guld

guld FileSystem (guldFS) Specification

Author: Ira Miller <public@iramiller.com>

DRAFT
v0.0.1

Overview

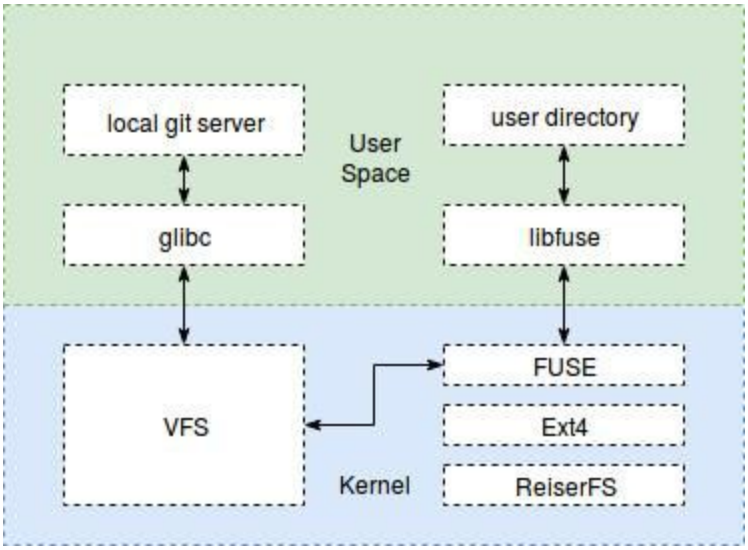
The Guld FileSystem (guldFS) is a distributed, signed, encryptable, and version controlled filesystem in user space. GuldFS is designed to run in Linux using stable, open source components like git, GnuPG, bittorrent, and FUSE.

FUSE

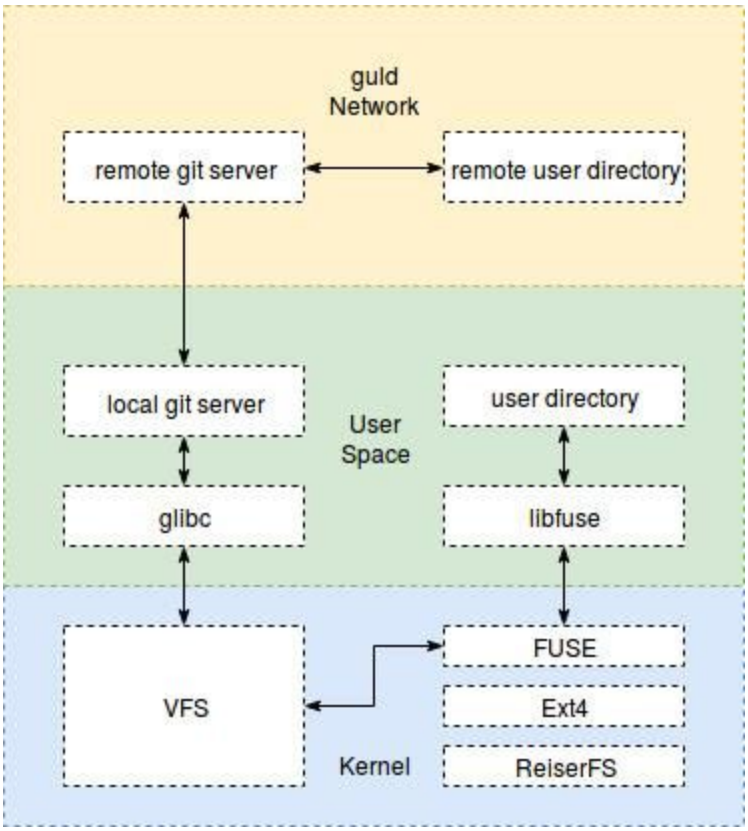
GuldFS uses Filesystem in Userspace (FUSE) to create a native filesystem experience from the kernel level up.

Local users mount directories using fuse, and make changes like they would edit any other file. The changes are then passed through libfuse to the kernel, and then back out into the local git data directory.

The local git data directory should be kept in a “clean” state, in the fashion of a git server. Also following git server convention, this data should be managed by a system user named `git`.



Local Git Server



The git server can act only as a local backup of user data, but it can also be configured to synchronize with third party git repositories. To manage the permissions, gitolite is the recommended server.[1]

Gitolite can be configured to allow limited secure shell (SSH) sessions by other users identified by their usernames and corresponding SSH keys.

For example, a group of users `family` could be given read-only access to all subdirectories of your pictures, but not to your work documents. Anyone authenticating themself with an SSH key known to be in the `family` group would be allowed to download a picture from you.

These remote users could be using plain old git repositories, or could also be running guldFS, allowing them to mount your pictures directories on their local system.

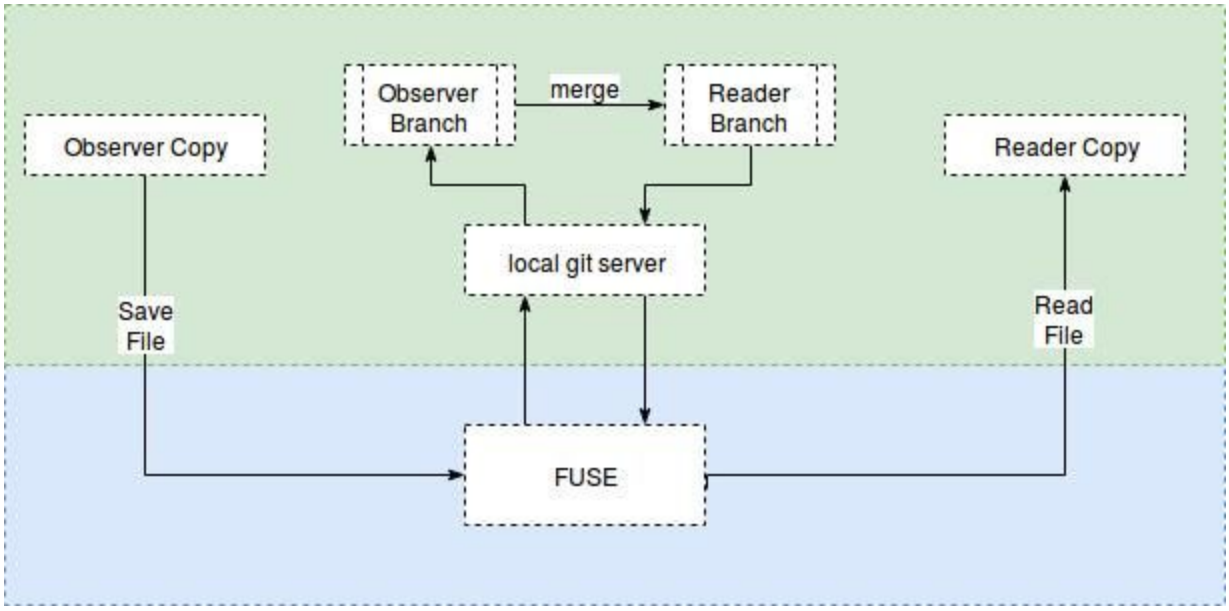
Branches and Merging

Each user will have a named branch for each git repository, so there should never be any ambiguity about which branch to checkout or commit to. The middleware can always commit to and pull from branch `isysd` for user `isysd`.

Each repository should have a designated Observer, who governs the contents.[2] When the Observer merges a commit, the rule is for all user branches to also merge. This process should be managed for all users and all repositories by a service running under the `git` user. Since

[1] http://gitolite.com/
[2] Consensus mechanism as described in the guld whitepaper (2017) by Ira Miller

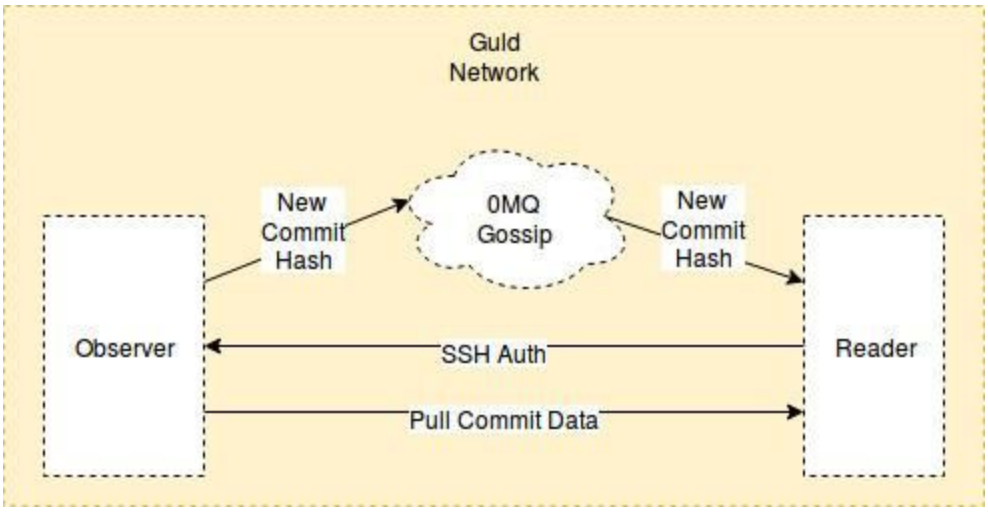
only merges are to be performed by this service, it does not need an identity or signing key, and only needs write access to the clean copy of each repository.



The Observer could be another user on the same local system, or it could be a remote user identified by username and SSH key. All that is required for consensus, and for merging, is that the official Observer key(s) have signed the Observer branch. Once the threshold of signatures is met, the commit(s) should be merged into all Reader branches.

Guld Network

The guld network is peer to peer and split into two halves: messaging (metadata) and SSH (data). The messaging portion is a pub/sub 0MQ network, where each network node publishes known commit hashes. The SSH portion of the network is for retrieving and pushing the contents of each commit.



The contents of the 0MQ messages should consist of `$user:$repository:$commit-hash`. Because the contents are not sent, and not recoverable based on the hash, these messages are safe to distribute all across the network in a `gossip` pool. The term gossip is used because the messages are not necessarily observed by the sending node, and neither node may be able to access the full contents that the message refers to. Should an interested and authorized user hear about the message however, he need only open an SSH connection to the user in the message, and request to pull down the commit hash.

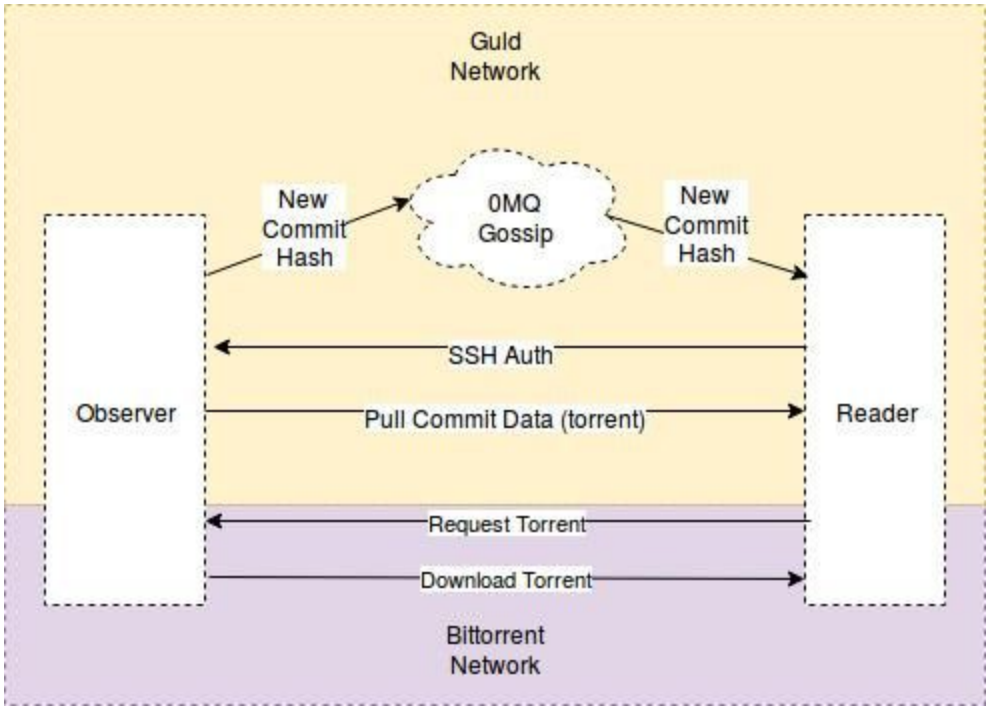
Configuration

Each repository should be configured using a version controlled file. The recommended name for this file is `.gap.json`. The file must, at minimum, describe the path to the repository, and it's official Observer.

``` .gap.json example
{
   "path": "/isysd/pictures/family/",
   "observer": "isysd"
}
```

This lets other users know the governance for the repository, and it's address. This address can be used in local file structures (i.e. mounted at /home/isysd/pictures/family) but that is not required. It is, however, required to name the repository after this path in kebab case (i.e. isysd-pictures-family), and to use the path in all related network messages.

Optional Configuration



For additional security, and/or for large file support, it is recommended to use pre and post commit hooks to transform the commit contents. For example, the pre-commit hook might do something like

encrypt the files using another user's PGP key, create a torrent out of the encrypted files, and then commit the torrent to git, ignoring both the plaintext and encrypted files. The reader would have a post-commit hook that would do the reverse, opening the torrent, downloading the encrypted files, then decrypting them to finally access the plaintext file.

Because git hooks are arbitrarily programmable, it is not necessary to describe every possible use case. The important thing is that each repository declare pre and post commit hooks in .gap.json, so that all members of that repository follow the same rules.

``` .gap.json example w/ hooks
{
   “path”: “/isysd/pictures/family/”,
   “observer”: “isysd”,
   “pre-commit”: “path/to/pre-commit.sh”,
   “post-commit”: “path/to/post-commit.sh”
}
```