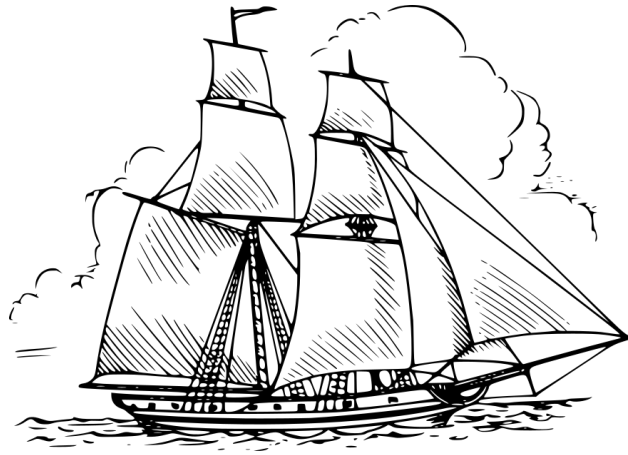

brig

Release v0.0.0

Mar 23, 2021

1	What is brig?	3
2	What is brig not?	5
3	I have questions!	7
4	Current Status	9
5	Table of Contents	11
5.1	Installation	11
5.2	Specific distributions	11
5.3	Compiling yourself	11
5.4	Setting up IPFS	12
5.5	Getting started	13
5.6	Creating a repository	14
5.7	Running the daemon and viewing logs	16
5.8	Locking the repository.	17
5.9	Adding & Viewing files	17
5.10	Coreutils	18
5.11	Hints - Configuring encryption & compression	18
5.12	Mounting repositories	19
5.13	Remotes	21
5.14	Syncing	23
5.15	Version control	26
5.16	Pinning	30
5.17	Using the gateway / UI	31
5.18	Configuration	35
5.19	Quickstart	36
5.20	Frequently Asked Questions	37
5.21	Features	39
5.22	Comparison with other tools	41
5.23	Roadmap	42
5.24	How to contribute	43



CHAPTER 1

What is brig?

`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. Think of it as a swiss army knife for file synchronization or as a peer to peer alternative to *Dropbox*.

Key feature highlights:

- Encryption of data during storage 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`.
- Gateway and Web based UI to share normal HTTP/S links with other users.
- Auto-updating facility that will sync on any change.
- Completely free software under the terms of the AGPL.
- ...

Please refer to the [Features](#) for more details. If you want a visual hint how `brig` looks on the commandline, refer to the [Quickstart](#).

CHAPTER 2

What is `brig` not?

`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 and should not be used when you are in need of high throughput - at least not at the moment.

CHAPTER 3

I have questions!

Please ask in one of those places:

- [GitHub Issue Tracker](#): All things like bug reports or feature requests.
- The matrix chat room `#brig` on `matrix.org`. Just [pick a client](#) and join the room or click [this link](#) directly.

CHAPTER 4

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. Other platforms should be able to compile, but there are currently not guarantees that it will work.

5.1 Installation

We provide pre-compiled binaries on every release. `brig` comes to your computer as a single binary that includes everything you need. See here for the release list:

<https://github.com/sahib/brig/releases>

Just download the binary for your platform, unpack it and put it somewhere in your `$PATH` (for example `/usr/local/bin`).

If you trust us well enough, you can also use this online installer to download the latest stable `brig` binary to your current working directory:

```
$ bash <(curl -s https://raw.githubusercontent.com/sahib/brig/master/scripts/install.  
↪sh)
```

5.2 Specific distributions

Some distributions can install `brig` directly via their package manager. Those are currently:

- Arch Linux ([PKGBUILD](#); builds `develop` branch)

5.3 Compiling yourself

If you use a platform we don't provide binaries for or if you want to use a development version, you're going to have to compile `brig` yourself. But don't worry that's quite easy. We do not have many dependencies. You only need two things: The programming language *Go* and the version control system `git`.

5.3.1 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 (≥ 1.10). 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 it's too old you can always use tools like gvm to get a more recent version.

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.

5.3.2 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.
$ ./scripts/install-task.sh             # Install the build system.
$ task                                  # Build the binary.
```

Execution might take a few minutes though because all of brig is being compiled during the task step - this also includes the download of all dependencies.

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.

5.3.3 Step 2: Test if the installation is working

If everything worked, there will be a brig binary in \$GOBIN.

```
$ brig help
```

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

5.4 Setting up IPFS

brig requires a running IPFS daemon. While brig has ways to do install a IPFS daemon for you, it is preferable to install it via your package manager or via the official way:

<https://docs.ipfs.io/introduction/install>

Continue with [Getting started](#) or directly go to [Quickstart](#) if you just need a refresh on the details.

5.5 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. Most of the steps here will include working in a terminal, since this is the primary way to interact with brig. Once setup you have to choice to use a browser application though.

5.5.1 Precursor: The help system

Before we dive in, we go over a few things that will make your life easier along the way. brig has some built-in helpers to serve as support for your memory. If you're not interested in that you can skip right to the next section. But please check those help texts before asking questions.

Built-in reference documentation

Every command offers detailed built-in help, which you can view using the `brig help` command. This often usage examples too:

```
$ 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

Warning: The shell autocompletion is still under development. It might still yield weird results and the usability needs to be improved definitely. Any help welcome!

If you don't like to remember the exact name of each command, you can use the provided autocomplete. 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>`. Other shells are not supported right now sadly.

Open the online documentation

By typing `brig docs` you'll get a tab opened in your default browser with this domain loaded. Please stop typing `brig docs` into Google and save some energy.

Reporting bugs

If you need to report a bug (thank you!) 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. If you don't have internet or do not want to sign up, you can still generate a bug report template via `brig bug -s`.

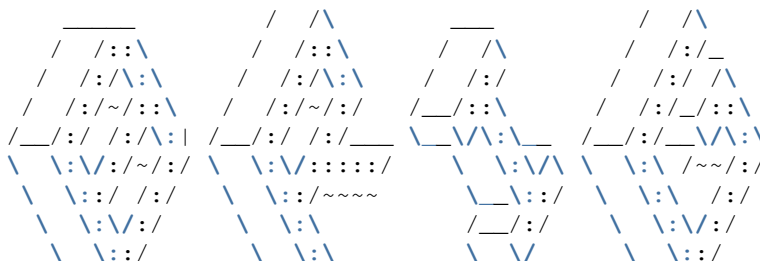
5.6 Creating a repository

You need a central place where `brig` stores its metadata. This place is called a »repository« or short »repo«. This is not the place, where your files are stored. Those are copied (if you did setup IPFS in a normal way) to `~/.ipfs`. Keep in mind that `brig` will copy files and thus will never modify the original files on your hard drive.

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.
# The repository is created by default in the current working directory.
# (This can be changed via `brig --repo`)

$ mkdir repo && cd repo
$ brig init ali@woods.org/desktop -w 'echo my-password'
```



(continues on next page)

(continued from previous page)

```

  \_/\      \_/\      \_/\

A new file README.md was automatically added.
Use 'brig cat README.md' to view it & get started.

$ ls
config.yml  gateway      immutable.yml  keyring
metadata    README.md    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 in the background. Every other `brig` commands will communicate with it via a local network socket. If the daemon does not run yet, it will be started for you in the background without you noticing.

Note: If no IPFS daemon is running, `brig` will start one for you. If you don't have `ipfs` installed, it will even install and set it up for you. By default, `brig init` will also set some default options that help `brig` to run a bit smoother. If you do not want those, please add `--no-ipfs-optimization` to the `init` command above.

5.6.1 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 user name could be `dad@smith.org/desktop` and the other `dad@smith.org/server`.

5.7 Running the daemon and viewing logs

The following sections are not a required read. They are useful to keep in mind, but in the ideal case you're don't even need to think about the daemon.

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
```

If you want to quit the instance, either just hit CTRL-C or type `brig daemon quit` into another terminal window.

5.7.1 Logging

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

```
# Follow the actual daemon log:
$ journalctl -ft brig
```

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

5.7.2 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. In order for the `brig` client to know what daemon to talk to, you have to be specific about the repository (`--repo`) path. Here is an example:

```
# Be explicit
$ brig --repo /tmp/ali init ali -x --ipfs-path ~/.ipfs
$ brig --repo /tmp/bob init bob -x --ipfs-path ~/.ipfs2

# Since you specified --repo we know what daemon to talk to.
# You can also set BRIG_PATH for the same effect:
$ BRIG_PATH=/tmp/ali brig ls
<file list of ali>

# Add some alias to your .bashrc to save you some typing:
$ alias brig-ali="brig --repo /tmp/ali"
$ alias brig-bob="brig --repo /tmp/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)
```

Note: It is possible to have several repositories per IPFS instances. Since things might get confusing though when it comes to pinning, it is recommended to have several IPFS daemons running in this case. This is done via the `--ipfs-port` flag in the example above.

5.8 Locking the repository.

The repository on disk is not encrypted. If you plan on moving the repository to somewhere else, e.g. by copying it onto a USB stick and physically moving it somewhere else you should always consider to first create an encrypted archive out of it and unpack it on the target machine. `brig` has a built-in helper for this. Please refer to `brig pack-repo --help` and `brig unpack-repo --help`.

5.9 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
986 B	Mon Mar 4 23:04:07 CET 2019	/README.md	✓
12 B	Mon Mar 4 23:04:23 CET 2019	/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.

Note: `brig` always copy the data. If you happen to change the original file, the change will not propagate to the file in `brig`. You have to re-stage it to reflect the change.

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 state¹.

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

¹ Pinning and pin states are explained [Pinning](#) and are not important for now.

5.10 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
. ✓
├─ hello.world ✓
├─ photos/ ✓
│   └─ me.png ✓
└─ README.md ✓

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. They work in most cases like their pendant. Also note that there is no `brig cd` currently. All paths must be absolute.

5.11 Hints - Configuring encryption & compression

Often times you might want not encrypt all files. A typical use case would be to have a `/public` folder where you put in files to share with your friends. Probably there are some freely available files in there, you got from some corners of the internet (for example your excellent meme collection). Those files don't need encryption and probably not even compression. If you want to exclude the `/public` folder from both you can give `brig` a hint:

```
# let's assume /public exists already:
$ brig hints set /public --compression none --encryption none
$ brig hints
PATH      ENCRYPTION  COMPRESSION
/          aes256gcm   guess
/public   none        none
```

As you might notice, there is already one hint set by default for the root directory. If you want to change the global defaults, you can simply modify this one. Below you see the hint you just created. This however does not change any existing files. It just tells `brig` »next time you modify those files, please use those algorithms«. If you want to make sure the files are changed to use the algorithm you set, then you can use the `stage --recode` command:

```
$ brig stage --recode /public
```

If you do this, you can observe a small change when looking at the `IsRaw` attribute of the file's info:

```
# This was 'true' before the recode.
$ brig info --format '{{ .IsRaw }}' /public/cat-meme.png
false
```

The `IsRaw` attribute tells you if you could download this file by its hash from an IPFS gateway. If its true, `brig` does not touch it at all. This is an useful attribute you want to share a file with your non-tech friends who prefer to click on a regular HTTP URL: You can just point them a [IPFS gateway](<https://docs.ipfs.io/concepts/ipfs-gateway>).

5.11.1 Available encryption algorithms

Note: The THROUGHPUT numbers shows the relative, average performance compared to `none`. Your mileage may vary a lot. Those number should serve as rough guideline and were obtained by the built-in `brig debug iobench` utility using the `fuse-{read,write}-mem` benchmark. If you want the details you can run the benchmarks yourself. As you can see from the numbers, the additional encoding by brig does not make things substantially slower.

If you wonder how some benchmark are faster than `none`: Compression compacts the stream heavily (if the data is well compressible). Therefore less bytes need to be transferred and encrypted or decrypted. Quite surprisingly, in some cases compression can make things faster.

Also note that this was measured without caching. If no data is modified your operating system will likely cache data for you and speed up things.

NAME	DESCRIPTION	READ THROUGHPUT	WRITE THROUGHPUT
<code>aes256-gcm</code>	The default. AES with 256 bit key in GCM cipher mode. Fast on modern CPUs.	80-85%	85-95%
<code>chacha20</code>	Streaming cipher with Poly1305 MAC. Good for old CPUs without AES-NI.	70-85%	80-90%
<code>none</code>	Disables encryption. Fast, but only good for public files.	100%	100%

5.11.2 Available compression algorithms

NAME	DESCRIPTION	READ THROUGHPUT	WRITE THROUGHPUT
<code>snappy</code>	High throughput, relative low compression ratio.	80-105%	95-130%
<code>lz4</code>	Middle throughput, slightly higher compression ratio than snappy.	77-93%	85-105%
<code>zstd</code>	Low throughput, highest compression ratio.	55-95%	35-100%
<code>guess</code>	Chooses suitable algorithm based on file ending, size and mime type.	–	–
<code>none</code>	Disables compression.	100%	100%

5.12 Mounting repositories

Using commands like `brig cp` 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 way for scripting own, more elaborate 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 the tools you are used to. 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
```

(continues on next page)

(continued from previous page)

```
$ 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 like a normal one, but check for the CAVEATS below. You can have any number of mounts. This proves especially useful when only mounting a subdirectory (let's say we have a directory called `/Public`) with the `--root` option of `brig mount` and mounting all other files as read only (`--readonly`).

```
$ 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
```

5.12.1 Remote access

Working with remote data does often not work extremely well with the file abstraction that does not play well with timeouts. This often causes applications to hang for indefinite times, since they are not most of the time not build for data that might not be delivered immediately. For this very common case we have the `--offline` flag. It will error out immediately on files that are not in our local cache:

```
$ brig mount /tmp/mount --offline
# Or with fstab:
$ brig fstab add some-mount /tmp/mount --offline
```

If you have a remote file you want to read, you can do this to make it cached locally:

```
$ brig cat /remote-file > /dev/null
```

After `brig cat` run, you should be able to view the file normally in the mount.

5.12.2 Making mounts permanent

All mounts that are created via `brig mount` will be gone after a daemon restart. If you a typical set of mounts, you can persist them with the `brig fstab` facility:


```

$ 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` or restart the daemon. The opposite can be achieved by executing `brig fstab apply --unmount`.

CAVEATS: The FUSE filesystem is not yet perfect and somewhat experimental. 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.

5.13 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 term used in the `git` world.

A remote consists of three things:

- **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 email¹ 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.
- **A bunch of settings and state:** `brig` knows about every remote if it is online and/or authenticated. Additionally you can set a few remote-specific configuration settings like automatic updating.

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

¹ To be more exact, it resembles an `XMPP` or `Jabber-ID`.

```
# NOTE: The hash will look different for you:
$ brig whoami
ali@woods.org/desktop_
↪QmTTJbkfG267gidFKfDTV4j1c843z4tkUG93Hw8r6kZ17a:WlnayTG5UMcVxy9mFFNjuZDUB7uVTnmwFYiJ4Ajr1TP3bg
```

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. There is no way to know if somebody is the person he is pretending to be, so validate that over a separate channel - that's sadly something where `brig` can't help you yet.

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[...]:WlUDvKzjRPb4rbbk[...]
```

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*:²

```
$ 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:
    WlnayTG5UMcVxy9mFFNjuZDUB7uVTnmwFYiJ4Ajr1TP3bg
```

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:

² 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 list
NAME      FINGERPRINT  ROUNDTrip  ONLINE AUTHENTICATED LASTSEEN          AUTO-UPDATE
bob       QmUDSxt27    0s         ✓       ✓               Apr 16 17:31:01   no
$ 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 its 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.

5.14 Syncing

Now that we added a remote, a whole new set of features are available to us. 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 a bit of 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 <code>--missing</code>)

Note: Remember that `brig` does not do any actual diffs between files, i.e. it will not show you what line changed. 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](#))

5.14.1 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.

5.14.2 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. On the other hand you maybe want to share the `/lectures` folder with them. 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 very short aliases for most subcommands:

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

In some cases you might not trust your peers with some folders or don't want to have modifications in that specific folder. For this case, `brig` supports adding a folder as `--read-only`. Other remotes still will have access to the folder, but whenever we sync with them the changes they made are ignored. You can add a read-only folder by adding the `--read-only` switch to the command above:

```
$ brig rmt f a bob /videos --read-only
```

Note: If you want to overwrite an existing folder with new settings, you can use the `set` subcommand:

```
$ brig remote folder set bob /videos -c embrace --read-only
```

See below for explanation on those additional options.

5.14.3 Conflicts

Whenever two repositories have a file at the same path, `brig` needs to do some conflict resolving. If those files are equal or if they share common history and did not diverge there is nothing to fear. But what if both sides have different versions of a file without common history? In this case `brig` offers you to handle conflict by one of the three strategies:

- `ignore`: Ignore the change from the remote side.
- `embrace`: Ignore our state and take over the remote's change.
- `marker`: Create a conflict file with the same name but a `.conflict` ending. Leave it to the user to resolve the conflict. This is the **default**.

You can configure this behavior by using `brig cfg`:

```
$ brig cfg set fs.sync.conflict_strategy marker
```

In some cases this might not be enough though. Sometimes you might want to say »I trust this remote, always accept their changes«. You can do this by setting the conflict strategy per remote. If no specific conflict strategy is set, `fs.sync.conflict_strategy` is used. You can set the strategy by using a subcommand of the `brig remote` family:

```
# Always take the versions of bob on conflicts:
$ brig remote conflict-strategy embrace bob
```

Still not enough? You can also set the conflict strategy per folder. This will trump the per-remote folder strategy:

```
# Use the default in all folders but use "embrace" in this one:
$ brig remote folder add bob /collab -c embrace
```

5.14.4 Automatic Updating

Warning: This feature is experimental and builds upon the also experimental pubsub experiment of the IPFS project. Use with care.

If you do not want to hit `brig sync` every time somebody in the network changed something, you can enable the automatic updating for any remote you like. Let's suppose we are `ali` and want to receive updates on every change of `bob`, we should simply add the following:

```
$ brig remote auto-update enable bob

# (You can also abbreviate most of that:)
# brig rmt au e bob
```

Alternatively, we could have used the `-a` switch when adding `bob` as remote:

```
$ brig remote add bob -a
```

In any case, an initial sync is performed with this remote and a sync on every change that `bob` published. Keep in mind that `bob` will not receive your updates by default, he needs to decide to use auto updating for himself. You can watch the times when your repository was updated automatically by looking at `brig log`:

```
$ brig log
-      Sun Dec 16 18:24:27 CET 2018 • (curr)
WlkGKKviWCBY Sun Dec 16 18:24:27 CET 2018 sync due to notification from »bob« (head)
...
```

5.14.5 Pushing changes

As you saw above, doing a `brig sync` won't do a bidirectional synchronisation. It will only fetch metadata from the remote and modify our local state with it. In some cases you might want to push data to a remote - especially when it is on one of your machines and you use for example as archival repository. By default pushing to a remote is rejected. You can enable it on a per-remote basis with this command out of the `brig remote` family of commands:

```
# Allow bob and charlie to auto push to us.
$ brig remote auto-push enable bob charlie
```

Now either `bob` or `charlie` can do this from their machines:

```
# bob's machine:
$ brig push ali
```

This will simply ask `ali` to do a sync with `bob`.

5.15 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.

5.15.1 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 [Pinning](#) section. 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 we'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.

5.15.2 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
W1D9KsLNNv4  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 themselves**. 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)
WlhZoY7TrxyK 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 "WlhZoY7TrxyK" you also could use "head" here:
$ brig tag WlhZoY7TrxyK breadcrumbs
$ brig log | grep breadcrumbs
$ WlhZoY7TrxyK Sun Oct 14 22:46:00 CEST 2018 user: better leave some bread crumbs_
↪ (breadcrumbs, head)
```

5.15.3 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   WlocyBsS28SD     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   WlocyBsS28SD              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 where nothing was changed will be jumped over except if you pass `--empty`.

5.15.4 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 WlhZoY7TrxyK WlkAySD3aKLt
.
└─ + A_NEW_FILE
```


Often, those hashes are quite hard to remember and annoying to look up. That's why you can use 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
```

5.15.5 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 lose any history:

```
$ brig log | head -2
-      Mon Oct 15 00:51:12 CEST 2018 • (curr)
WlhZoY7TrxyK 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 lose 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.

5.16 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 from the network when being accessed again. Since you still have the metadata for this file, you won't notice the difference beside some possible network lag. When you pin a file however, it will not be garbage collected and stays in your local cache until unpinned.

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 is always shows as blue pin. Implicit pins are created by brig whenever you create a new file, or update the contents of a file. Implicit pins are managed by brig and as you will see later, it might decide to save you some space by unpinning old versions.
- **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 by the user. This is a good way of telling brig to never unpin this specific version. Use this with care, since it is easy to forget about explicit pins.

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.

5.16.1 Garbage collection

Strongly related to pinning is garbage collection. Whenever you need to clean up some space, you can just type `brig gc` to remove all unpinned files from the cache.

By default, the garbage collector is also run once every hour. You can change this interval by setting `brig config set repo.autogc.interval` to 30m for example. You can also disable this automatic garbage collection by issuing `brig config set repo.autogc.enabled false`.

5.16.2 Repinning

Repinning allows you to control how many versions of each file you want to store and/or how much space you want to store at most. The repinning feature is controlled by the following configuration variables:

- **fs.repin.quota:** Maximum amount of data to store in a repository.
- **fs.repin.min_depth:** Keep this many versions definitely pinned. Trumps quota.
- **fs.repin.max_depth:** Unpin versions beyond this depth definitely. Trumps quota.
- **fs.repin.enabled:** Whether we should allow the repinning to run at all.
- **fs.repin.interval:** How much time to wait between calling repinning automatically.

Normally repinning will run for you every 15 minutes. You can also trigger it manually:

```
$ brig pin repin
```

By default, brig will keep 1 version definitely (**fs.repin.min_depth**) and delete all versions starting with the 10th (**fs.repin.max_depth**). The default quota (**fs.repin.quota**) is 5GB. If repin detects files that need to be unpinned, then it will first unpin all files that are beyond the max depth setting. If this is not sufficient to stay under the quota, it will delete old versions, layer by layer starting with the biggest version first.

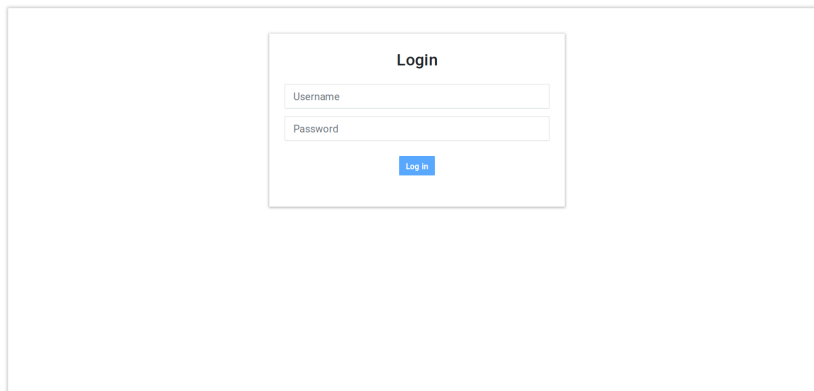
5.17 Using the gateway / UI

5.17.1 Gateway Screenshots

The gateway UI consists of several tabs, which are briefly shown below to give you a short impression of it.

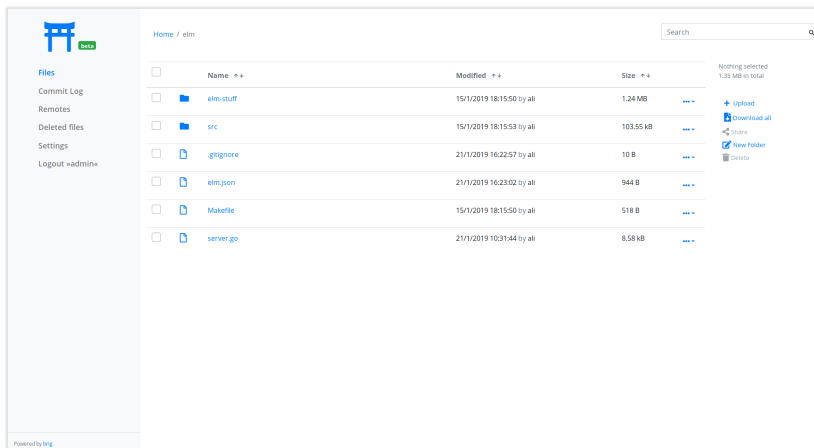
Login screen

Allows you to login. You can also come back here to change the user. It is also possible to login anonymously, as you will see below.



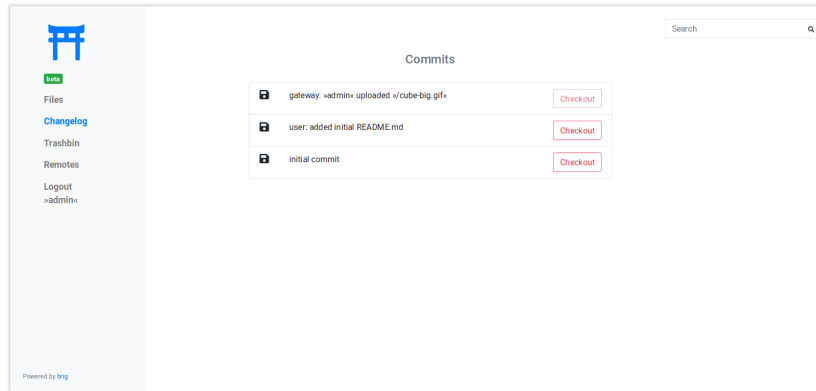
File Browser

The main view. Lists the directory tree and file attributes. Allows for modification, uploading and everything what you'd expect.



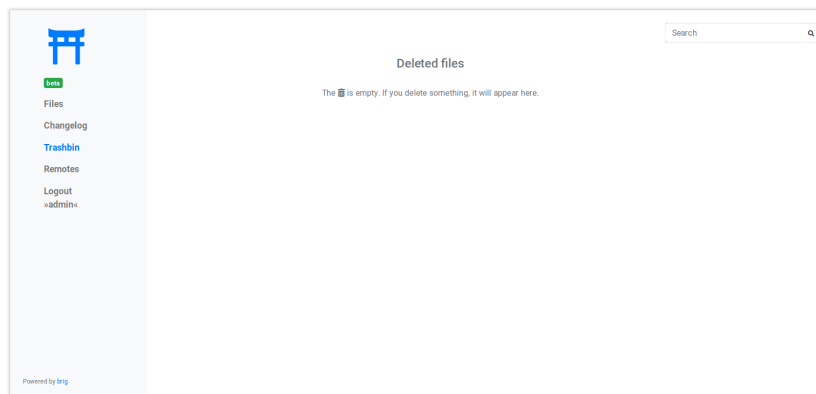
Changelog View

A list of commits. You are able to jump back to a specific commit.



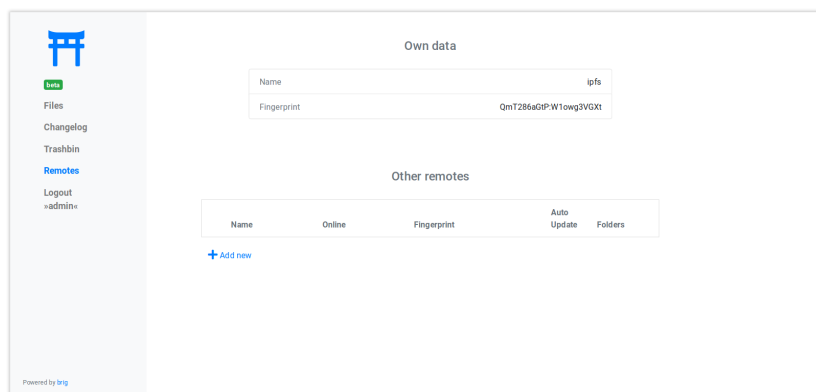
Trashbin

A list of deleted files. If you deleted something you will be able to get it back here.



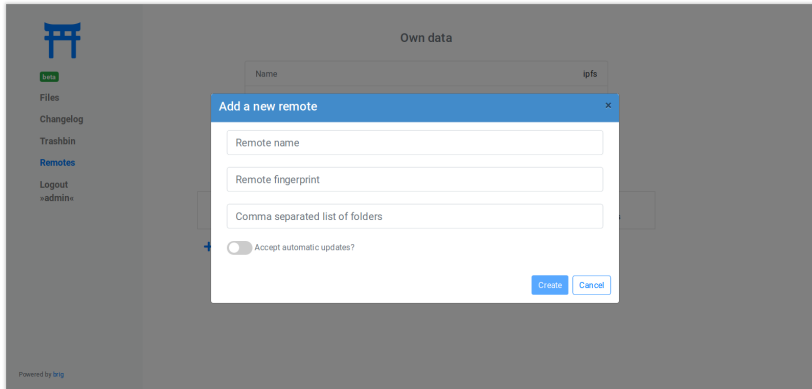
Remote List

If your user is privileged enough, you can see and edit the list of remotes and adjust settings in it.



Remote Add Dialog

A sample dialog. The UI uses many of them.



5.17.2 Introduction

Many users will not run `brig` themselves, so you won't be able to `brig sync` with them. Chances are that you still want to send or present them your files without too much hassle. `brig` features a *Gateway* to HTTP(S), which comes particularly handy if you happen to run a public server and/or want to provide a GUI to your users. It also includes an easy to use UI that is enabled by default.

Before you do anything, you need to add a »user« to your gateway. This user is different than remotes and describes what credentials can be used to access the gateway. You can add a new user like this:

```
$ brig gateway user add admin my-password
# or shorter:
# brig gw u a admin my-password
$ brig gateway user list
NAME  FOLDERS
admin /
```

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 (**not HTTPS!**) server on port 6001, which can be used already. If you access it under `http://localhost:6001` you will see a login mask where you can log yourself in with the credentials you entered earlier.

If you'd like to use another port than 6001, 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:

```
$ brig gateway stop
```

There is also 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:6001/get/README.md
```

5.17.3 Folder management

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) for individual users. By default a user is allowed to see everything. If you want a user that can only access the `/public` folder simply add him as follows:

```
$ brig gw user add my-new-user /public
```

Now only the files in `/public` (and including `/public` itself) are accessible from the gateway.

5.17.4 User right management

We already discussed the adding of a user above. There is a little more to that though. You can add users with different rights. In total there are 5 different rights currently:

- **fs.view**: View and list all files.
- **fs.edit**: Edit and create new files.
- **fs.download**: Download file content.
- **remotes.view**: View the remotes tab.
- **remotes.edit**: Edit the remotes tab.

When you add users you can give a new user a comma separated list of rights via the `-r` switch:

```
$ brig gw user add my-new-user -r 'remotes.view,remotes.edit'
```

For your convenience there are a bunch of presets which will do the work for you in 99% of the cases:

- `--role-admin, -a`: Add this user as admin (short for `»-r 'fs.view,fs.edit,fs.download,remotes.view,remotes.edit'«`)
- `--role-editor, -b`: Add this user as collaborator (short for `»-r 'fs.view,fs.edit,fs.download,remotes.view'«`)
- `--role-collaborator, -c`: Add this user as collaborator (short for `»-r 'fs.view,fs.edit,fs.download'«`)
- `--role-viewer, -d`: Add this user as viewer (short for `»-r 'fs.view,fs.download'«`)
- `--role-link-only, -e`: Add this user as linker (short for `»-r 'fs.download'«`)

5.17.5 Running the gateway with HTTPS

By default, we run with `http` only. If you want to expose the gateway under a domain to the internet you should secure it with `https`. Since brig's gateway might is likely not the only service you want to expose we recommend a webserver like [Caddy](#) (which is great software in general!) that automatically fetches certificates and reverse-proxies traffic to the gateway. A minimal `Caddyfile` can look like this:

With this setup, your gateway would be reachable under `https://your.domain.org/gateway`. You can of course choose a different route or even a sub-domain. Maybe you also want to setup compression or require a client certificate. Refer to the [Caddy documentation](#) for more information.

5.17.6 Allowing anonymous access

If you want to run a public gateway (for example for a group of friends), then you might want to enable anonymous access. In this mode you will be logged in right away to the gateway without facing the login screen. You still have the option to go to the login screen and become another user.

You can enable the anonymous mode like this:

```
$ brig cfg set gateway.auth.anon_allowed true
```

Additionally you have to create an anon user. This allows you to define what rights the anonymous users have and what folders they may access:

```
# Give the anonymous users only access to /public and don't let them modify anything:
$ brig gw u add anon anon --role-viewer /public
```

If you want to change the name of the anon user to something else (for whatever reason) you can do so by setting the `auth.anon_user` variable. You also have to re-add the user above with the new name.

```
$ brig cfg set gateway.auth.anon_user some_other_anon_name_that_is_not_used
```

5.18 Configuration

As mentioned earlier, we can use the built-in configuration system to configure many aspects of `brig` functionality to our liking. Every config entry of `brig` consists of 4 values:

- Key - always a dotted, hierarchical path like `fs.sync.ignore_moved`.
- Value - some value that is validated depending on the key.
- Default - The default value.
- Documentation - A short description of what this entry can do for you.
- Needs restart - A boolean indicating whether you have to restart the service to take effect.

When you type `brig cfg` you will see all keys with the aforementioned entries:

```
$ brig config ls
[...]
fs.sync.ignore_moved: false (default)
  Default:      false
  Documentation: Do not move what the remote moved
  Needs restart: no
[...]
```

Additionally, we support of course the usual operations:

```
$ brig config get repo.password_command
pass brig/repo/password
$ brig config set repo.password_command "pass brig/repo/my-password"
```

5.18.1 Profiles

Todo: Implement configuration profiles.

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.

5.19 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.

5.19.1 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:

5.19.2 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.

5.19.3 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.

5.19.4 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](#).

5.19.5 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.

5.19.6 6. History

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

5.19.7 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 elsewhere. 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.

5.19.8 8. Sync & Diff

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

5.19.9 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).

5.20 Frequently Asked Questions

5.20.1 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.
- Words like »bright«, »brigade« and many others start with it

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.

5.20.2 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:

- If 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.

5.21 Features

Note: The features below are actually available, but before version 1.0.0 we won't give any guarantees regarding stability or edge cases. Your mileage may vary currently.

5.21.1 Encrypted and compression built-in

- All data is encrypted during storage and transport using AES-256 in GCM mode.
- Optional compression algorithm is selected based on the file type.
- Hints can be given to change the default algorithm for certain or all files.
- Keys are stored as part of the metadata during synchronisation.

5.21.2 Easy Version control

- Simplified `git`-like version control only limited by your storage space.
- Synchronization algorithm that can handle `moved files` and `empty directories` and files.
- Auto-updating facility that will sync on any change.
- Configurable conflict handling.

5.21.3 Separation between data and metadata

- Your data does not need to be stored on the device you are currently using.
- Pin the data you want to use to your local storage. Every repository acts as cache of all the files you have access to.
- Keep a range of versions cached locally and delete older versions if they exceed a quota.

5.21.4 Truly Distributed

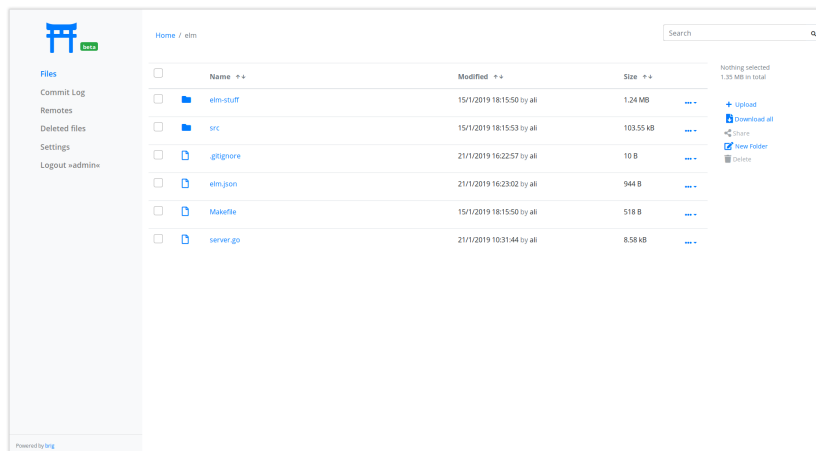
- No central server at all. All infrastructure is based on IPFS.
- Still, central architectures can be build with `brig`.
- Simple user identification and discovery.
- You do not store data you don't want to store. Pin what you need, fetch the rest from the network on request.

5.21.5 FUSE filesystem

- FUSE filesystem mirrors your data to a local directory.
- Allows your normal tools to work seamlessly with brig.
- Mounts can be persisted to stay where they are.
- Not high performance, but fast enough for daily usage.

5.21.6 Gateway and Web UI

- Gateway to share normal HTTP/S links with other users.
- Simple UI provided to execute the typical tasks without the need of a command line.
- User and right management included.



5.21.7 100% Open-Source

- Completely free software under the terms of the AGPL.
- Development driven by the community.
- Written in Go and Elm.

5.22 Comparison with other tools

When showing `brig` (or any other software in general) to someone 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 section 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 trade offs 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 is proprietary or has something else that does not fit your liking. I won't go into an exhaustive list of competitors, but more highlight the things that are special in `brig` and cannot be done easily in other systems.

I said *»competitors«* earlier, which is a silly term, since I don't see this as a competition. For me it's more about giving the user a choice and improving by adapting good ideas from other implementations. Let's list a few of those *»competitors«* to give you an impression about the place of `brig` in the world:

- **Syncthing**: Probably conceptually the nearest relative. Also a peer-to-peer based filesystem, but with its own protocol. Focus seems to be on ease-of-use and general high quality usability. Does not have strong versioning. Excellent tool and battle tested.
- **Resilio**: Proprietary solution based on BitTorrent. Seems to focus on performance and enterprise level resilience. Being proprietary is a show stopper for me.
- **Perkeep**: Not focused on files, but on storing personal *»objects«*. Would be probably more interesting as a backend for `brig`.
- **Uppspin**: A global name system that glues together filesystems and other data storage. Could be also a backend for `brig` and is not directly targeted to end users.
- **Bazil**: Basically `brig` minus IPFS. While apparently discontinued it seems to have a great deal of common features with `brig`. The same author also maintains the FUSE bindings of FUSE and his writeups helped me writing the FUSE implementation of `brig`. Thank you very much for your work @tv42!
- **Git LFS**: The large file storage extension to `git`. Similar to `brig`'s pinning in the sense that large files are replaced with links that will be fetched from a LFS server.
- **git annex**: Extension to `git` that tracks filenames and metadata instead of file content. Has a great deal of powerful features but can be a bit intimidating to the end users since it does not seem to focus much on usability. Features like the number of minimum copies a file must have before you can delete it are still on `brig`'s roadmap.

There are probably more. Some of these inspired quite a bit how `brig` looks today. So what are the unique features of `brig` that you would not get easily with other tools?

- **Pinning**: The fact that not all data needs to be on the same machine as the `brig` daemon opens up interesting possibilities. Also the ability of repinning is something I did not see in other tools.
- **Strong versioning of big files**: High level versioning that is comparable to `git`, but simplified and meant for whole-file version control (and not for individual diffs).

Of course there are drawbacks. Choosing `brig` currently means using software that is not in widespread use. It did not go through a security audit. It is by far not as efficient as other tools in all use cases. But many of the current hurdles are solvable and it's just a matter of time.

The best advice I can give you: Try it out and see if it fits your use case. If it doesn't I'm happy to hear from you and wish you all the best with another tool.

5.23 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 out 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.

5.23.1 Current state

The first real release (0.3.0 »Galloping Galapagos«) was released on the 7th December 2018. It includes all basic features and is working somewhat. The original goals were met:

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

For day-to-day use there are quite some other features that make `brig` easier to use and capable of forming a Dropbox-like backend out of several nodes.

There will be no stability guarantees before version 1.0.0.

5.23.2 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. *Done as of version 0.3.0.*
- **Config profiles:** Make it easy to configure `brig` in a way to serve either as thin client or as archival node. Archival nodes can be used in cases where a brig network spans over computers that lie in a different timezone. The archival node would accumulate all changes and repositories would see it as some sort of "blessed repository" which holds the latest and greatest state.
- **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. *Done as of version 0.4.0.*
- **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. *Done as of version 0.4.0 (see repinning)*
- **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. Also, the network performance is often very low and ridden by network errors and timeouts. This can be tackled since IPFS v0.4.19 supports an `--offline` switch to error out early if a file is not available locally.

- *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.

5.23.3 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.
- **Implement the encryption in IPFS:** Having the encryption/compression layer in brig effectively disables the usage of deduplication. This is unfortunate and could be mitigated by either implementing deduplication ourselves or moving to a block based encryption scheme.
- **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`.

5.24 How to contribute

Something we would be especially interested in are *experience reports*: We want you to try out the current state of the software and write down the following:

- Was it easy to get »brig« running?
- Was it easy to understand its 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.

Those reports should be posted as GitHub issue. They will help us to develop brig further in the “bigger picture”.

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

5.24.1 What to improve

We try to open a ticket for anything that can be worked right now

The following general improvements are of course also greatly appreciated:

- Bug reports & fixes.
- Documentation improvements.
- Writing more and better tests.

- Porting to other platforms.

5.24.2 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.

Here's a small checklist before publishing your pull request:

- Did you go `fmt` all code?
- Does your code style fit with the rest of the code base?
- Did you run `task lint`?
- Did you write tests if necessary?
- Did you consider if changes to the docs are necessary?
- Did you check if you need something to `CHANGELOG.md`?

Thank you for your contribution.