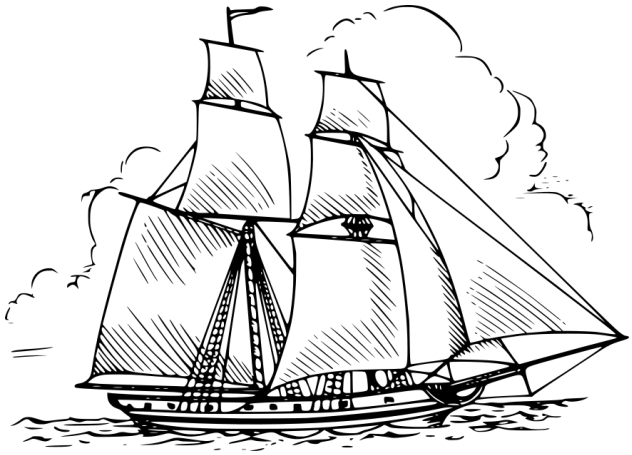
brig Documentation

Release v0.1.0-beta

Chris Pahl

Dec 16, 2018

User documentation:



`brig` is a distributed & secure file synchronization tool with version control. It is based on `ipfs`, written in Go and will feel familiar to `git` users.

Key feature highlights:

- Encryption of data in rest and transport, plus optional compression on the fly.
- Simplified `git` version control only limited by your storage space.
- Synchronization algorithm that can handle moved files and empty directories and files.
- Your data does not need to be stored on the device you are currently using.
- FUSE filesystem that feels like a normal sync folder.
- No central server at all. Still, central architectures can be build with `brig`.
- Simple user identification and discovery.
- Completely free software under the terms of the `AGPL`.

`brig` tries to focus on being up conceptually simple, by hiding a lot of complicated details regarding storage and security. Therefore the end result is hopefully easy and pleasant to use, while being secure by default. Since `brig` is a »general purpose« tool for file synchronization it of course cannot excel in all areas. It won't replace high performance network file systems.

CHAPTER 1

Current Status

This software is in active development and probably not suited for production use yet! But to get it in a stable state, it is **essential** that people play around with it. Consider this is as an open beta phase. Also don't take anything granted for now, everything might change wildly before version `1.0.0`.

With that being said, `brig` is near a somewhat usable state where you can play around with it quite well. All aforementioned features do work, besides possibly being a little harder to use than ideally possible. A lot of work is currently going into stabilizing the current feature set.

At this moment `brig` is **only tested on Linux**. Porting and testing efforts are welcome. The only incompatible feature so far is *FUSE* which would be needed to either disabled or replaced on other platforms.

1.1 Installation

At the time of writing, there are no pre-compiled binaries. So you gonna have to compile `brig` yourself - but don't worry that is quite easy. We do not have many dependencies, you only need two things: The programming language *Go* and the version control system `git`.

1.1.1 Step 0: Installing Go

This is only required if you don't already have `Go` installed. Please consult your package manager for that.

> **Warning:** `brig` only works with a newer version of Go (>= 1.9). The version in your package manager might be too outdated, if you're on e.g. Debian. Make sure it's rather up to date!

If you did not do that, you gonna need to install `Go`. Refere here for possible ways of doing so. Remember to set the `GOPATH` environment variable to a place where you'd like to have your `.go` sources being placed. For example you can put this in your `.bashrc`:

```
# Place the go sources in a "go" directory inside your home directory:
export GOPATH=~/go
# This is needed for the go toolchain:
export GOBIN="$GOPATH/bin"
# Make sure that our shell finds the go binaries:
export PATH="$GOPATH/bin:$PATH"
```

By choosing to have the `GOPATH` in your home directory you're not required to have `sudo` permissions later on. You also need to have `git` installed for the next step.

1.1.2 Step 1: Compile & Install `brig`

This step requires setting `GOPATH`, as discussed in the previous section.

```
$ go get -d -v -u github.com/sahib/brig  # Download the sources.
$ cd $GOPATH/src/github.com/sahib/brig   # Go to the source directory.
$ make                                   # Build the software.
$ sudo make install                      # Install it system-wide (optional)
```

All dependencies of brig are downloaded for you during the first step. Execution might take a few minutes though because all of `brig` is being compiled during the `make` step.

If you cannot or want to install `git` for some reason, you can manually download a zip from GitHub and place its contents into `$GOPATH/src/github.com/sahib/brig`. In this case, you can skip the `go get` step.

1.1.3 Step 2: Test if the installation is working

```
$ brig help
```

If above command prints out documentation on how to use the program's commandline switches then the installation worked. Happy shipping!

Continue with *Getting started* or directly go to *Quickstart* if you just need a refresh on the details.

1.2 Getting started

This guide will walk you through the steps of synchronizing your first files over `brig`. You will learn about the concepts behind it along the way. Everything is hands on, so make sure to open a terminal. For now, `brig` has no other user interface.

1.2.1 Precursor: The help system

`brig` has some built-in helpers to serve as support for your memory. Before you dive into the actual commands, you should learn how to get help.

Built-in documentation

Every command offers detailed built-in help, which you can view using the `brig help` command:

```
$ brig help stage
NAME:
   brig stage - Add a local file to the storage

USAGE:
   brig stage [command options] (<local-path> [<path>]|--stdin <path>)

CATEGORY:
   WORKING TREE COMMANDS

DESCRIPTION:
   Read a local file (given by »local-path«) and try to read
   it. This is the conceptual equivalent of »git add«. [...]

EXAMPLES:

   $ brig stage file.png                          # gets added as /file.png
   $ brig stage file.png /photos/me.png           # gets added as /photos/me.png
   $ cat file.png | brig stage --stdin /file.png # gets added as /file.png

OPTIONS:
   --stdin, -i  Read data from stdin
```

Shell autocompletion

If you don't like to remember the exact name of each command, you can use the provided autocompletion. For this to work you have to insert this at the end of your `.bashrc`:

```
source $GOPATH/src/github.com/sahib/brig/autocomplete/bash_autocomplete
```

Or if you happen to use `zsh`, append this to your `.zshrc`:

```
source $GOPATH/src/github.com/sahib/brig/autocomplete/zsh_autocomplete
```

After starting a new shell you should be able to autocomplete most commands. Try this for example by typing `brig remote <tab>`.

Open the online documentation

By typing `brig docs` you'll get a tab opened in your browser with this domain loaded.

Reporting bugs

If you need to report a bug you can use a built-in utility to do that. It will gather all relevant information, create a report and open a tab with the *GitHub* issue tracker in a browser for you. Only thing left for you is to fill out some questions in the report and include anything you think is relevant.

```
$ brig bug
```

To actually create the issue you sadly need an *GitHub* account.

1.2.2 Creating a repository

Let's get started with the actual working commands.

You need a central place where `brig` stores files you give it. This place is called a »repository« or short »repo«. Think of it as a database where all files (and some metadata about them) is **copied** to. It is important to keep in mind that `brig` **copies** the file and does not do anything destructive to the original file.

By creating a new repository you also generate your identity, under which your buddies can later **find** and **authenticate** you.

But enough of the mere theory, let's get started:

```
# Create a place where we store our metadata.
$ mkdir ~/sync
$ brig --repo ~/sync init ali@woods.org/desktop
27.12.2017/14:44:39  Starting daemon from: /home/sahib/go/bin/brig
  39 New passphrase:

Well done! Please re-type your password now:
  39 Retype passphrase:

      _____           /  /\         ___           /  /\
     /  /::\         /  /::\       /  /\         /  /:/_
    /  /:/\:\       /  /:/\:\     /  /:/        /  /:/ /\
   /  /:/~/::\     /  /:/~/:/    /__/::\       /  /:/_/::\
  /__/:/ /:/\:| /__/:/ /:/___ \__\/\:\__   /__/:/__\/\:\
  \  \:\/:/~/:/ \  \:\/:::::/     \  \:\/\ \  \:\ /~~/:/
   \  \::/ /:/   \  \::/~~~~       \__\::/  \  \:\  /:/
    \  \:\/:/     \  \:\           /__/:/    \  \:\/:/
     \  \::/       \  \:\          \__\/      \  \::/
      \__\/         \__\/                     \__\/


    A new file README.md was automatically added.
    Use 'brig cat README.md' to view it & get started.
$ cd ~/sync
$ ls
config.yml  data  gpg.prv  gpg.pub  logs  metadata
meta.yml  passwd.locked  remotes.yml
```

The name you specified after the `init` is the name that will be shown to other users and by which you are searchable in the network. See *Choosing and finding names* for more details on the subject.

Once the `init` ran successfully there will be a daemon process running the background. Every other `brig` commands will communicate with it via a local network socket.

Also note that a lot of files were created in the current directory. This is all part of the metadata that is being used by the daemon that runs in the background. Please try not to modify them.

Passwords

You will be asked to enter a new password. The more secure the password is you entered, the greener the prompt gets[1]. This password is used to store the metadata in an encrypted manner on your filesystem and without further configuration it needs to be re-entered every time you start the daemon. There are two ways to prevent that:

[1] This uses Dropbox's password strength library »zxcvbn«.

1. Use a password helper and tell `brig` how to get a password from it by using `-w / --password-helper` on the `init` command. We recommend using pass to do that:

```
# Generate a password and store it in "pass":
$ pass generate brig/ali -n 20
# Tell brig how to get the password out of "pass":
# (This is an alternative to the way shown above,
#  if you try it out we first need to shut down the previous daemon)
$ brig daemon quit
$ brig --repo ~/sync-with-password init -w "pass brig/ali" ali@woods.org/desktop
# Now pass will ask you for the master password with
# a nice dialog whenever one if its passwords is first used.
```

   If you like this setup for an existing repo, take a look at the *Configuration* section.

2. Do not use a password. You can do this by passing `-x` to the `init` command. This is obviously not recommended.

Note: Using a good password is especially important if you're planning to move the repo, i.e. carrying it around you on a usb stick. When the daemon shuts down it locks and encrypts all files in the repository (including all metadata and keys), so nobodoy is able to access them anymore.

1.2.3 Adding & Viewing files

Now let's add some files to `brig`. We do this by using `brig stage`. It's called `stage` because all files first get added to a staging area. If you want, and are able to remember that easier, you can also use `brig add`.

```
$ echo "Hello World" > /tmp/hello.world
$ brig stage /tmp/hello.world
$ brig cat hello.world
Hello World
$ brig ls
SIZE   MODTIME          PATH          PIN
443 B  Dec 27 14:44:44  /README.md
12 B   Dec 27 15:14:16  /hello.world
```

This adds the content of `/tmp/hello.world` to a new file in `brig` called `/hello.world`. The name was automatically chosen from looking at the base name of the added file. All files in `brig` have their own name, possibly differing from the content of the file they originally came from. Of course, you can also add whole directories.

If you want to use a different name, you can simply pass the new name as second argument to `stage`:

```
$ brig stage /tmp/hello.world /hallo.welt
```

You also previously saw `brig cat` which can be used to get the content of a file again. `brig ls` in contrast shows you a list of currently existing files, including their size, last modification time, path and pin state[2].

One useful feature of `brig cat` is that you can output directories as well. When specifying a directory as path, a `.tar` archive is being outputted. You can use that easily to store whole directories on your disk or archive in order to send it to some client for example:

```
# Create a tar from root and unpack it to the current directory.
$ brig cat | tar xfv -
```

(continues on next page)

[2] Pinning and pin states are explained *Pinning* and are not important for now.

(continued from previous page)

```
# Create .tar.gz out of of the /photos directory.
$ brig cat photos | gzip -f > photos.tar.gz
```

1.2.4 Coreutils

You probably already noticed that a lot of commands you'd type in a terminal on a normal day have a sibling as `brig` command. Here is a short overview of the available commands:

```
$ brig mkdir photos
$ brig touch photos/me.png
$ brig tree
    .
├──photos
│   └── me.png
├── README.md
└── hello.world

2 directories, 2 files
$ brig cp photos/me.png photos/moi.png
$ brig mv photos/me.png photos/ich.png
# NOTE: There is no "-r" switch. Directories are always deleted recursively.
$ brig rm photos
```

Please refer to `brig help <command>` for more information about those. Sometimes they work a little bit different[3] and a bit less surprising than their counterparts. Also note that there is no `brig cd` currently. All paths must be absolute.

1.2.5 Mounting repositories

Using those specialized `brig` commands might not feel very seamless, especially when being used to tools like file browsers. And indeed, those commands are only supposed to serve as a low-level way of interacting with `brig` and as means for scripting own workflows.

For your daily workflow it is far easier to mount all files known to `brig` to a directory of your choice and use it with your normal tools. To accomplish that `brig` supports a FUSE filesystem that can be controlled via the `mount` and `fstab` commands. Let's look at `brig mount`:

```
$ mkdir ~/data
$ brig mount ~/data
$ cd ~/data
$ cat hello-world
Hello World
$ echo 'Salut le monde!' > salut-monde.txt
# There is no difference between brig's "virtual view"
# and the contents of the mount:
$ brig cat salut-monde.txt
Salut le monde!
```

You can use this directory exactly[4] like a normal one. You can have any number of mounts. This proves especially useful when only mounting a subdirectory of `brig` (let's say `Public`) with the `--root` option of `brig mount` and mounting all other files as read only (`--readonly`).

[3] `brig rm` for example deletes directories without needing a `-r` switch.

[4] Well almost. See the *Caveats* below.

```
$ brig mount ~/data --readonly
$ brig mkdir /writable
$ brig touch /writable/please-edit-me
$ mkdir ~/rw-data
$ brig mount ~/rw-data --root /writable
$ echo 'writable?' > ~/data/test
read-only file system: ~/data/test
$ echo 'writable!' > ~/rw-data/test
$ cat ~/rw-data/test
writable!
```

An existing mount can be removed again with `brig unmount <path>`:

```
$ brig unmount ~/data
$ brig unmount ~/rw-data
$ brig rm writable
```

Making mounts permanent

It can get a little annoying when having to manage several mounts yourself. It would be nice to have some *typical* mounts you'd like to have always and it should be only one command to mount or unmount all of them, kind of what `mount -a` does. That's what `brig fstab` is for:

```
$ brig fstab add tmp_rw_mount /tmp/rw-mount
$ brig fstab add tmp_ro_mount /tmp/ro-mount -r
$ brig fstab
NAME          PATH           READ_ONLY  ROOT  ACTIVE
tmp_ro_mount  /tmp/ro-mount  yes        /
tmp_rw_mount  /tmp/rw-mount  no         /
$ brig fstab apply
$ brig fstab
NAME          PATH           READ_ONLY  ROOT  ACTIVE
tmp_ro_mount  /tmp/ro-mount  yes        /
tmp_rw_mount  /tmp/rw-mount  no         /
$ brig fstab apply -u
NAME          PATH           READ_ONLY  ROOT  ACTIVE
tmp_ro_mount  /tmp/ro-mount  yes        /
tmp_rw_mount  /tmp/rw-mount  no         /
```

Et Voilà, all mounts will be created and mounted once you enter `brig fstab apply`. The opposite can be achieved by executing `brig fstab apply --unmount`. On every restart of the daemon, all mounts are mounted by default, so the only thing you need to make sure is that the daemon is running.

Caveats: The FUSE filesystem is not yet perfect. Keep those points in mind:

- **Performance:** Writing to FUSE is currently somewhat *memory and CPU intensive*. Generally, reading should be fast enough for most basic use cases, but also is not enough for high performance needs. If you need to edit a file many times, it is recommended to copy the file somewhere to your local storage (e.g. `brig cat the_file > /tmp/the_file`), edit it there and save it back for syncing purpose. Future releases will work on optimizing the performance.
- **Timeouts:** Although it tries not to look like one, we're operating on a networking filesystem. Every file you access might come from a different computer. If no other machine can serve this file we might block for a long time, causing application hangs and general slowness. This is a problem that still needs a proper solution and leaves much to be desired in the current implementation.

1.2.6 Remotes

Until now, all our operations were tied only to our local computer. But `brig` is a synchronization tool and that would be hardly very useful without supporting other peers. We call other peers »remotes« similar to the slang used in the `git` world.

Every remote possesses two things that identifies him:

- **A human readable name:** This name can be choose by the user and can take pretty much any form, but we recommend to sticking for a form that resembles an extended email[5] like »ali@woods.org/desktop«. This name is **not** guaranteed to be unique! In theory everyone could take it and it is therefore only used for display purposes. There is no central place where users are registered.
- **A unique fingerprint:** This serves both as address for a certain repository and as certificate of identity. It is long and hard to remember, which is the reason why `brig` offers to loosely link a human readable to it.

If we want to find out what our name and fingerprint is, we can use the `brig whoami` command to ask a very existential questions:

```
# NOTE: The hash will look different for you:
$ brig whoami
ali@woods.org/desktop␣
↪QmTTJbkfG267gidFKfDTV4j1c843z4tkUG93Hw8r6kZ17a:W1nayTG5UMcVxy9mFFNjuZDUb7uVTnmwFYiJ4Ajr1TP3bg
```

Note: The fingerprint consists of two hashes divided by a colon (:). The first part is the identity of your `IPFS` node, the second part is the fingerprint of a keypair that was generated by `brig` during init and will be used to authenticate other peers.

When we want to synchronize with another repository, we need to exchange fingerprints and each other as remote. There are three typical scenarios here:

1. Both repositories are controlled by you. In this case you can simple execute `brig whoami` on both repositories and add them with `brig remote add` as described in the following.
2. You want to sync with somebody you know. In this case you should both execute `brig whoami` and send its output over a trusted side channel. Personally, I use a secure messenger like Signal, but you can also use any channel you like, including encrypted mail or meeting up with the person in question.
3. You don't know each other. Get to know each other and the proceed like in the second point. If you need to get a hint of what users use a certain domain, you can use `brig net locate` to get a list of those:

   ```
   # This command might take some time to yield results:
   $ brig net locate -m domain woods.org
   NAME           TYPE    FINGERPRINT
   ali@woods.org  domain  QmTTJbk[...]:W1UDvKzjRPb4rbbk[...]
   ```

   Please note again: Do not blindly add the fingerprint you see here. Always make sure the person you're syncing with is the one you think they are.

   Todo: This seems currently broken as it does not yield any results.

Once you have exchanged the fingerprints, you add each other as **remotes**. Let's call the other side *bob*:[6]

[5] To be more exact, it resembles an XMPP or Jabber-ID.

[6] The name you choose as remote can be anything you like and does not need to match the name the other person chose for themselves. It's not a bad idea though.

```
$ brig remote add bob \
        QmUDSXt27LbCCG7NfNXfnwUkqwCig8RzV1wzB9ekdXaag7:
        W1e3rNGGCuuQnzyoiBKLdoN41yQ4NfNy9nRD3MwXk6h8Vy
```

Bob has do the same on his side. Otherwise the connection won't be established, because the other side won't be authenticated. By adding somebody as remote we **authenticate** them:

```
$ brig remote add ali \
QmTTJbkfG267gidFKfDTV4j1c843z4tkUG93Hw8r6kZ17a:
W1nayTG5UMcVxy9mFFNjuZDUb7uVTnmwFYiJ4Ajr1TP3bg
```

Thanks to the fingerprint, `brig` now knows how to reach the other repository over the network. This is done in the background via `IPFS` and might take a few moments until a valid route to the host was found.

The remote list can tell us if a remote is online:

```
$ brig remote list
NAME   FINGERPRINT  ROUNDTRIP  ONLINE AUTHENTICATED LASTSEEN
bob    QmUDSXt27    0s                            Apr 16 17:31:01
$ brig remote ping bob
ping to bob:  (0.00250s)
```

Nice. Now we know that bob is online () and also that he authenticated us (). Otherwise `brig remote ping bob` would have failed.

Note: About open ports:

While `ipfs` tries to do it's best to avoid having the user to open ports in his firewall/router. This mechanism might not be perfect though and maybe never is. If any of the following network operations might not work it might be necessary to open the port 4001 and/or enable UPnP. For security reasons we recommend to only open the required ports explicitly and not to use UPnP unless necessary. This is only necessary if the computers you're using `brig` on are not in the same local network.

Choosing and finding names

You might wonder what the name you pass to `init` is actually for. As previously noted, there is no real restriction for choosing a name, so all of the following are indeed valid names:

- `ali`
- `ali@woods.org`
- `ali@woods.org/desktop`
- `ali/desktop`

It's however recommended to choose a name that is formatted like a XMPP/Jabber-ID. Those IDs can look like plain emails, but can optionally have a »resource« part as suffix (separated by a »/« like `desktop`). Choosing such a name has two advantages:

- Other peers can find you by only specifying parts of your name. Imagine all of the *Smith* family members use `brig`, then they'd possibly those names:
  - `dad@smith.org/desktop`
  - `mom@smith.org/tablet`
  - `son@smith.org/laptop`

When `dad` now sets up `brig` on his server, he can use `brig net locate -m domain 'smith.org'` to get all fingerprints of all family members. Note however that `brig net locate` **is not secure**. Its purpose is solely discovery, but is not able to verify that the fingerprints really correspond to the persons they claim to be. This due to the distributed nature of `brig` where there is no central or federated authority that coordinate user name registrations. So it is perfectly possible that one name can be taken by several repositories - only the fingerprint is unique.

- Later development of `brig` might interpret the user name and domain as email and might use your email account for verification purposes.

Having a resource part is optional, but can help if you have several instances of `brig` on your machines. i.e. one username could be `dad@smith.org/desktop` and the other `dad@smith.org/server`.

1.2.7 Syncing

Before we move on to do our first synchronization, let's do a quick recap of what we have done so far:

- Create a repository (`brig init <name>`) - This needs to be done only once.
- Create optional mount points (`brig fstab add <name> <path>`) - This needs to be done only once.
- Find & add remotes (`brig remote add`) - This needs to be done once for each peer.
- Add some files (`brig stage <path>`) - Do as often as you like.

As you can see, there is some initial setup work, but the actual syncing is pretty effortless now. Before we attempt to sync with anybody, it's always a good idea to see what changes they have. We can check this with `brig diff <remote>`:

```
# The "--missing" switch also tells us what files the remote does not possess:
$ brig diff bob --missing
.
├── _ hello.world
├── + videos/
└── README.md  README.md
```

This output resembles the one we saw from `brig tree` earlier. Each node in this tree tells us about something that would happen when we merge. The prefix of each file and the color in the terminal indicate what would happen with this file. Refer to the table below to see what prefix relates to what action:

Symbol	Description
+	This file is only present on the remote side.
−	This file was removed on the remote side.
→	This file was moved to a new location.
*	This file was ignored because we chose to, due to our settings.
	Both sides have changes, but they are compatible and can be merged.
	Both sides have changes, but they are incompatible and result in conflict files.
_	This file is missing on the remote side (needs to be enabled with `--missing`)

Note: Remember that `brig` does not do any actual diffs between files. It does not care a lot about the content. It only records how the file metadata changes and what content hash the file has at a certain point.

If you prefer a more traditional view, similar to `git`, you can use `--list` on `brig diff`.

So in the above output we can tell that *Bob* added the directory `/videos`, but does not possess the `/hello.world` file. He also apparently modified `README.md`, but since we did not, it's safe for us to take over his changes. If we sync now we will get this directory from him:

```
$ brig sync bob
$ brig ls
SIZE   MODTIME          OWNER    PATH                        PIN
443 B  Dec 27 14:44:44  ali      /README.md
443 B  Dec 27 14:44:44  bob      /README.md.conflict.0
12 B   Dec 27 15:14:16  ali      /hello.world
32 GB  Dec 27 15:14:16  bob      /videos
```

You might notice that the `sync` step took only around one second, even though `/videos` is 32 GB in size. This is because `sync` **does not transfer actual data**. It only transferred the metadata, while the actual data will only be loaded when required. This might sound a little inconvenient at first. When I want to watch the video, I'd prefer to have it cached locally before viewing it to avoid stuttering playback. If you plan to use the files immediately, you should be using pinning (see *Pinning*)

Data retrieval

If the data is not on your local machine, where is it then? Thanks to `IPFS` it can be transferred from any other peer that caches this particular content. Content is usually cached when the peer either really stores this file or if this peer recently used this content. In the latter case it will still be available in its cache. This property is particularly useful when having a small device for viewing data (e.g. a smartphone, granted `brig` would run there) and a big machine that acts as storage server (e.g. a desktop).

How are the files secure then if they essentially could be everywhere? Every file is encrypted by `brig` before giving it to `IPFS`. The encryption key is part of the metadata and is only available to the peers that you chose to synchronize with. Think of each brig repository only as a cache for the whole network it is in.

Partial synchronisation

Sometimes you only want to share certain things with certain people. You probably want to share all your `/photos` directory with your significant other, but not with your fellow students where you maybe want to share the `/lectures` folder. In `brig` you can define what folder you want to share with what remote. If you do not limit this, **all folders will be open to a remote by default.** Also note, that if a remote already got some content of a folder you did not want to share, he will still be able to access it. If you're unsure, you should better be restrictive than too permissive.

To add a folder for a specific remote, you can use the `folders` subcommand of `brig remote`:

```
# Starting with next sync, bob will only see the /videos folder:
$ brig remote folder add bob /videos
$ brig remote folder ls bob
/videos
```

If you're tired of typing all of this, be reminded that there are aliases for most subcommands:

```
$ brig rmt f a bob /videos
```

1.2.8 Pinning

How can we control what files are stored locally and which should be retrieved from the network? You can do this by **pinning** each file or directory you want to keep locally. Normally, files that are not pinned may be cleaned up from time to time, that means they are evaded from the local cache and need to be fetched again when being accessed

afterwards. Since you still have the metadata for this file, you won't notice difference beside possible network lag. When you pin a file, it will not be garbage collected.

`brig` knows of two types of pins: **Explicit** and **implicit**.

- **Implicit pins:** This kind of pin is created automatically by `brig` and cannot be created by the user. In the command line output it always shows as blue pin. Implicit pins are created by `brig` whenever you create a new file, or update the contents of a file. The old version of a file will then be unpinned.
- **Explicit pins:** This kind of pin is created by the user explicitly (hence the name) and is never done by `brig` automatically. It has the same effect as an implicit pin, but cannot be removed again by `brig`, unless explicitly unpinned. It's the user's way to tell `brig` »Never forgot these!«.

Note: The current pinning implementation is still under conceptual development. It's still not clear what the best way is to modify/view the pin state of older versions. Time and user experience will tell.

When syncing with somebody, all files retrieved by them are by default **not pinned**. If you want to keep them for longer, make sure to pin them explicitly.

If you never pin something explicitly, only the newest version of all files will be stored locally. If you decide that you need older versions, you can pin them explicitly, so brig cannot unpin them implicitly. For this you should also look into the `brig pin set` and `brig pin clear` commands, which are similar to `brig pin add` and `brig pin rm` but can operate on whole commit ranges.

Garbage collection

Strongly related to pinning is garbage collection. This is normally being run for you every few minutes, but you can also trigger it manually via the `brig gc` command. While not usually needed, it can help you understand how `brig` works internally as it shows what hashes it throws away.

1.2.9 Version control

One key feature of `brig` over other synchronisation tools is the built-in and quite capable version control. If you already know `git` that's a plus for this chapter since a lot of stuff will feel similar. This isn't a big surprise, since `brig` implements something like `git` internally. Don't worry, knowing `git` is however not needed at all for this chapter.

Key concepts

I'd like you to keep the following mantra in your head when thinking about versioning (repeating before you go to sleep may or may not help):

Metadata and actual data are separated. This means that a repository may contain metadata about many files, including older versions of them. However, it is not guaranteed that a repository caches all actual data for each file or version. This is solely controlled by pinning described in the section before. If you check out earlier versions of a file, you're always able to see the metadata of it, but being able to view the actual data depends on having a peer that is being able to deliver the data in your network (which might be yourself). So in short: `brig` **only versions metadata and links to the respective data for each version**.

This is a somewhat novel approach to versioning, so feel free to re-read the last paragraph, since I've found that it does not quite fit what most people are used to. Together with pinning this offers a high degree of freedom on how you can decide what repositories store what data. The price is that this fine-tuned control can get a little annoying. Future versions of `brig` will try to solve that.

For some more background, you can invoke `brig info` to see what metadata is being saved per file version:

```
$ brig show README.md
Path          /README.md
User          ali
Type          file
Size          832 bytes
Inode         4
Pinned        yes
Explicit      no
ModTime       2018-10-14T22:46:00+02:00
Tree Hash     W1gX8NMQ9m8SBnjHRGtamRAjJewbnSgi6C1P7YEunfgTA3
Content Hash  W1pzHcGbVpXaePa1XpehW4HGPatDUJs8zZzSRbpNCGbN2u
Backend Hash  QmPvNjR1h56EFK1Sfb7vr7tFJ57A4JDJS9zwn7PeNbHCsK
```

Most of it should be no big surprise. It might be a small surprise that three hashes are stored per file. The `Backend Hash` is really the link to the actual data. If you'd type `ipfs cat QmPvNjR1h56EFK1Sfb7vr7tFJ57A4JDJS9zwn7PeNbHCsK` you will get the encrypted version of your file dumped to your terminal. The `Content Hash` is being calculated before the encryption and is the same for two files with the same content. The `Tree Hash` is a hash that uniquely identifies this specific node for internal purposes. The `Inode` is a number that stays unique over the lifetime of a file (including moves and removes). It is used mostly in the FUSE filesystem.

Commits

Now that we know that only metadata is versioned, we have to ask »what is the smallest unit of modification that can be saved?«. This smallest unit is a commit. A commit can be seen as a snapshot of the whole repository.

The command `brig log` shows you a list of commits that were made already:

```
      -      Sun Oct 14 22:46:00 CEST 2018 • (curr)
W1kAySD3aKLt Sun Oct 14 22:46:00 CEST 2018 user: Added ali-file (head)
W1ocyBsS28SD Sun Oct 14 22:46:00 CEST 2018 user: Added initial README.md
W1D9KsLNnAv4 Sun Oct 14 22:46:00 CEST 2018 initial commit (init)
```

Each commit is identified by a hash (e.g. `W1kAySD3aKLt`) and records the time when it was created. Apart from that, there is a message that describes the commit in some way. In contrast to `git`, **commits are rarely done by the user themselve**. More often they are done by `brig` when synchronizing.

All commits form a long chain (**no branches**, just a linear chain) with the very first empty commit called `init` and the still unfinished commit called `curr`. Directly below `curr` there is the last finished commit called `head`.

Note: `curr` is what `git` users would call the staging area. While the staging area in `git` is "special", the `curr` commit can be used like any other one, with the sole difference that it does not have a proper hash yet.

Sometimes you might want to do a snapshot or »savepoint« yourself. In this case you can do a commit yourself:

```
$ brig touch A_NEW_FILE
$ brig commit -m 'better leave some breadcrumbs'
$ brig log | head -n 2
      -      Mon Oct 15 00:27:37 CEST 2018 • (curr)
W1hZoY7TrxyK Sun Oct 14 22:46:00 CEST 2018 user: better leave some bread crumbs (head)
```

This snapshot can be useful later if you decide to revert to a certain version. The hash of the commit is of course hard to remember, so if you need it very often, you can give it a tag yourself. Tags are similar to the names, `curr`, `head` and `init` but won't be changed by `brig` and won't move therefore:

```
# instead of "W1hZoY7TrxyK" you also could use "head" here:
$ brig tag W1hZoY7TrxyK breadcrumbs
$ brig log | grep breadcrumbs
$ W1hZoY7TrxyK Sun Oct 14 22:46:00 CEST 2018 user: better leave some bread crumbs␣
↪(breadcrumbs, head)
```

File history

Each file and directory in `brig` maintains its own history. Each entry of this history relates to exactly one distinct commit. In the life of a file or directory there are four things that can happen to it:

- *added:* The file was added in this commit.
- *moved:* The file was moved in this commit.
- *removed:* The file was removed in this commit.
- *modified:* The file's content (i.e. hash changed) was altered in this commit.

You can check an individual file or directorie's history by using the `brig history` command:

```
# or "hst" for short:
$ brig hst README.md
CHANGE  FROM  TO              WHEN
added   INIT  W1ocyBsS28SD    Oct 14 22:46:00
$ brig mv README.md README_LATER.md
$ brig hst README_LATER.md
CHANGE  FROM  TO            HOW                              WHEN
moved   HEAD  CURR          /README.md → /README_LATER.md Oct 15 00:27:37
added   INIT  W1ocyBsS28SD                                   Oct 14 22:46:0
```

As you can see, you will be shown one line per history entry. Each entry denotes which commit the change was in. Some commits were nothing was changed will be jumped over except if you pass `--empty`.

Viewing differences

If you're interested what changed in a range of your own commits, you can use the `brig diff` command as shown previously. The `-s` (`--self`) switch says that we want to compare only two of our own commits (as opposed to comparing with the commits of a remote).

```
# Let's compare the commit hashes from above:
$ brig diff -s W1hZoY7TrxyK W1kAySD3aKLt
.
└── + A_NEW_FILE
```

Often, those hashes are quite hard to remember and annoying to look up. That's why you can the special syntax `<tag or hash>^` to denote that you want to go »one commit up«:

```
brig diff -s head head^
.
└── + A_NEW_FILE
# You can also use this several times:
brig diff -s head^^^ head^^^^^
.
└── + README.md
```

If you just want to see what you changed since `head`, you can simply type `brig diff`. This is the same as `brig diff -s curr head`:

```
$ brig diff
•
└── README.md → README_LATER.md
$ brig diff -s curr head
•
└── README.md → README_LATER.md
```

Reverting to previous state

Until now we were only looking at the version history and didn't modify it. The most versatile command to do that is `brig reset`. It is able to revert changes previously made:

```
# Reset to the "init" commit (the very first and empty commit)
$ brig reset init
$ brig ls  # nothing, it's empty.
```

The key here is that you did not loose any history:

```
$ brig log | head -2
       -     Mon Oct 15 00:51:12 CEST 2018 • (curr)
W1hZoY7TrxyK Sun Oct 14 22:46:00 CEST 2018 user: better leave some bread crumbs␣
↪(breadcrumbs)
```

As you can see, we still have the previous commits. `brig revert` did one thing more than restoring the state of `init` and put that result in `curr`. This also means that you can't really *modify* history. But you can revert it. Let's revert your complete wipe-out:

```
# Reset to the state we had in »breadcrumbs«
$ brig reset breadcrumbs
```

`brig reset` cannot only restore old commits, but individual files and directories:

```
$ brig reset head^^ README.md
```

Note: It is a good idea to do a `brig commit` before a `brig reset`. Since it modifies `curr` you might loose uncommitted changes. It will warn you about that, but you can overwrite that warning with `--force`. If you did a `brig commit` you can simply use `brig reset head` to go back to the last good state.

Nodes that were overwritten with `brig reset` will be unpinned (unless pinned explicitly). Those nodes and their content will be garbage collected after some time. The content may still be accessed through the use of other remotes though.

Other commands

There are a few other commands, but they are not (yet) very useful for most end users. Therefore they will not be explained in depth to save you some mental space. The commands in question are:

- `brig become`: View the metadata of another remote. Good for debugging.
- `brig daemon`: Start the daemon manually. Good for init systems like `systemd`.

- `brig net`: Commands to modify the network status and find other peers.
- `brig edit`: Edit a file in brig with the `$EDITOR`. Good for quick edits.
- `brig fetch`: Manually trigger the fetching of a remote's metadata. Also good for debugging.

Please use `brig help <command>` to find out more about them if you're interested.

1.2.10 Configuration

Quite a few details can be configured in a different way to your liking. `brig config` is the command that allows you to list, get and set individual configuration values. Each config entry already brings some documentation that tells you about its purpose:

```
$ brig config ls
[... output truncated ...]
fs.sync.ignore_moved: false (default)
Default:       false
Documentation: Do not move what the remote moved
Needs restart: no
[... output truncated ...]
$ brig config get repo.password_command
pass brig/repo/password
$ brig config set repo.password_command "pass brig/repo/my-password"
```

1.2.11 Using the gateway

Many users will not run `brig`. Chances are, that you still want to send them your files with too much hassle. `brig` features a *Gateway* to HTTP(S), which comes particularly handy if you happen to run a public server.

The gateway is disabled by default. If you want to start it, use this command:

```
$ brig gateway start
```

Without further configuration, this will create a HTTP server on port `5000`, which can be queried already. There is a small helper that will print you a nice hyperlink to a certain file called `brig gateway url`:

```
$ brig gateway url README.md
http://localhost:5000/get/README.md
```

Opening this in your browser will show you the file's content. If you'd like to use another port than `5000`, you can do so by setting the respective config key:

```
$ brig cfg set gateway.port 7777
```

Note: You can always check the status of the gateway:

```
$ brig gateway status
```

This will also print helpful diagnostics if something might be wrong.

The gateway can be stopped anytime with the following command. It tries to still serve all open requests, so that no connections are dropped:

```
$ brig gateway stop
```

Note: If you want to forward the gateway to the outside, but do not own a dedicated server, you can forward port 5000 to your computer. With this setup you should also get a certficate which in turn requires a DNS name. An easy way to get one is to use dynamic DNS.

Securing access

You probably do not want to offer your files to everyone that have a link. Therefore you can restrict access to a few folders (`/public` for example) and require a user to authenticate himself with a user and password upon access.

By default all files are accessible. You can change this by changing the config:

```
$ brig cfg set gateway.folders /public
```

Now only the files in `/public` (and including `/public` itself) are accessible from the gateway. If you want to add basic HTTP authentication:

```
$ brig cfg set gateway.auth.enabled true
$ brig cfg set gateway.auth.user <user>
$ brig cfg set gateway.auth.pass <pass>
```

If you use HTTP authentication, it is strongly recommended to enable HTTPS. Otherwise the password will be transmitted in clear text.

Running the gateway with HTTPS

The gateway has built-in support for LetsEncrypt. If the gateway is reachable under a DNS name, it is straightforward to get a TLS certificate for it. In total there are three methods:

Method one: Automatic: This works by telling the gateway the domain name. Since the retrieval process for getting a certificate involves binding on port 80, you need to prepare the brig binary to allow that without running as root:

```
$ brig daemon quit
$ sudo setcap CAP_NET_BIND_SERVICE=+ep $(which brig)
```

Afterwards you can set the domain in the config. If the gateway is already running, it will restart immediately.

```
$ brig cfg set gateway.cert.domain your.domain.org
```

You can check after a few seconds if it worked by checking if the `certfile` and `keyfile` was set:

```
$ brig cfg get gateway.cert.certfile
/home/user/.cache/brig/your.domain.org_cert.pem
$ brig cfg get gateway.cert.keyfile
/home/user/.cache/brig/your.domain.org_key.pem
$ curl -i https://your.domain.org:5000
HTTP/2 200
vary: Accept-Encoding
content-type: text/plain; charset=utf-8
content-length: 38
date: Wed, 05 Dec 2018 11:53:57 GMT
```

(continues on next page)

(continued from previous page)

```
This brig gateway seems to be working.
```

This method has the advantage that the certificate can be updated automatically before it expires.

Method two: Half-Automated:

If the above did not work for whatever reasons, you can try to get a certificate manually. There is a built-in helper called `brig gateway cert` that can help you doing that:

```
$ brig gateway cert your.domain.org
You are not root. We need root rights to bind to port 80.
I will re-execute this command for you as:
$ sudo brig gateway cert nwzmlh4iouqikobq.myfritz.net --cache-dir /home/sahib/.cache/
↪brig

A certificate was downloaded successfully.
Successfully set the gateway config to use the certificate.
Note that you have to re-run this command every 90 days currently.
```

If successful, this command will set the `certfile` and `keyfile` config values for you. You can test if the change worked by doing the same procedure as in *method one*. Sadly, you have to re-execute once the certificate expires.

Method three: Manual:

If you already own a certificate you can make the gateway use it by setting the path to the public certificate and the private key file:

```
$ brig cfg set gateway.cert.certfile /path/to/cert.pem
$ brig cfg set gateway.cert.keyfile /path/to/key.pem
```

If you do not own a certificate yet, but want to setup an automated way to download one for usages outside of brig, you should look into certbot.

Redirecting HTTP traffic

This section only applies to you if you choose **method one** from above and want to run the gateway on port 80 (http) and port 443 (https). This has the advantage that a user does not need to specify the port in a gateway URL have which looks a little bit less *»scary«*. With this setup all traffic on port 80 will be redirected directly to port 443.

```
$ brig cfg set gateway.port 443
$ brig cfg set gateway.cert.redirect.enabled true
$ brig cfg set gateway.cert.redirect.http_port 80
```

1.2.12 Running the daemon and viewing logs

As discussed before, the daemon is being started on demand in the background. Subsequent commands will then use the daemon. For debugging purposes it can be useful to run in the daemon in the foreground. You can do this with the `brig daemon` commands:

```
# Make sure no prior daemon is running:
$ brig daemon quit
# Start the daemon in the foreground and log to stdout:
$ brig daemon launch -s
```

The last step will ask for your password if you did not set a password helper program. If you want to quit the instance, either just hit CTRL-C or type `brig daemon quit` into another terminal window.

Logging

Unless you pass the `-s` (`--log-to-stdout` flag) all logs are being piped to the system log. You can follow the log like this:

```
# The actual daemon log:
$ journalctl -ft brig

# The ipfs log:
$ journalctl -ft brig-ipfs
```

This assumes you're using a `systemd`-based distribution. If not, refer to the documentation of your syslog daemon.

Using several repositories in parallel

It can be useful to run more than one instance of the `brig` daemon in parallel. Either for testing purposes or as actual production configuration. If you're planning to do that it is advisable to be always explicit about the port number you're using. Here's an example how you can run two daemons at the same time:

```
# It might be a good idea to keep that in your .bashrc:
alias brig-ali='brig --port 6666'
alias brig-bob='brig --port 6667'

# Subsitute your password helper here:
brig-ali --repo /tmp/ali init ali -w "echo brig/repo/ali"
brig-bob --repo /tmp/bob init bob -w "echo brig/repo/bob"

# Now you can use them normally,
# e.g. by adding them as remotes each:
brig-ali remote add bob $(brig-bob whoami -f)
brig-bob remote add ali $(brig-ali whoami -f)
```

Warning: The examples below are slightly outdated and will be revisited at some point. All commands should still work, but the output might be a little different now. Please refer to the *Getting started* guide for a more up-to-date version.

1.3 Quickstart

This does not really explain the philosophy behind `brig`, but gives a good idea what the tool is able to do and how it's supposed to be used. Users familiar to `git` should be able to grok most of the commands intuitively.

1.3.1 1. Init

Before you can do anything with `brig` you need to create a repository. During this step, also your online identity will be created. So make sure to use a sane username (`sahib@wald.de`) and resource (`laptop`).

As an alternative to entering your password manually, you can use an existing password manager:

1.3.2 2. Adding files

Before synchronizing them, you need to *stage* them. The files will be stored encrypted (and possibly compressed) in blobs on your hard disks.

1.3.3 3. Coreutils

`brig` provides implementations of most file related core utils like `mv`, `cp`, `rm`, `mkdir` or `cat`. Handling of files should thus feel familiar for users that know the command line.

1.3.4 4. Mounting

For daily use and for use with other tools you might prefer a folder that contains the file you gave to `brig`. This can be done via the built-in FUSE layer.

Note: Some built-in commands provided by brig are faster. `brig cp` for example only copies metadata, while the real `cp` will copy the whole file.

If you wish to always have the mount when `brig` is running, you should look into *Making mounts permanent*.

1.3.5 5. Commits

In it's heart, `brig` is very similar to `git` and also supports versioning via commits. In contrast to `git` however, there are no branches and you can't go back in history – you can only bring the history back up front.

1.3.6 6. History

Each file (and directory) maintains a history of the operations that were done to this file.

1.3.7 7. Discovery & Remotes

In order to sync with your buddies, you need to add their *fingerprint* as remotes. How do you get their fingerprint? In the best case by using a separate side channel like telephone, encrypted email or elsewhise. But `brig` can assist finding remotes via the `brig net locate` command.

Note: You should **always** verify the fingerprint is really the one of your buddy. `brig` cannot do this for you.

1.3.8 8. Sync & Diff

Once both parties have setup each other as remotes, we can easily view and sync with their data.

1.3.9 9. Pinning

By default `brig` will only keep the most recent files. All other files will be marked to deletions after a certain timeframe. This is done via *Pins*. If a file is pinned, it won't get deleted. If you don't need a file in local storage, you can also unpin it. On the next access `brig` will try to load it again from a peer that provides it (if possible).

1.4 Frequently Asked Questions

1.4.1 General questions

1. Why is the software named `brig`?

It is named after the ship with the same name. When we named it, we thought it's a good name for the following reason:

- A `brig` is a very lightweight and fast ship.
- It was commonly used to transport small amount of goods.
- A ship operates on streams (sorry)
- The name is short and somewhat similar to `git`.
- It gives you a few nautical metaphors and a logo for free.

Truth be told, only half of the two name givers thought it's a good name, but I still kinda like it.

2. Who develops it?

Although this documentation sometimes speaks of »we«, the only developer is currently Chris Pahl. He writes it entirely in his free time, mostly during commuting with the train.

1.4.2 Technical questions

1. How is the encryption working?

A stream is chunked into equal sized blocks that are encrypted in GCM mode using AES-256. Additionally ChaCha20 (with Poly1305) is currently supported but it might be removed soon. The overall file format is somewhat similar to NaCL secretboxes, but it is more tailored to supporting efficient seeking.

The current default is `ChaCha20`, although machines with the `aes-ni` instruction set might yield significant higher throughput. The source of the encryption layer can be found here. Here's a basic overview over the format:

The key of each file is currently being derived from the content hash of the file (See also Convergent Encryption). If the content changes later, the key does not change since the key is only generated once during the first staging of the file.

Please refer to the implementation for all implementation details for now. No security audits of the implementation have been done yet, therefore I'd appreciate every pair of eyes. Especially while everything is still in flux and won't harm any users.

2. Is there compression implemented?

Yes. The compression is being done before encryption and is only enabled if the file looks compression-worthy. The »worthiness« is determined by looking at its header to guess a mime-type. Depending on the mime-type either `snappy` or `lz4` is selected or no compression is added at all.

The source of the compression layer can be found here. Here's a basic overview over the format:

3. What hash algorithms are used?

Two algorithms are used:

- `SHA256` is used by `IPFS` for every backend hash.
- `SHA3-256` is used as general purpose hash for everything `brig` internal (Content and Tree hash).

Each hash is encoded as multihash. For output purposes this representation is encoded additionally in `base58`. Therefore, all hashes that start with `W1` are `sha3-256` hashes while the ones starting with `Qm` are `sha256` hashes. Keep in mind that `base58` is case-sensitive.

4. What kind of deduplication is currently used?

It is currently only possible to deduplicate between individual versions of a file. And there also only the portion before the modification.

`IPFS` implements deduplication, but it is circumvented by encrypting blocks before giving them over to the backend. Implementing a more proper and informed deduplication is one of the long term goals, which require more thorough interaction with `IPFS`. It is also possible to do some basic deduplication purely on `brig` side since we have more info on the file than `IPFS` has.

5. How fast is the I/O when using `brig`?

Here are some rather outdated graphs where you can get a rough feeling how fast it can be. There are a few rules of thumb with mostly obvious content:

- It it goes over the network, it's the network speed plus a smaller constant overhead.
- If it comes over FUSE, it is quite a bit slower than over `brig cat`.
- If you do not use compression, writing and reading will be faster.

The graphs below only measure in-memory performance compared to a `dd` like speed (see the »baseline« line).

Your mileage may vary and you better do your own benchmarks for now.

Todo: Explain/Update those graphs.

1.5 Comparison with other tools

When announcing `brig` (or any other software in general) the first question is usually something like *»But isn't there already X?«* and sometimes even *»Why don't you just contribute to other projects?«*. This document tries to find an answer to both questions. The answer will obviously be biased, so take it with a fair grain of salt.

Yes, there is other software in this world. But this is always a matter of tradeoffs the author of each individual package has chosen. One application might not run on your platform, the next might not be secure enough for your needs, the other one decides to steal your data or are proprietary/too complicated to setup or maintain. This comparison does not aim to blame any tool (well, except if it's stealing data) but tries to give a (subjective) view on where the focus of each tool differs.

Todo: Translate the comparison from master thesis, update and restructure. Low priority though.

1.6 Roadmap

This document lists the improvements that can be done to `brig` and (if possible) when. All features below are not guaranteed to be implemented and, can be seen more as possible improvements that might change during implementation. Also it should be noted that each future is only an idea and not a fleshed ou implementation plan.

Bug fixes and minor changes in handling are not included since this document is only for »big picture« ideas. Also excluded are stability/performance improvements, documentation and testing work, since this is part of the »normal« development.

1.6.1 First Release

The first real release (0.2.0 »Baffling Buck«) is planned for end of November 2018. Until then, the software should provide the following basic features:

- Stable command line interface.
- Git-like version control
- User discovery
- User authentication
- Fuse filesystem

All of the above features are currently already implemented and work. Focus is on stabilizing the features and making it somewhat release ready. All those features combined do already provide some usefulness, but for being a day-to-day useful tool, it takes a few more features, especially being able to sync with offline peers over a trusted partner.

Note that there will be no stability guarantees before version 1.0.0.

1.6.2 Future

Those features should be considered after releasing the first prototype. A certain amount of first user input should be collected to see if the direction we're going is valid.

Gateway: Provide a built-in (and optional) http server, that can »bridge« between the internal ipfs network and people that use a regular browser. Instances that run on a public server can then provide hyperlinks of files to non-brig users.

Shelf instances: Special instaces of brig, that operate automatically and are meant to be run on public servers. They can be used to exchange data between users that are not online at the same day (e.g. due to timezone differences).

Automatic syncing: Automatically publish changes after a short amount of time. If an instance modified some file other nodes are notified and can decide to pull the change.

Intelligent pinning strategies: By default only the most recent layer of files are being kept. This is very basic and can't be configured currently. Some users might only want to have only the last few used files pinned, archive instances might want to pin almost everything up to a certain depth.

Improve read/write performance: Big files are currently hold in memory completely by the fuse layer (when doing a flush). This is suboptimal and needs more intelligent handling and out-of-memory caching of writes.

More automated authentication scheme: E-Mail-like usernames could be used to verify a user without exchanging fingerprints. This could be done by e.g. sending an activation code to the email of an user (assuming the brig name is the same as his email), which the brig daemon on his side could read and send back.

Format and read fingerprint from QR-Code: Fingerprints are hard to read and not so easy to transfer and verify. QR-Code could be a solution here, since we could easily snap a picture with a phone camera or print it on a business card.

1.6.3 Far Future

Those features are also important, but require some more in-depth research or more work and are not the highest priority currently.

Port to other platforms: Especially Windows and eventually Android. This relies on external help, since I'm neither capable of porting it, nor really a fan of both operating systems.

Implement alternative to fuse: FUSE currently only works on Linux and is therefore not usable outside of that. Windows has something similar (called Dokan). Alternatively we could also go on by implementing a WebDAV server, which can also be mounted.

Ensure N-Copies: It should be possible to define a minimum amount of copies a file has to have on different peers. This could be maybe incorporated into the pinning concept. If a user wants to remove a file, brig should warn him if he would violate the min-copies rule. This idea is shamelessly stolen from `git-annex`.

Implement a portable GUI: Many user will rely on a GUI to configure brig and hit the »sync button«. We should optionally provide this in a portable fashion (browser based app? I kinda hate myself for proposing this though...). Most of the time the GUI should be a simple tray icon that can be clicked to sync. A rough and a little exaggerated mock-up was already drawn up for GNOME:

1.7 How to contribute

Note: Pure feature requests will currently **NOT BE** considered. Read on for the reasoning and details.

This software is still in very early stages and still needs to find the direction where it's usable for a high number of people. Implementing features that only a very limited number of users will benefit from is one of the highest risk currently. Since we also want to make sure that the feature set of »brig« makes sense as a whole and all features are orthogonal, we will ignore typical feature request at the moment.

What we want instead are *experience reports*. We want you to use the current state of the software and write down the following:

- Was it easy to get »brig« running?
- Was it easy to understand it's concepts?
- What is your intended usecase for it? Could you make it work?
- If no, what's missing in your opinion to make the usecase possible?
- Anything else that you feel free to share.

After this we'll try to analyze your reports and create feature requests acccordingly. Once those were implemented we will probably allow traditional feature requests to be made.

What do we want to prevent by this? Getting 20 feature requests, 5 of them contradicting each other and. In short: featuritis. It's quite hard to figure out what users wants from a developers standpoint. So this will hopefully give us some more insights.

Are bug reports okay? Sure. If you already fix the bug it's even better. Please use the `brig bug` command to get a template with all the info we need.

Are very small feature requests okay? If it's only about changing or extending an existing feature, it's probably fine. Feel free to create an issue on GitHub to check back on this before you do any actual change.

Also, the developer of this software is currently doing all of this is in his free time. If you're willing to offer any financial support feel free to contact me.

1.7.1 What to improve

The following improvements are greatly appreciated:

- Bug reports & fixes.
- Documentation improvements.
- Porting to other platforms.
- Writing tests.

1.7.2 Workflow

Please adhere to the general *GitHub workflow*_, i.e. fork the repository, make your changes and open a pull request that can be discussed.

If you contribute code, make sure:

- Tests are still running and you wrote test for your new code.
- You ran `gofmt` over your code.
- Your pull requests is opended against the `develop` branch.