alive on behalf of clients who cannot take on this responsibility for themselves. These refresh

capabilities include the storage index, but do *not* include the readkeys or writekeys, so the repairer does not get to read the files or directories that it is helping to keep alive.

After each change to the user's file store, the client creates a manifest and looks for differences from their previous version. Anything which was removed prompts the client to send out lease-cancellation messages, allowing the data to be deleted.

Starting Points: root dirnodes

Any client can record the URI of a directory node in some external form (say, in a local file) and use it as the starting point of later traversal. Each Tahoe-LAFS user is expected to create a new (unattached) dirnode when they first start using the grid, and record its URI for later use.

Mounting and Sharing Directories

The biggest benefit of this dirnode approach is that sharing individual directories is almost trivial. Alice creates a subdirectory that she wants to use to share files with Bob. This subdirectory is attached to Alice's file store at "alice:shared-with-bob". She asks her file store for the read-only directory URI for that new directory, and emails it to Bob. When Bob receives the URI, he attaches the given URI into one of his own directories, perhaps at a place named "bob:shared-with-alice". Every time Alice writes a file into this directory, Bob will be able to read it. (It is also possible to share read-write URIs between users, but that makes it difficult to follow the Prime Coordination Directive .) Neither Alice nor Bob will get access to any files above the mounted directory: there are no 'parent directory' pointers. If Alice creates a nested set of directories, "alice:shared-with-bob/subdir2", and gives a read-only URI to shared-with-bob to Bob, then Bob will be unable to write to either shared-with-bob/ or subdir2/.

A suitable UI needs to be created to allow users to easily perform this sharing action: dragging a folder from their file store to an IM or email user icon, for example. The UI will need to give the sending user an opportunity to indicate whether they want to grant read-write or read-only access to the recipient. The recipient then needs an interface to drag the new folder into their file store and give it a home.

Revocation

When Alice decides that she no longer wants Bob to be able to access the shared directory, what should she do? Suppose she's shared this folder with both Bob and Carol, and now she wants Carol to retain access to it but Bob to be shut out. Ideally Carol should not have to do anything: her access should continue unabated.

The current plan is to have her client create a deep copy of the folder in question, delegate access to the new folder to the remaining members of the group (Carol), asking the lucky survivors to replace their old reference with the new one. Bob may still have access to the old folder, but he is now the only one who cares: everyone else has moved on, and he will no longer be able to see their new changes. In a strict sense, this is the strongest form of revocation that can be accomplished: there is no point trying to force Bob to forget about the files that he read a moment before being kicked out. In addition it must be noted that anyone who can access the directory can proxy for Bob, reading files to him and accepting changes whenever he wants. Preventing delegation between communication parties is just as pointless as asking Bob to forget previously accessed files. However, there may be value to configuring the UI to ask Carol to not share files with Bob, or to removing all files from Bob's view at the same time his access is revoked.

Servers of Happiness

When you upload a file to a Tahoe-LAFS grid, you expect that it will stay there for a while, and that it will do so even if a few of the peers on the grid stop working, or if something else goes wrong. An upload health metric helps to make sure that this actually happens. An upload health metric is a test that looks at a file on a Tahoe-LAFS grid and says

whether or not that file is healthy; that is, whether it is distributed on the grid in such a way as to ensure that it will probably survive in good enough shape to be recoverable, even if a few things go wrong between the time of the test and the time that it is recovered. Our current upload health metric for immutable files is called 'servers-of-happiness'; its predecessor was called 'shares-of-happiness'.

shares-of-happiness used the number of encoded shares generated by a file upload to say whether or not it was healthy. If there were more shares than a user-configurable threshold, the file was reported to be healthy; otherwise, it was reported to be unhealthy. In normal situations, the upload process would distribute shares fairly evenly over the peers in the grid, and in that case shares-of-happiness worked fine. However, because it only considered the number of shares, and not where they were on the grid, it could not detect situations where a file was unhealthy because most or all of the shares generated from the file were stored on one or two peers.

servers-of-happiness addresses this by extending the share-focused upload health metric to also consider the location of the shares on grid. servers-of-happiness looks at the mapping of peers to the shares that they hold, and compares the cardinality of the largest happy subset of those to a user-configurable threshold. A happy subset of peers has the property that any k (where k is as in k-of-n encoding) peers within the subset can reconstruct the source file. This definition of file health provides a stronger assurance of file availability over time; with 3-of-10 encoding, and happy=7, a healthy file is still guaranteed to be available even if 4 peers fail.

Measuring Servers of Happiness

We calculate servers-of-happiness by computing a matching on a bipartite graph that is related to the layout of shares on the grid. One set of vertices is the peers on the grid, and one set of vertices is the shares. An edge connects a peer and a share if the peer will (or does, for existing shares) hold the share. The size of the maximum matching on this graph is the size of the largest happy peer set that exists for the upload.

First, note that a bipartite matching of size n corresponds to a happy subset of size n. This is because a bipartite matching of size n implies that there are n peers such that each peer holds a share that no other peer holds. Then any k of those peers collectively hold k distinct shares, and can restore the file.

A bipartite matching of size n is not necessary for a happy subset of size n, however (so it is not correct to say that the size of the maximum matching on this graph is the size of the largest happy subset of peers that exists for the upload). For example, consider a file with k = 3, and suppose that each peer has all three of those pieces. Then, since any peer from the original upload can restore the file, if there are 10 peers holding shares, and the happiness threshold is 7, the upload should be declared happy, because there is a happy subset of size 10, and 10 > 7. However, since a maximum matching on the bipartite graph related to this layout has only 3 edges, Tahoe-LAFS declares the upload unhealthy. Though it is not unhealthy, a share layout like this example is inefficient; for k = 3, and if there are n peers, it corresponds to an expansion factor of 10x. Layouts that are declared healthy by the bipartite graph matching approach have the property that they correspond to uploads that are either already relatively efficient in their utilization of space, or can be made to be so by deleting shares; and that place all of the shares that they generate, enabling redistribution of shares later without having to re-encode the file. Also, it is computationally reasonable to compute a maximum matching in a bipartite graph, and there are well-studied algorithms to do that.

Issues

The uploader is good at detecting unhealthy upload layouts, but it doesn't always know how to make an unhealthy upload into a healthy upload if it is possible to do so; it attempts to redistribute shares to achieve happiness, but only in certain circumstances. The redistribution algorithm isn't optimal, either, so even in these cases it will not always find a happy layout if one can be arrived at through redistribution. We are investigating improvements to address these issues.

We don't use servers-of-happiness for mutable files yet; this fix will likely come in Tahoe-LAFS version 1.13.

Upload Strategy of Happiness

As mentioned above, the uploader is good at detecting instances which do not pass the servers-of-happiness test, but the share distribution algorithm is not always successful in instances where happiness can be achieved. A new placement algorithm designed to pass the servers-of-happiness test, titled 'Upload Strategy of Happiness', is meant to fix these instances where the uploader is unable to achieve happiness.

Calculating Share Placements

We calculate share placement like so:

0. Start with an ordered list of servers. Maybe *2N* of them.
1. Query all servers for existing shares.

1a. Query remaining space from all servers. Every server that has enough free space is considered "readwrite" and every server with too little space is "readonly".

2. Construct a bipartite graph G1 of *readonly* servers to pre-existing shares, where an edge exists between an arbitrary readonly server S and an arbitrary share T if and only if S currently holds T.
3. Calculate a maximum matching graph of G1 (a set of S->T edges that has or is-tied-for the highest "happiness score"). There is a clever efficient algorithm for this, named "Ford-Fulkerson". There may be more than one maximum matching for this graph; we choose one of them arbitrarily, but prefer earlier servers. Call this particular placement M1. The placement maps shares to servers, where each share appears at most once, and each server appears at most once.
4. Construct a bipartite graph G2 of readwrite servers to pre-existing shares. Then remove any edge (from G2) that uses a server or a share found in M1. Let an edge exist between server S and share T if and only if S already holds T.
5. Calculate a maximum matching graph of G2, call this M2, again preferring earlier servers.
6. Construct a bipartite graph G3 of (only readwrite) servers to shares (some shares may already exist on a server). Then remove (from G3) any servers and shares used in M1 or M2 (note that we retain servers/shares that were in G1/G2 but *not* in the M1/M2 subsets)
7. Calculate a maximum matching graph of G3, call this M3, preferring earlier servers. The final placement table is the union of M1+M2+M3.
8. Renew the shares on their respective servers from M1 and M2.
9. Upload share T to server S if an edge exists between S and T in M3.
10. If any placements from step 9 fail, mark the server as read-only. Go back to step 2 (since we may discover a server is/has-become read-only, or has failed, during step 9).

Rationale (Step 4): when we see pre-existing shares on read-only servers, we prefer to rely upon those (rather than the ones on read-write servers), so we can maybe use the read-write servers for new shares. If we picked the read-write server's share, then we couldn't re-use that server for new ones (we only rely upon each server for one share, more or less).

Properties of Upload Strategy of Happiness

The size of the maximum bipartite matching is bounded by the size of the smaller set of vertices. Therefore in a situation where the set of servers is smaller than the set of shares, placement is not generated for a subset of shares. In this case the remaining shares are distributed as evenly as possible across the set of writable servers.

If the servers-of-happiness criteria can be met, the upload strategy of happiness guarantees that H shares will be placed on the network. During file repair, if the set of servers is larger than N, the algorithm will only attempt to spread shares over N distinct servers. For both initial file upload and file repair, N should be viewed as the maximum number of distinct servers shares can be placed on, and H as the minimum amount. The uploader will fail if the number of distinct servers is less than H, and it will never attempt to exceed N.

Redundant Array of Independent Clouds: Share To Cloud Mapping

Introduction

This document describes a proposed design for the mapping of LAFS shares to objects in a cloud storage service. It also analyzes the costs for each of the functional requirements, including network, disk, storage and API usage costs.

Terminology

LAFS share A Tahoe-LAFS share representing part of a file after encryption and erasure encoding.

LAFS shareset The set of shares stored by a LAFS storage server for a given storage index. The shares within a shareset are numbered by a small integer.

Cloud storage service A service such as Amazon S3 [2], Rackspace Cloud Files [3], Google Cloud Storage [4], or Windows Azure [5], that provides cloud storage.

Cloud storage interface A protocol interface supported by a cloud storage service, such as the S3 interface [6], the OpenStack Object Storage interface [7], the Google Cloud Storage interface [8], or the Azure interface [9]. There may be multiple services implementing a given cloud storage interface. In this design, only REST-based APIs [10] over HTTP will be used as interfaces.

Store object A file-like abstraction provided by a cloud storage service, storing a sequence of bytes. Store objects are mutable in the sense that the contents and metadata of the store object with a given name in a given backend store can be replaced. Store objects are called "blobs" in the Azure interface, and "objects" in the other interfaces.

Cloud backend store A container for store objects provided by a cloud service. Cloud backend stores are called "buckets" in the S3 and Google Cloud Storage interfaces, and "containers" in the Azure and OpenStack Storage interfaces.

Functional Requirements

- *Upload*: a LAFS share can be uploaded to an appropriately configured Tahoe-LAFS storage server and the data is stored to the cloud storage service.
- *Scalable shares*: there is no hard limit on the size of LAFS share that can be uploaded.

  If the cloud storage interface offers scalable files, then this could be implemented by using that feature of the specific cloud storage interface. Alternately, it could be implemented by mapping from the LAFS abstraction of an unlimited-size immutable share to a set of size-limited store objects.
- *Streaming upload*: the size of the LAFS share that is uploaded can exceed the amount of RAM and even the amount of direct attached storage on the storage server. I.e., the storage server is required to stream the data directly to the ultimate cloud storage service while processing it, instead of to buffer the data until the client is finished uploading and then transfer the data to the cloud storage service.
- *Download*: a LAFS share can be downloaded from an appropriately configured Tahoe-LAFS storage server, and the data is loaded from the cloud storage service.

- *Streaming download*: the size of the LAFS share that is downloaded can exceed the amount of RAM and even the amount of direct attached storage on the storage server. I.e. the storage server is required to stream the data directly to the client while processing it, instead of to buffer the data until the cloud storage service is finished serving and then transfer the data to the client.
- *Modify*: a LAFS share can have part of its contents modified.

  If the cloud storage interface offers scalable mutable files, then this could be implemented by using that feature of the specific cloud storage interface. Alternately, it could be implemented by mapping from the LAFS abstraction of an unlimited-size mutable share to a set of size-limited store objects.
- *Efficient modify*: the size of the LAFS share being modified can exceed the amount of RAM and even the amount of direct attached storage on the storage server. I.e. the storage server is required to download, patch, and upload only the segment(s) of the share that are being modified, instead of to download, patch, and upload the entire share.
- *Tracking leases*: The Tahoe-LAFS storage server is required to track when each share has its lease renewed so that unused shares (shares whose lease has not been renewed within a time limit, e.g. 30 days) can be garbage collected. This does not necessarily require code specific to each cloud storage interface, because the lease tracking can be performed in the storage server's generic component rather than in the component supporting each interface.

Mapping

This section describes the mapping between LAFS shares and store objects.

A LAFS share will be split into one or more "chunks" that are each stored in a store object. A LAFS share of size *C* bytes will be stored as *ceiling(C / chunksize)* chunks. The last chunk has a size between 1 and *chunksize* bytes inclusive. (It is not possible for *C* to be zero, because valid shares always have a header, so, there is at least one chunk for each share.)

For an existing share, the chunk size is determined by the size of the first chunk. For a new share, it is a parameter that may depend on the storage interface. It is an error for any chunk to be larger than the first chunk, or for any chunk other than the last to be smaller than the first chunk. If a mutable share with total size less than the default chunk size for the storage interface is being modified, the new contents are split using the default chunk size.

> *Rationale*: this design allows the *chunksize* parameter to be changed for new shares written via a particular storage interface, without breaking compatibility with existing stored shares. All cloud storage interfaces return the sizes of store objects with requests to list objects, and so the size of the first chunk can be determined without an additional request.

The name of the store object for chunk $i > 0$ of a LAFS share with storage index *STORAGEINDEX* and share number *SHNUM*, will be

> shares/*ST*/*STORAGEINDEX*/*SHNUM*.*i*

where *ST* is the first two characters of *STORAGEINDEX*. When *i* is 0, the *.0* is omitted.

> *Rationale*: this layout maintains compatibility with data stored by the prototype S3 backend, for which Least Authority Enterprises has existing customers. This prototype always used a single store object to store each share, with name
>
> > shares/*ST*/*STORAGEINDEX*/*SHNUM*
>
> By using the same prefix "shares/*ST*/*STORAGEINDEX*/" for old and new layouts, the storage server can obtain a list of store objects associated with a given shareset without having to know the layout in advance, and without having to make multiple API requests. This also simplifies sharing of test code between the disk and cloud backends.

Mutable and immutable shares will be "chunked" in the same way.

Rationale for Chunking

Limiting the amount of data received or sent in a single request has the following advantages:

- It is unnecessary to write separate code to take advantage of the "large object" features of each cloud storage interface, which differ significantly in their design.
- Data needed for each PUT request can be discarded after it completes. If a PUT request fails, it can be retried while only holding the data for that request in memory.

Costs

In this section we analyze the costs of the proposed design in terms of network, disk, memory, cloud storage, and API usage.

Network usage—bandwidth and number-of-round-trips

When a Tahoe-LAFS storage client allocates a new share on a storage server, the backend will request a list of the existing store objects with the appropriate prefix. This takes one HTTP request in the common case, but may take more for the S3 interface, which has a limit of 1000 objects returned in a single "GET Bucket" request.

If the share is to be read, the client will make a number of calls each specifying the offset and length of the required span of bytes. On the first request that overlaps a given chunk of the share, the server will make an HTTP GET request for that store object. The server may also speculatively make GET requests for store objects that are likely to be needed soon (which can be predicted since reads are normally sequential), in order to reduce latency.

Each read will be satisfied as soon as the corresponding data is available, without waiting for the rest of the chunk, in order to minimize read latency.

All four cloud storage interfaces support GET requests using the Range HTTP header. This could be used to optimize reads where the Tahoe-LAFS storage client requires only part of a share.

If the share is to be written, the server will make an HTTP PUT request for each chunk that has been completed. Tahoe-LAFS clients only write immutable shares sequentially, and so we can rely on that property to simplify the implementation.

When modifying shares of an existing mutable file, the storage server will be able to make PUT requests only for chunks that have changed. (Current Tahoe-LAFS v1.9 clients will not take advantage of this ability, but future versions will probably do so for MDMF files.)

In some cases, it may be necessary to retry a request (see the *Structure of Implementation* section below). In the case of a PUT request, at the point at which a retry is needed, the new chunk contents to be stored will still be in memory and so this is not problematic.

In the absence of retries, the maximum number of GET requests that will be made when downloading a file, or the maximum number of PUT requests when uploading or modifying a file, will be equal to the number of chunks in the file.

If the new mutable share content has fewer chunks than the old content, then the remaining store objects for old chunks must be deleted (using one HTTP request each). When reading a share, the backend must tolerate the case where these store objects have not been deleted successfully.

The last write to a share will be reported as successful only when all corresponding HTTP PUTs and DELETEs have completed successfully.

Disk usage (local to the storage server)

It is never necessary for the storage server to write the content of share chunks to local disk, either when they are read or when they are written. Each chunk is held only in memory.

A proposed change to the Tahoe-LAFS storage server implementation uses a sqlite database to store metadata about shares. In that case the same database would be used for the cloud backend. This would enable lease tracking to be implemented in the same way for disk and cloud backends.

Memory usage

The use of chunking simplifies bounding the memory usage of the storage server when handling files that may be larger than memory. However, this depends on limiting the number of chunks that are simultaneously held in memory. Multiple chunks can be held in memory either because of pipelining of requests for a single share, or because multiple shares are being read or written (possibly by multiple clients).

For immutable shares, the Tahoe-LAFS storage protocol requires the client to specify in advance the maximum amount of data it will write. Also, a cooperative client (including all existing released versions of the Tahoe-LAFS code) will limit the amount of data that is pipelined, currently to 50 KiB. Since the chunk size will be greater than that, it is possible to ensure that for each allocation, the maximum chunk data memory usage is the lesser of two chunks, and the allocation size. (There is some additional overhead but it is small compared to the chunk data.) If the maximum memory usage of a new allocation would exceed the memory available, the allocation can be delayed or possibly denied, so that the total memory usage is bounded.

It is not clear that the existing protocol allows allocations for mutable shares to be bounded in general; this may be addressed in a future protocol change.

The above discussion assumes that clients do not maliciously send large messages as a denial-of-service attack. Foolscap (the protocol layer underlying the Tahoe-LAFS storage protocol) does not attempt to resist denial of service.

Storage

The storage requirements, including not-yet-collected garbage shares, are the same as for the Tahoe-LAFS disk backend. That is, the total size of cloud objects stored is equal to the total size of shares that the disk backend would store.

Erasure coding causes the size of shares for each file to be a factor *shares.total* / *shares.needed* times the file size, plus overhead that is logarithmic in the file size [11].

API usage

Cloud storage backends typically charge a small fee per API request. The number of requests to the cloud storage service for various operations is discussed under "network usage" above.

Structure of Implementation

A generic "cloud backend", based on the prototype S3 backend but with support for chunking as described above, will be written.

An instance of the cloud backend can be attached to one of several "cloud interface adapters", one for each cloud storage interface. These adapters will operate only on chunks, and need not distinguish between mutable and immutable shares. They will be a relatively "thin" abstraction layer over the HTTP APIs of each cloud storage interface, similar to the S3Bucket abstraction in the prototype.

For some cloud storage services it may be necessary to transparently retry requests in order to recover from transient failures. (Although the erasure coding may enable a file to be retrieved even when shares are not stored by or not readable from all cloud storage services used in a Tahoe-LAFS grid, it may be desirable to retry cloud storage service requests in order to improve overall reliability.) Support for this will be implemented in the generic cloud backend, and used whenever a cloud storage adaptor reports a transient failure. Our experience with the prototype suggests that it is necessary to retry on transient failures for Amazon's S3 service.

There will also be a "mock" cloud interface adaptor, based on the prototype's MockS3Bucket. This allows tests of the generic cloud backend to be run without a connection to a real cloud service. The mock adaptor will be able to simulate transient and non-transient failures.

Known Issues

This design worsens a known "write hole" issue in Tahoe-LAFS when updating the contents of mutable files. An update to a mutable file can require changing the contents of multiple chunks, and if the client fails or is disconnected during the operation the resulting state of the store objects for that share may be inconsistent—no longer containing all of the old version, but not yet containing all of the new version. A mutable share can be left in an inconsistent state even by the existing Tahoe-LAFS disk backend if it fails during a write, but that has a smaller chance of occurrence because the current client behavior leads to mutable shares being written to disk in a single system call.

The best fix for this issue probably requires changing the Tahoe-LAFS storage protocol, perhaps by extending it to use a two-phase or three-phase commit (ticket #1755).

References

[1] omitted [2] "Amazon S3" Amazon (2012)

https://aws.amazon.com/s3/

[3] "Rackspace Cloud Files" Rackspace (2012)

https://www.rackspace.com/cloud/cloud_hosting_products/files/

[4] "Google Cloud Storage" Google (2012)

https://developers.google.com/storage/

[5] "Windows Azure Storage" Microsoft (2012)

https://www.windowsazure.com/en-us/develop/net/fundamentals/cloud-storage/

[6] "Amazon Simple Storage Service (Amazon S3) API Reference: REST API" Amazon (2012)

http://docs.amazonwebservices.com/AmazonS3/latest/API/APIRest.html

[7] "OpenStack Object Storage" openstack.org (2012)

http://openstack.org/projects/storage/

[8] "Google Cloud Storage Reference Guide" Google (2012)

https://developers.google.com/storage/docs/reference-guide

[9] "Windows Azure Storage Services REST API Reference" Microsoft (2012)

http://msdn.microsoft.com/en-us/library/windowsazure/dd179355.aspx

[10] "Representational state transfer" English Wikipedia (2012)

https://en.wikipedia.org/wiki/Representational_state_transfer

[11] "Performance costs for some common operations" tahoe-lafs.org (2012)

Performance costs for some common operations

CHAPTER 35

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.

Lease database design

The target audience for this document is developers who wish to understand the new lease database (leasedb) planned to be added in Tahoe-LAFS v1.11.0.

Introduction

A "lease" is a request by an account that a share not be deleted before a specified time. Each storage server stores leases in order to know which shares to spare from garbage collection.

Motivation

The leasedb will replace the current design in which leases are stored in the storage server's share container files. That design has several disadvantages:

- Updating a lease requires modifying a share container file (even for immutable shares). This complicates the implementation of share classes. The mixing of share contents and lease data in share files also led to a security bug (ticket #1528).
- When only the disk backend is supported, it is possible to read and update leases synchronously because the share files are stored locally to the storage server. For the cloud backend, accessing share files requires an HTTP request, and so must be asynchronous. Accepting this asynchrony for lease queries would be both inefficient and complex. Moving lease information out of shares and into a local database allows lease queries to stay synchronous.

Also, the current cryptographic protocol for renewing and cancelling leases (based on shared secrets derived from secure hash functions) is complex, and the cancellation part was never used.

The leasedb solves the first two problems by storing the lease information in a local database instead of in the share container files. The share data itself is still held in the share container file.

At the same time as implementing leasedb, we devised a simpler protocol for allocating and cancelling leases: a client can use a public key digital signature to authenticate access to a foolscap object representing the authority of an account. This protocol is not yet implemented; at the time of writing, only an "anonymous" account is supported.

The leasedb also provides an efficient way to get summarized information, such as total space usage of shares leased by an account, for accounting purposes.

Design constraints

A share is stored as a collection of objects. The persistent storage may be remote from the server (for example, cloud storage).

Writing to the persistent store objects is in general not an atomic operation. So the leasedb also keeps track of which shares are in an inconsistent state because they have been partly written. (This may change in future when we implement a protocol to improve atomicity of updates to mutable shares.)

Leases are no longer stored in shares. The same share format is used as before, but the lease slots are ignored, and are cleared when rewriting a mutable share. The new design also does not use lease renewal or cancel secrets. (They are accepted as parameters in the storage protocol interfaces for backward compatibility, but are ignored. Cancel secrets were already ignored due to the fix for #1528.)

The new design needs to be fail-safe in the sense that if the lease database is lost or corruption is detected, no share data will be lost (even though the metadata about leases held by particular accounts has been lost).

Accounting crawler

A "crawler" is a long-running process that visits share container files at a slow rate, so as not to overload the server by trying to visit all share container files one after another immediately.

The accounting crawler replaces the previous "lease crawler". It examines each share container file and compares it with the state of the leasedb, and may update the state of the share and/or the leasedb.

The accounting crawler may perform the following functions (but see ticket #1834 for a proposal to reduce the scope of its responsibility):

- Remove leases that are past their expiration time. (Currently, this is done automatically before deleting shares, but we plan to allow expiration to be performed separately for individual accounts in future.)
- Delete the objects containing unleased shares — that is, shares that have stable entries in the leasedb but no current leases (see below for the definition of "stable" entries).
- Discover shares that have been manually added to storage, via `scp` or some other out-of-band means.
- Discover shares that are present when a storage server is upgraded to a leasedb-supporting version from a previous version, and give them "starter leases".
- Recover from a situation where the leasedb is lost or detectably corrupted. This is handled in the same way as upgrading from a previous version.
- Detect shares that have unexpectedly disappeared from storage. The disappearance of a share is logged, and its entry and leases are removed from the leasedb.

Accounts

An account holds leases for some subset of shares stored by a server. The leasedb schema can handle many distinct accounts, but for the time being we create only two accounts: an anonymous account and a starter account. The starter account is used for leases on shares discovered by the accounting crawler; the anonymous account is used for all other leases.

The leasedb has at most one lease entry per account per (storage_index, shnum) pair. This entry stores the times when the lease was last renewed and when it is set to expire (if the expiration policy does not force it to expire earlier), represented as Unix UTC-seconds-since-epoch timestamps.

For more on expiration policy, see *Garbage Collection in Tahoe*.

Share states

The leasedb holds an explicit indicator of the state of each share.

The diagram and descriptions below give the possible values of the "state" indicator, what that value means, and transitions between states, for any (storage_index, shnum) pair on each server:

```
#         STATE_STABLE -------.
#          ^   |   ^ |         |
#          |   v   | |         v
#    STATE_COMING | |    STATE_GOING
#          ^      | |         |
#          |      | v         |
#          '----- NONE <------'
```

NONE: There is no entry in the `shares` table for this (storage_index, shnum) in this server's leasedb. This is the initial state.

STATE_COMING: The share is being created or (if a mutable share) updated. The store objects may have been at least partially written, but the storage server doesn't have confirmation that they have all been completely written.

STATE_STABLE: The store objects have been completely written and are not in the process of being modified or deleted by the storage server. (It could have been modified or deleted behind the back of the storage server, but if it has, the server has not noticed that yet.) The share may or may not be leased.

STATE_GOING: The share is being deleted.

State transitions

- **STATE_GOING → NONE**

  trigger: The storage server gains confidence that all store objects for the share have been removed.

  implementation:

  1. Remove the entry in the leasedb.

- **STATE_STABLE → NONE**

  trigger: The accounting crawler noticed that all the store objects for this share are gone.

  implementation:

  1. Remove the entry in the leasedb.

- **NONE → STATE_COMING**

triggers: A new share is being created, as explicitly signalled by a client invoking a creation command, *or* the accounting crawler discovers an incomplete share.

implementation:

1. Add an entry to the leasedb with **STATE_COMING**.
2. (In case of explicit creation) begin writing the store objects to hold the share.

- **STATE_STABLE → STATE_COMING**

  trigger: A mutable share is being modified, as explicitly signalled by a client invoking a modification command.

  implementation:

  1. Add an entry to the leasedb with **STATE_COMING**.
  2. Begin updating the store objects.

- **STATE_COMING → STATE_STABLE**

  trigger: All store objects have been written.

  implementation:

  1. Change the state value of this entry in the leasedb from **STATE_COMING** to **STATE_STABLE**.

- **NONE → STATE_STABLE**

  trigger: The accounting crawler discovers a complete share.

  implementation:

  1. Add an entry to the leasedb with **STATE_STABLE**.

- **STATE_STABLE → STATE_GOING**

  trigger: The share should be deleted because it is unleased.

  implementation:

  1. Change the state value of this entry in the leasedb from **STATE_STABLE** to **STATE_GOING**.
  2. Initiate removal of the store objects.

The following constraints are needed to avoid race conditions:

- While a share is being deleted (entry in **STATE_GOING**), we do not accept any requests to recreate it. That would result in add and delete requests for store objects being sent concurrently, with undefined results.
- While a share is being added or modified (entry in **STATE_COMING**), we treat it as leased.
- Creation or modification requests for a given mutable share are serialized.

Unresolved design issues

- What happens if a write to store objects for a new share fails permanently? If we delete the share entry, then the accounting crawler will eventually get to those store objects and see that their lengths are inconsistent with the length in the container header. This will cause the share to be treated as corrupted. Should we instead attempt to delete those objects immediately? If so, do we need a direct **STATE_COMING → STATE_GOING** transition to handle this case?

- What happens if only some store objects for a share disappear unexpectedly? This case is similar to only some objects having been written when we get an unrecoverable error during creation of a share, but perhaps we want to treat it differently in order to preserve information about the storage service having lost data.
- Does the leasedb need to track corrupted shares?

Future directions

Clients will have key pairs identifying accounts, and will be able to add leases for a specific account. Various space usage policies can be defined.

Better migration tools ('tahoe storage export'?) will create export files that include both the share data and the lease data, and then an import tool will both put the share in the right place and update the recipient node's leasedb.

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.

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 *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 not equal to 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 not equal to 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:
  - 1. 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.

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
  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.

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:

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.

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.

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.

CHAPTER 36

Filesystem-specific notes

1. *ext3*

Tahoe storage servers use a large number of subdirectories to store their shares on local disk. This format is simple and robust, but depends upon the local filesystem to provide fast access to those directories.

ext3

For moderate- or large-sized storage servers, you'll want to make sure the "directory index" feature is enabled on your ext3 directories, otherwise share lookup may be very slow. Recent versions of ext3 enable this automatically, but older filesystems may not have it enabled:

```
$ sudo tune2fs -l /dev/sda1 |grep feature
Filesystem features:      has_journal ext_attr resize_inode dir_index filetype needs_
↪recovery sparse_super large_file
```

If "dir_index" is present in the "features:" line, then you're all set. If not, you'll need to use tune2fs and e2fsck to enable and build the index. See http://wiki2.dovecot.org/MailboxFormat/Maildir for some hints.

CHAPTER 37

Old Configuration Files

Tahoe-LAFS releases before v1.3.0 had no `tahoe.cfg` file, and used distinct files for each item listed below. If Tahoe-LAFS v1.9.0 or above detects the old configuration files at start up it emits a warning and aborts the start up. (This was issue ticket #1385.)

Config setting	File	Comment
[node]nickname	BASEDIR/nickname	
[node]web.port	BASEDIR/webport	
[node]tub.port	BASEDIR/client.port	(for Clients, not Introducers)
[node]tub.port	BASEDIR/introducer.port	(for Introducers, not Clients) (note that, unlike other keys, tahoe.cfg overrode this file from Tahoe-LAFS v1.3.0 up to and including Tahoe-LAFS v1.8.2)
[node]tub.location	BASEDIR/advertised_ip_addresses	
[node]log_gatherer.furl	BASEDIR/log_gatherer.furl	(one per line)
[node]timeout.keepalive	BASEDIR/keepalive_timeout	
[node]timeout.disconnect	BASEDIR/disconnect_timeout	
[client]introducer.furl	BASEDIR/introducer.furl	
[client]helper.furl	BASEDIR/helper.furl	
[client]key_generator.furl	BASEDIR/key_generator.furl	
[client]stats_gatherer.furl	BASEDIR/stats_gatherer.furl	
[storage]enabled	BASEDIR/no_storage	(False if no_storage exists)
[storage]readonly	BASEDIR/readonly_storage	(True if readonly_storage exists)
[storage]sizelimit	BASEDIR/sizelimit	
[storage]debug_discard	BASEDIR/debug_discard_storage	
[helper]enabled	BASEDIR/run_helper	(True if run_helper exists)

Note: the functionality of [node]ssh.port and [node]ssh.authorized_keys_file were previously (before Tahoe-LAFS v1.3.0) combined, controlled by the presence of a BASEDIR/authorized_keys.SSHPORT file, in which the suffix of the filename indicated which port the ssh server should listen on, and the contents of the file provided the ssh public keys to accept. Support for these files has been removed completely. To ssh into your Tahoe-LAFS node, add [node]ssh.port and [node].ssh_authorized_keys_file statements to your tahoe.cfg.

Likewise, the functionality of [node]tub.location is a variant of the now (since Tahoe-LAFS v1.3.0) unsupported BASEDIR/advertised_ip_addresses . The old file was additive (the addresses specified in advertised_ip_addresses were used in addition to any that were automatically discovered), whereas the new tahoe.cfg directive is not (tub.location is used verbatim).

CHAPTER 38

Using Tahoe as a key-value store

There are several ways you could use Tahoe-LAFS as a key-value store.

Looking only at things that are *already implemented*, there are three options:

1. Immutable files

   API:

   - key ← put(value)

     This is spelled "PUT /uri" in the API.

     Note: the user (client code) of this API does not get to choose the key! The key is determined programmatically using secure hash functions and encryption of the value and of the optional "added convergence secret".

   - value ← get(key)

     This is spelled "GET /uri/$FILECAP" in the API. "$FILECAP" is the key.

   For details, see "immutable files" in *Performance costs for some common operations*, but in summary: the performance is not great but not bad.

   That document doesn't mention that if the size of the A-byte mutable file is less than or equal to 55 bytes then the performance cost is much smaller, because the value gets packed into the key. Added a ticket: #2226.

2. Mutable files

   API:

   - key ← create()

     This is spelled "PUT /uri?format=mdmf".

     Note: again, the key cannot be chosen by the user! The key is determined programmatically using secure hash functions and RSA public key pair generation.

   - set(key, value)

- value ← get(key)

  This is spelled "GET /uri/$FILECAP". Again, the "$FILECAP" is the key. This is the same API as for getting the value from an immutable, above. Whether the value you get this way is immutable (i.e. it will always be the same value) or mutable (i.e. an authorized person can change what value you get when you read) depends on the type of the key.

Again, for details, see "mutable files" in *Performance costs for some common operations* (and these tickets about how that doc is incomplete), but in summary, the performance of the create() operation is *terrible*! (It involves generating a 2048-bit RSA key pair.) The performance of the set and get operations are probably merely not great but not bad.

3. Directories

   API:

   - directory ← create()

     This is spelled "POST /uri?t=mkdir".

     Performance costs for some common operations does not mention directories (#2228), but in order to understand the performance of directories you have to understand how they are implemented. Mkdir creates a new mutable file, exactly the same, and with exactly the same performance, as the "create() mutable" above.

   - set(directory, key, value)

     This is spelled "PUT /uri/$DIRCAP/[SUBDIRS../]FILENAME". "$DIRCAP" is the directory, "FILENAME" is the key. The value is the body of the HTTP PUT request. The part about "[SUBDIRS../]" in there is for optional nesting which you can ignore for the purposes of this key-value store.

     This way, you *do* get to choose the key to be whatever you want (an arbitrary unicode string).

     To understand the performance of `PUT /uri/$directory/$key`, understand that this proceeds in two steps: first it uploads the value as an immutable file, exactly the same as the "put(value)" API from the immutable API above. So right there you've already paid exactly the same cost as if you had used that API. Then after it has finished uploading that, and it has the immutable file cap from that operation in hand, it downloads the entire current directory, changes it to include the mapping from key to the immutable file cap, and re-uploads the entire directory. So that has a cost which is easy to understand: you have to download and re-upload the entire directory, which is the entire set of mappings from user-chosen keys (Unicode strings) to immutable file caps. Each entry in the directory occupies something on the order of 300 bytes.

     So the "set()" call from this directory-based API has obviously much worse performance than the the equivalent "set()" calls from the immutable-file-based API or the mutable-file-based API. This is not necessarily worse overall than the performance of the mutable-file-based API if you take into account the cost of the necessary create() calls.

   - value ← get(directory, key)

     This is spelled "GET /uri/$DIRCAP/[SUBDIRS../]FILENAME". As above, "$DIRCAP" is the directory, "FILENAME" is the key.

     The performance of this is determined by the fact that it first downloads the entire directory, then finds the immutable filecap for the given key, then does a GET on that immutable filecap. So again, it is strictly worse than using the immutable file API (about twice as bad, if the directory size is similar to the value size).

What about ways to use LAFS as a key-value store that are not yet implemented? Well, Zooko has lots of ideas about ways to extend Tahoe-LAFS to support different kinds of storage APIs or better performance. One that he thinks is pretty promising is just the Keep It Simple, Stupid idea of "store a sqlite db in a Tahoe-LAFS mutable".

CHAPTER 39

Indices and tables

- genindex
- modindex
- search