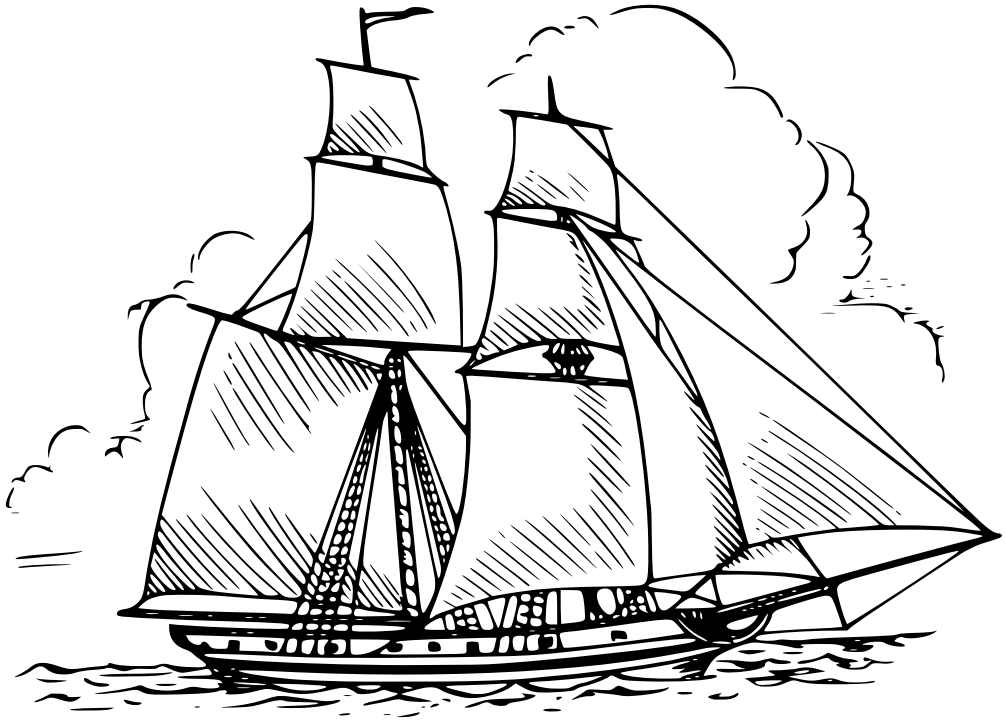
Hochschule Augsburg

University of Applied Sciences

Masterarbeit an der Fakultät Informatik



»brig«: Ein Werkzeug zur sicheren und verteilten Dateisynchronisation

Verfasser:
Christopher Pahl

Holzbachstraße 35 / *86152 Augsburg*
Telefon: +49 015 121340235
E-Mail: sahib@online.de

Prüfer:
Prof. Dr.-Ing. Schöler

Zweitkorrektor:
Prof. Dr. Hubert Högl

17. Oktober 2016

Abstract

A new open source tool for secure and distributed file synchronization called »brig«is presented. Most established ways to exchange files rely on a central instance, often exclusively controlled by a single company or have other design deficits. The shown alternative describes the usage of a distributed peer to peer network to achieve flexible, convenient and encrypted file exchange between any amount of users. The main novelty from a technical point of view is the complete decoupling of metadata from the actual data, combined with `git`–like version control of the former and encrypted compression of the latter. Special attention is given to the balance of usability and security in order to enable inexperienced users to use the software.

This paper describes the architecture and implementation of the software, but also reflects critically on how the featured ideas and techniques could be improved. In addition, there is a partner work at [1], which discusses »brig«from a security perspective.

Zusammenfassung

Es wird ein neuartiges, quelloffenes Werkzeug zur sicheren und verteilten Dateisynchronisation namens »brig«vorgestellt. Die meisten etablierten Möglichkeiten zum Austausch von Dateien verlassen sich auf eine zentrale Instanz, die oft von einem einzigen Unternehmen abhängig ist, oder die andere architetktonische Defizite haben. Der hier vorgestellte Gegenentwurf beschreibt die Verwendung eines verteilten Peer–to–Peer–Netzwerks, um flexibel, bequem und verschlüsselt Dateien zwischen einer beliebigen Menge an Nutzern auszutauschen. Technischen Neuwert hat dabei die vollständige Entkopplung der Metadaten von den eigentlichen Daten, kombiniert mit `git`–ähnlicher Versionskontrolle, sowie verschlüsstelter Kompression der Daten. Besonderer Wert wird dabei auf die Balance zwischen Usability und Sicherheit gelegt, um die Nutzung des Werkzeugs auch unerfahrenen Nutzern zu ermöglichen.

Die vorliegende Arbeit beschreibt die Architektur und Implementierung der Software, reflektiert aber auch kritisch wie die vorgestellten Ideen und Techniken erweitert und verbessert werden können. Zudem existiert eine Zwillings–Arbeit (vgl. [1]), welche die Sicherheitsmechanismen von »brig«genauer betrachtet und evaluiert.

Danksagung: Dank sei an folgende lebenden Personen und toten Dinge gerichtet: Die Gattungen Felis silvestris, Ursus arctos und Procyon lotor, sowie der Obergattung der Aves, Espressohersteller aller Länder und Freunde des weichgummierten Holzschlägers und Cellulose–Ellipsoids. Der Kapelle *Moonsorrow* gilt Dank für die kraftvolle und melancholische Musik, die mich die langen Stunden unterhalten hat.

Insbesondere gilt Herrn Prof. Dr. Schöler 23–facher Dank für alles. Meinen Eltern sei gedankt, dass sie das überraschte Wesen in die Welt gesetzt haben, welches meinen Namen trägt. Dass ich jetzt wohl mein Studium abschließen werde, kommt für mich übrigens genauso plötzlich wie für euch.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Einfache und sichere Dateisynchronisation ist trotz vieler Lösungsansätze im Jahre 2016 noch immer kein Standard. Versucht man beispielsweise eine Datei zwischen zwei Personen zu teilen (oder noch schwieriger: synchron zu halten), so kann man unter anderem zwischen den folgenden Möglichkeiten wählen:

- Übertragung mittels USB–Stick, Speicherkarte oder Ähnlichem.
- Übertragung über einen zentralen Dienst im lokalen Netz (wie FTP oder *ownCloud*[1]).
- Übertragung über das Internet mit zentralen Diensten wie Dropbox.
- Direkte Übertragung im Netzwerk mittels Protokollen wie `ssh`.
- …oder sehr häufig auch einfach via E–Mail.

Jede dieser Ansätze funktioniert auf seine Weise, doch ergeben sich in der Praxis meist sehr unterschiedliche Probleme. Bei E–Mails kann oft nur eine maximale Dateigröße übermittelt werden, die Übertragung von Dateien mittels `ssh` ist für die meisten Nutzer zu kompliziert und zentrale Dienste rufen einerseits Sicherheitsbedenken hervor, andererseits sind sie meist nur bedingt kostenlos und können unvermittelt ausfallen oder mittels Zensurmaßnahmen blockiert werden. Wie in Abb. 1.1 humoristisch gezeigt, muss also für jeden neuen Kontakt stets erst aufwendig der kleinste gemeinsame Nenner ausgehandelt werden.

1.1. Motivation

Zahlreiche Ansätze haben versucht, diese Probleme in der Praxis abzumildern oder zu lösen. Viele dieser Ansätze basieren nicht mehr auf einer zentralen Infrastruktur, sondern benutzen als Gegenentwurf einen dezentralen Ansatz. Dabei werden nicht alle Dateien an einem zentralen Punkt gespeichert, sondern können verteilt (ganz oder nur einzelne Blöcke einer Datei) im Netzwerk vorhanden sein. Dass dabei Dokumente auch durchaus doppelt oder öfters gespeichert werden dürfen, erhöht die Ausfallsicherheit und vermeidet den Flaschenhals zentraler Dienste, da der Ausfall einzelner Netzwerkknoten durch andere abgefangen werden kann. Anwender sind auch oft davon betroffen, dass viele Filehoster nur für einen bestimmten Zeitraum Dateien speichern. Ist dieser Zeitraum vorbei oder wird der Dienst eingestellt, entstehen vielfach tote Links. Hier könnte eine Lösung ansetzen, bei der die Dateien von jedem Interessenten gespiegelt werden und auch von diesen beziehbar sind. Dieser Gedanke entspricht dem *Permanent Web*[3].

Abseits der Dateisynchronisation konnte sich in anderen Bereichen sichere Open–Source–Software erfolgreich etablieren. Ein gutes Beispiel hierfür ist die Messenger–Anwendung *Signal*[4], welche sichere und einfache Kommunikation auf dem Smartphone ermöglicht. Vermutlich hat diese Software

[1] Eine Filehosting–Software für den Heimgebrauch; siehe auch https://owncloud.org
[2] Quelle: xkcd (https://xkcd.com/949)
[3] Siehe auch: https://motherboard.vice.com/read/the-interplanetary-file-system-wants-to-create-a-permanent-web
[4] Mehr Informationen unter: https://whispersystems.org

Abbildung 1.1.: Humorvolle Darstellung der Suche nach dem »kleinsten gemeinsamen Nenner«.[2]

nicht nur durch seine hohen Sicherheitsversprechen eine gewisse Verbreitung[5] erfahren, sondern weil es genauso leicht benutzbar und zugänglich war, wie die unsichereren Alternativen (wie *SMS* oder frühere Versionen von *WhatsApp*). Letztendlich führte dies sogar dazu, dass die von *Signal* genutzte Technik im deutlich populäreren *WhatsApp*–Messenger eingesetzt wurde. Gleichzeitig muss fairerweise gesagt werden, dass die gute Usability durch einige Vereinfachungen im Sicherheitsmodell erreicht wurde[6].

Erwähnenswert ist *Signal*, da auch viele Dateisynchronisationsdienste in der Praxis entweder an der Usability oder an den Sicherheitsanforderungen kranken, die insbesondere Unternehmen an eine solche Lösung stellen. Die vorliegende Arbeit stellt einen dezentralen Ansatz zur Dateisynchronisation vor, der eine *Balance zwischen Sicherheit, Usability und Effizienz* herstellt. Die hier vorgestellte und quelloffene Lösung trägt den Namen »`brig`«. Der aktuelle Quelltext findet sich auf der Hosting–Plattform *GitHub*[7].

1.2. Projektziel

Ziel des Projektes ist die Entwicklung einer sicheren, verteilten und versionierten Alternative zu Cloud–Storage Lösungen wie Dropbox, die sowohl für Unternehmen, als auch für Heimanwender nutzbar ist. Trotz der Prämisse, einfache Nutzbarkeit zu gewährleisten, wird auf Sicherheit sehr

[5] Zwischen 1 bis 5 Millionen Installationen im PlayStore (https://play.google.com/store/apps/details?id=org.thoughtcrime.securesms&hl=de)
[6] Mehr Informationen unter: https://de.wikipedia.org/wiki/Signal_(Software)#Kritik
[7] Offizielles GitHub Repository: http://github.com/disorganizer/brig

großen Wert gelegt.

Nutzbar soll das resultierende Produkt, neben dem Standardanwendungsfall der Dateisynchronisation, auch als Backup- bzw. Archivierungs–Lösung sein. Weiterhin kann es auch als verschlüsselter Daten–Safe oder als »Werkzeugkasten« für andere, verteilte Anwendungen dienen — wie beispielsweise aus dem Industrie–4.0–Umfeld.

Als weiteres Abgrenzungsmerkmal setzt `brig` nicht auf möglichst hohe Effizienz (wie es typischerweise verteilte Dateisysteme tun) sondern versucht möglichst generell anwendbar zu sein und über Netzwerkgrenzen hinweg zu funktionieren. Dadurch soll es zu einer Art »Standard« werden, auf den sich möglichst viele Anwender einigen können.

1.3. Der Name

Eine »Brigg« (englisch »brig«) ist ein kleines und wendiges Zweimaster–Segelschiff aus dem 18. Jahrhundert. Passend erschien den Autoren der Name einerseits, weil die Software flexibel »Güter« (in Form von Dateien) in der ganzen Welt verteilt, andererseits weil `brig` auf (Datei-)Strömen operiert.

Dass der Name ähnlich klingt und kurz ist wie `git`[8], ist kein Zufall. Das Versionsverwaltungssystem hat durch seine sehr flexible und dezentrale Arbeitsweise bestehende zentrale Alternativen wie `svn`[9] oder `cvs`[10] fast vollständig abgelöst. Zusätzlich ist der Gesamteinsatz von Versionsverwaltungssystemen durch die verhältnismäßig einfache Anwendung gestiegen. Die Autoren hoffen mit `brig` eine ähnlich flexible Lösung für »große« Dateien etablieren zu können.

1.4. Lizenz

Eine sicherheitskritische Lösung sollte den Nutzern die Möglichkeit geben zu validieren, wie die Sicherheitskonzepte implementiert sind. Aus diesem Grund und um eine freie Weiterentwicklung zu gewährleisten, wird die entwickelte Software unter die `AGPLv3` (*Affero General Public License, Version 3*[11]) gestellt. Diese stellt sicher, dass Verbesserungen am Projekt auch wieder in dieses zurückfließen müssen. Das Open–Source–Modell bietet aus unserer Sicht hierbei einige grundlegende Vorteile:

- Schnellere Verbreitung durch fehlende Kostenbarriere auf Nutzerseite.
- Kann von Nutzern und Unternehmen auf ihre Bedürfnissen angepasst werden.
- Transparenz in puncto Sicherheit (keine offensichtlichen Backdoors möglich).
- Fehlerkorrekturen und Weiterentwicklung durch die Community gewährleistet.

1.5. Gliederung der Arbeit

Diese Arbeit wird einen Überblick über die aktuelle Implementierung sowie die Techniken und Designentscheidungen dahinter geben, um sie anschließend kritisch zu reflektieren. Sicherheitsaspekte werden in dieser Arbeit nur oberflächlich angeschnitten, da Herr Piechula in seiner Arbeit

[8] Ein dezentrales Versionsverwaltungssystem; siehe auch: https://git-scm.com
[9] https://de.wikipedia.org/wiki/Apache_Subversion
[10] https://de.wikipedia.org/wiki/Concurrent_Versions_System
[11] Voller Lizenztext unter: http://www.gnu.org/licenses/agpl-3.0.de.html

»*Sicherheitskonzepte und Evaluation dezentraler Dateisynchronisationssysteme am Beispiel brig*«[1] die Sicherheitskonzepte der Software im Detail beleuchtet.

Die vorliegende Arbeit ist in drei größere logische Blöcke gegliedert:

- Kapitel 1 – Kapitel 4 (Einleitung, Stand der Technik, Anforderungen, Grundlagen): Eine Hinführung zum Thema Dateisynchronisation wird gegeben. Neben einer Analyse der Wettbewerber und Einsatzmöglichkeiten wird auch das nötige Grundlagenwissen vermittelt, um die nächsten Kapitel zu verstehen.
- Kapitel 5 – Kapitel 7 (Architektur, Implementierung, Usability): In diesen drei Kapiteln wird das technische Design des Prototypen erläutert und Begründungen zu den Designentscheidungen gegeben. Zuletzt wird noch ein Konzept für eine grafische Benutzeroberfläche vorgestellt.
- Kapitel 8 – Kapitel 9 (Evaluation, Fazit): Der aktuelle Prototyp wird auf Schwächen untersucht und mögliche Lösungen werden diskutiert. Zudem werden Möglichkeiten zur weiteren Entwicklung aufgezeigt.

Im Anhang A findet sich zudem ein Benutzerhandbuch, das losgekoppelt vom Rest gelesen werden kann und dazu dienen soll, einen praktischen Eindruck von der Implementierung zu bekommen. Es wird daher empfohlen, das Benutzerhandbuch frühzeitig zu lesen.

1.6. Über die Autoren der Software

Die Autoren sind zwei Master–Studenten an der Hochschule Augsburg, die von »Freier Software« begeistert sind. Momentan entwickeln wir `brig` im Rahmen unserer Masterarbeiten bei Prof. Dr.-Ing. Thorsten Schöler in der Distributed–Systems–Group[12] und wollen auch nach unserem Abschluss weiter daran arbeiten. Beide Autoren haben Erfahrung und Spaß daran, Open–Source–Software zu entwickeln und zu betreuen, was neben dem Eigennutzen einen großen Teil der Motivation ausmacht.

1.7. Konventionen

Es werden einige wenige typografische Konventionen im Textsatz vereinbart:

- Programmnamen werden `monospaced` geschrieben.
- Wichtige Aussagen werden *hervorgehoben*.
- Spezielle Ausdrücke und Eigennamen werden in »Chevrons« gesetzt.

Zudem werden die Namen *Alice*, *Bob* und manchmal *Charlie* verwendet, um Testnutzer zu kennzeichnen. Sofern nicht anders angegeben, kann angenommen werden, dass Abläufe aus Sicht von *Alice* geschildert werden. Die Grafiken in dieser Arbeit sind in englischer Sprache gehalten, da diese auch für die offizielle Dokumentation genutzt werden sollen.

[12] Siehe auch: http://dsg.hs-augsburg.de

2 Stand der Technik

In diesem Kapitel wird ein kurze Einführung zum Thema Peer–to–Peer–Netzwerke gegeben. Danach wird eine Einordnung der Arbeit zu den bisher existierenden Arbeiten zum Thema Dateisynchronisation gegeben. Im Anschluss wird `brig` zudem in Relation zu einigen auf dem Markt verfügbaren Produkten gesetzt. Darauf aufbauend wird von verschiedenen Perspektiven aus überlegt, welche Eigenschaften `brig` übernehmen kann und von wem und in welchem Rahmen die Software eingesetzt werden kann.

2.1. Peer–to–Peer Netzwerke

Bilden viele Rechner ein dezentrales Netzwerk, bei dem jeder Rechner (ein »Peer«) die gleichen Rechte besitzt und die gleichen Aktionen ausführt wie jeder andere, so wird dieses Netz ein *Peer–to–Peer–Netzwerk* genannt (kurz *P2P–Netzwerk*, vgl. auch [2], S. 4 ff.). Statt Verbindungen über einen Mittelsmann aufzubauen, kommunizieren die einzelnen Peers für gewöhnlich direkt miteinander. Jeder Knoten des Netzwerks kann Anfragen an andere Knoten richten, trägt aber selbst etwas bei indem er selbst Anfragen beantwortet. Im Client–Server–Modell entspricht ein Peer also sowohl Server als auch Client (siehe auch Abb. 2.1).

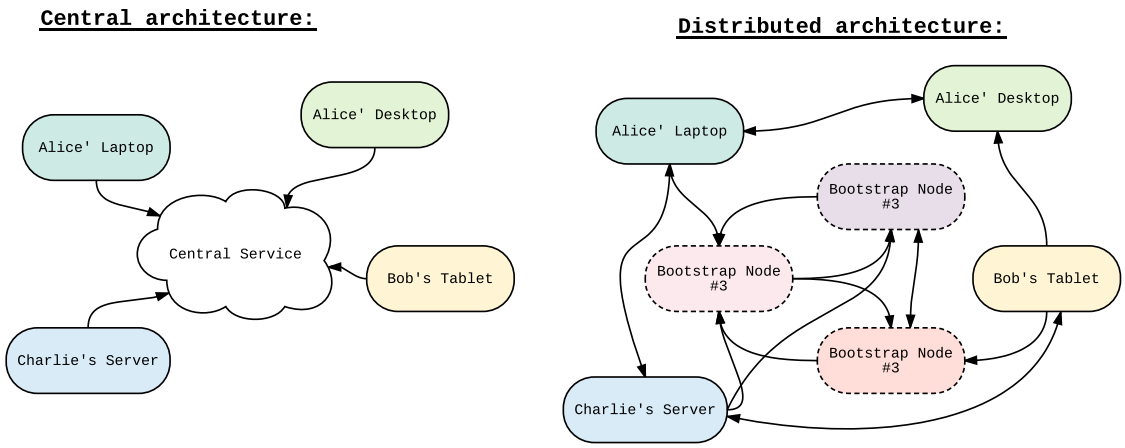


Abbildung 2.1.: Anschaulicher Unterschied zwischen zentralen und verteilten Systemen.

Im alltäglichen Gebrauch der meisten »Otto–Normal–Nutzer« scheinen P2P–Netzwerke derzeit eine eher untergeordnete Rolle zu spielen. Die bekanntesten und populärsten P2P–Netzwerke sind vermutlich das BitTorrent- und Skype–Protokoll (vgl. [2], S. 232 ff. und S. 2). Darüber hinaus gibt es auch viele sehr große Filesharing–Netzwerke, wie Gnutella (vgl. auch [2], S. 57 ff.). Gemeinsam ist allen, dass sie als sogenanntes *Overlay–Netzwerk*[1] über das Internet gelegt werden und dessen existierende Infrastruktur wiederverwenden.

[1] Siehe auch: https://de.wikipedia.org/wiki/Overlay-Netz

2.1.1. Zugrundeliegende Technik

Die meisten Dienste im Internet basieren hingegen auf dem Client–Server–Modell, bei dem viele anonyme Clients eine Anfrage an einen zentralen Server stellen. Dieser muss mit der steigenden Anzahl an Clients skalieren, indem er typischerweise mehr Prozessorleistung und Bandbreite zur Verfügung stellt. Dieses Modell passt auf viele heterogene Anwendungsfälle, wo Client und Server grundverschiedene Rollen zugeordnet sind (Beispiel: Dienstleiter und Kunde). Eine weitere Eigenschaft, ist dass das Client–Server–Modell kein Problem mit dem sogenannten *NAT-Traversal* hat.

NAT steht dabei für *Network Address Translation* (dt. Netzwerkadressübersetzung, siehe auch [2], S. 47 ff.) und ist eine Technik, um zwischen einer öffentlichen und mehreren lokalen IP–Adressen zu vermitteln. Es wird aufgrund der Knappheit von IPv4 sehr häufig eingesetzt, um einem Heim- oder Unternehmensnetzwerk eine einzige IP-Adresse nach Außen zu geben, die über bestimmte Ports dann den Verkehr auf die jeweiligen lokalen Adressen übersetzt. Der Nachteil in Bezug auf P2P–Netzwerke ist dabei, dass die Rechner hinter einem *NAT* nicht direkt erreichbar sind. Client–Server–Anwendungen haben damit kein Problem, da der »anonyme« Client die Verbindung zum »wohlbekannten« Server selbstständig aufbaut. Bei einer P2P–Kommunikation hingegen, muss eine Verbindung in beide Richtungen möglich sein — und das möglicherweise sogar über mehrere *NATs* hinweg. Die Umgehung dieser Grenzen ist in der Literatur als *NAT Traversal* bekannt. Eine populäre Technik ist dabei das UDP–Hole–Punching[2]. Dabei wird, grob erklärt, ein beiden Parteien bekannter Mittelsmann herangezogen, über den die eigentliche, direkte Verbindung aufgebaut wird. Eine technische Notwendigkeit dabei ist die Verwendung von *UDP* anstatt *TCP*.

Typischerweise ist dieser Mittelsmann ein sogenannter *Bootstrap-Knoten*. Dieser ist innerhalb eines P2P–Netzwerks einer von mehreren wohlbekannten Knoten, zu dem sich neue Netzwerkteilnehmer verbinden, um von ihm an weitere Teilnehmer vermittelt zu werden. Der Boostrap–Knoten führt aber normalerweise das gleiche Programm aus, wie jeder andere, ist aber vertrauenswürdiger. Bemerkenswert ist, dass sich keine zentrale Instanz um die Koordination des Datenflusses im Netzwerk kümmern muss. Die Grundlage für die Koordination bildet dabei die *Distributed Hashtable (DHT, vgl. [2], S. 63 ff.)* Diese Datenstruktur bildet sich durch den Zusammenschluss vieler Rechner und nutzt eine *Hashfunktion*[3], um für einen bestimmten Datensatz zu entscheiden, welche Knoten (mindestens aber einer) im Netzwerk für diesen Datensatz zuständig sind. Ist ein Teilnehmer an einem Datensatz interessiert, so muss er nur die Prüfsumme desselben kennen, um zu wissen von welchem Teilnehmer er den Datensatz beziehen kann. Jeder Knoten verwaltet dabei einen bestimmten Wertebereich der Prüfsummenfunktion und ist für diese Prüfsummen zuständig. Werden neue Knoten hinzugefügt oder andere verlassen das Netz, werden die Wertebereiche neu verteilt.

2.1.2. Dateisynchronisation in P2P–Netzwerken

In diesem Kontext meint der Begriff »Synchronisation« das Zusammenführen der Dateistände mehrerer Netzwerkteilnehmer. Typischerweise nutzen viele Nutzer heutzutage dafür einen zentralen Dienst. Dieser hält einen Dateistand vor, der von allen Teilnehmern als Referenz angesehen wird.

[2] Siehe auch: https://en.wikipedia.org/wiki/UDP_hole_punching

[3] Bildet einen Datensatz beliebiger Länge auf eine kurze Prüfsumme mit fixer Länge ab. Eine Rückrechnung von der Prüfsumme zum ursprünglichen Datensatz ist theoretisch möglich, aber extrem rechenaufwendig. Kleine Änderungen der Eingabe, erzeugen eine gänzlich andere Prüfsumme. Siehe auch: https://de.wikipedia.org/wiki/Hashfunktion

Ändert ein Teilnehmer seinen Stand, so wird die Änderung zum zentralen Server übertragen und erreicht so auch alle anderen Teilnehmer.

In vielen Fällen skalieren aber solche Client–Server Anwendungen bei weitem schlechter als verteilte Anwendungen. Man stelle sich einen Vorlesungssaal mit 50 Studenten vor, die ein Festplattenimage (Größe: 5 Gigabyte) aus dem Internet herunterladen sollen. Bei einer Client–Server Anwendung werden hier 50 Verbindungen zu einem zentralen Server (beispielsweise Dropbox) geöffnet. Der Server muss nun 50 Verbindungen gleichzeitig bearbeiten und muss eine entsprechende Bandbreite zur Verfügung stellen. Bei kleineren Diensten kann dies bereits der Flaschenhals sein, teilweise kann aber auch die Bandbreite auf Seiten des Empfängers limitiert sein. Fällt der zentrale Server aus (»Single–Point–of–Failure«), so kann kein neuer Nutzer mehr das Festplattenimage empfangen.

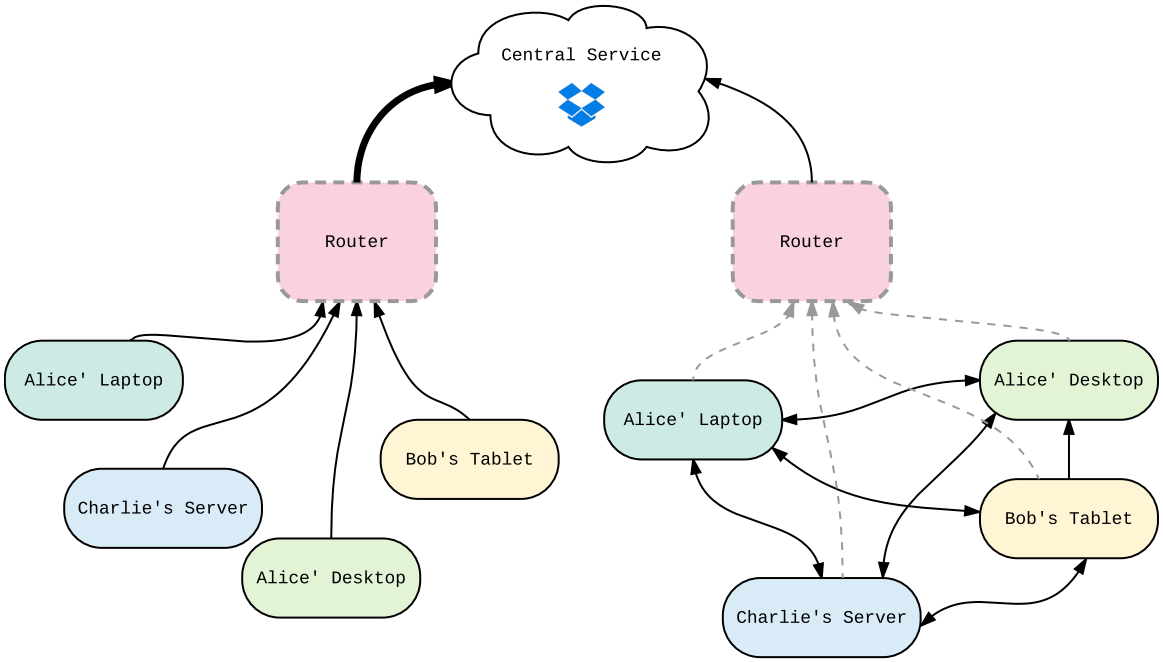


Abbildung 2.2.: Veranschaulichung der Netzwerklast bei zentralen und dezentralen Systemen.

Bilden die Rechner der Studenten ein verteiltes Netzwerk, so genügt es wenn nur ein Rechner einen Teil der Datei hat. Diesen Teil kann er im lokalen Netz anderen Teilnehmern wieder anbieten und sich Teile der Datei besorgen, die er selbst noch nicht hat. So muss in der Theorie die Datei nur maximal einmal vom zentralen Server übertragen werden. In diesem etwas konstruierten[4] Beispiel würde im dezentralen Netzwerk die Datei also bis zu 50-mal schneller verteilt werden, als im zentralen Anwendungsfall. Fällt der zentrale Server aus nachdem die Datei bereits einmal komplett heruntergeladen wurde, so werden die bereits existierenden Teile von den jeweiligen Teilnehmern weiter angeboten. Abb. 2.2 veranschaulicht diesen Zusammenhang noch einmal.

Dezentrale Netzwerke eignen sich sehr gut um Dateien auszutauschen, da ganze Dateien in kleine Blöcke unterteilt werden können. Diese können dann von interessierten Knoten vorgehalten und weitergegeben werden. Protokolle wie *BitTorrent* haben das Problem, dass ein Block nur solange

[4] Typischerweise sorgen auch vorgeschaltete *Caching Proxies* wie `Squid` (https://de.wikipedia.org/wiki/Squid) dafür, dass Dateien nicht zigmal heruntergeladen werden.

verfügbar ist, solange es Teilnehmer gibt, die diesen Block anbieten. Prinzipiell hat auch `brig` dieses Problem, doch besteht ein `brig`–Netzwerk nur aus den Teilnehmern, die einen gemeinsamen Dateistand synchronisieren wollen. Daher kann angenommen werden, dass alle darin enthaltenen Dateien von mindestens einem Teilnehmer angeboten werden können.

2.2. Ähnliche Arbeiten

Es gibt viele unterschiedliche wissenschaftliche Arbeiten rund um das Thema der Dateiverteilung in P2P–Netzwerken. Die meisten Arbeiten scheinen sich mehr auf das Thema des Dateiaustausches an sich zu konzentrieren und weniger auf das Thema der Dateisynchronisation, wo eine Menge von Dateien auf dem selben Stand gehalten werden muss. Die dazu vorhandenen Arbeiten legen ihren Fokus dabei meist auf die Untersuchung und Implementierung verteilter Dateisysteme, die sehr ähnliche Probleme lösen müssen, aber mehr auf Effizienz als auf Einfachheit Wert legen.

Stellvertretend für eine solche Arbeit soll hier die Dissertation von Julien Quintard *»Towards a worldwide storage infrastructure«*[3] genannt werden. In dieser wird die Implementierung und die Konzepte hinter dem verteilten Dateisystem *Infinit* vorgestellt. Obwohl der Fokus hier auf Effizienz liegt, hat *Infinit* einige auffällige Ähnlichkeiten mit den Zielen von `brig`:

- Weltumspannendes P2P-Netzwerk als Grundlage.
- Nutzung von FUSE[5] als Frontend zum Nutzer.
- Verschlüsselte Speicherung der Daten.
- Eingebaute Deduplizierung.
- Eine Versionierung der Dateien ist geplant.

Der Hauptunterschied ist allerdings die Zielgruppe. Während das bei `brig` der »Otto–Normal–Nutzer« als kleinster Nenner ist, so ist *Infinit* auf Entwickler und Administratoren ausgelegt und leider nur teilweise quelloffen,[6] also keine »Free Open Source Software« *(FOSS)*.

Eine sehr detaillierte Gegenüberstellung vieler Produkte rund um das Thema Dateisynchronisation findet sich in der Dokumentation von `inifinit`[7].

Es gibt eine Reihe nicht–kommerzieller Projekte, die teilweise eine ähnliche Ausrichtung wie `brig` haben und daher mindestens eine Erwähnung verdienen. Im Folgenden werden die Ähnlichkeiten zu `brig` genannt:

bazil:[8] Ein Werkzeug um Dateien verschlüsselt und dezentral zu verteilen. In seinen Zielen ist es sehr ähnlich zu `brig`, besonders da es ebenfalls ein FUSE–Dateisystem implementiert[9]. Es ist eher an technisch versierte Nutzer gerichtet und momentan noch nicht für den Produktivbetrieb geeignet. Zu diesem Zeitpunkt funktioniert es nur lokal auf einem System ohne mit anderen Knoten kommunizieren zu können.

[5] Eine Technik, um ein Dateisystem im Userspace zu implementieren. Dem Nutzer kann dadurch ein normaler Ordner mit beliebigen Daten als Dateien präsentiert werden.
[6] Siehe auch: https://infinit.sh/open-source
[7] Beispielsweise mit Dropbox: https://infinit.sh/documentation/comparison/dropbox
[8] Siehe auch: https://bazil.org
[9] Der Entwickler von `bazil` Tommi Virtanen betreut auch dankenswerterweise die FUSE–Bindings für *Go*, die auch `brig` nutzt.

Tahoe-LAFS:[10] Ein verteiltes Dateisystem, welches Dateien auf eine Menge an Rechnern möglichst ausfallsicher verteilen kann, selbst wenn einzelne Rechner ausfallen. Es richtet sich tendenziell an Administratoren und technisch versierte Nutzer, die eine große Menge an Daten sicher lagern wollen. Ähnliche Produkte in diesem Bereich gibt es mit *XtreemFs*[11], *LizardFs*[12] und *MooseFs*[13] mit jeweils unterschiedlichen Schwerpunkten.

restic:[14] Ein in *Go* geschriebenes Backup–Programm. Es synchronisiert zwar keine Dateien über das Netzwerk, setzt aber eine Versionsverwaltung mittels *Snapshots* um. Zudem verschlüsselt es alle ihm bekannten Dateien in einem *Repository* und gewährleistet mittels eines speziellen Dateiformats deren Integrität. `brig` verwendet analog zu `restic` (und `git`) den Begriff *Repository* für den Ordner, in dem es seine Daten ablegt.

2.3. Wissenschaftliche Lücke

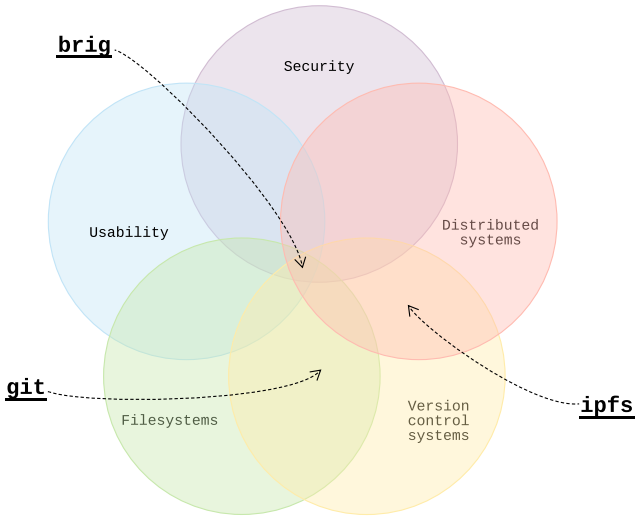


Abbildung 2.3.: Die Neuerung von `brig` liegt in der Zusammenführung vieler Teildisziplinen.

Die wissenschaftliche Neuerung der vorliegenden Arbeit ist die Zusammenführung vieler wissenschaftlicher Teildisziplinen, die es nach Wissen des Autors vorher noch nicht in dieser Kombination gab. Dabei werden viele bestehende Ideen und Konzepte genommen, um sie in einer Software zu vereinen, die ein versioniertes und verteiltes Dateisystem implementiert. Dieses soll nicht nur »sicher« (im weitesten Sinne, siehe [1] für eine Begriffseinordnung) sein, sondern auch für einen Großteil der Anwender benutzbar sein.

Im Konkreten besteht die Neuerung hauptsächlich aus der Kombination folgender Punkte:

- Eine Erweiterung des Datenmodells von `git`, welches Metadaten von den eigentlichen Daten trennt, leere Verzeichnisse sowie umbenannte Pfade nativ unterstützt und eine eigene Historie pro Datei verwaltet.

[10] Siehe auch: https://tahoe-lafs.org/trac/tahoe-lafs
[11] Siehe auch: http://www.xtreemfs.org
[12] Siehe auch: https://lizardfs.com/
[13] Siehe auch: http://moosefs.org/
[14] Siehe auch: https://restic.github.io

- Die Möglichkeit nur die Metadaten zu synchronisieren und die eigentlichen Daten dynamisch nachzuladen und nach Anwendungsfall zu »pinnen«. Dateien mit einem *Pin* werden dabei auf dem lokalen Rechner gespeichert, Dateien ohne Pin dürfen falls nötig wieder gelöscht werden.
- Ein Containerformat für Verschlüsselung (ähnlich dem Secretbox der freien NaCl[15] Bibliothek), welches effizienten wahlfreien Zugriff erlaubt und eine Austauschbarkeit des Algorithmus gewährleistet.
- Ein Containerformat zur Kompression, welches blockbasierten, wahlfreien Zugriff und den Einsatz verschiedener Algorithmen erlaubt.
- Ein Konzept und Implementierung zur dezentralen Benutzerverwaltung, ohne dass ein Nutzer dabei explizit registriert werden muss.
- Verschiedene Ansätze um die Usability zu verbessern ohne die Sicherheit einzuschränken (siehe Kapitel 7).

2.4. Markt und Wettbewerber

Bereits ein Blick auf Wikipedia[16] zeigt, dass der momentane Markt an Dateisynchronisationssoftware sehr unübersichtlich ist. Ein näherer Blick zeigt, dass die dortigen Softwareprojekte oft nur in Teilaspekten gut funktionieren und teilweise auch mit architektonischen Problemen behaftet sind.

Im Folgenden wird eine unvollständige Übersicht über bekannte Dateisynchronisationsprogramme gegeben. Davon stehen nicht alle in Konkurrenz zu `brig`, sind aber zumindest aus Anwendersicht ähnlich und sollten daher kurz aus dieser Perspektive verglichen werden.

2.4.1. Dropbox + Boxcryptor

Dropbox (siehe Abb. 2.4) ist der vermutlich bekannteste und am weitesten verbreitete zentrale Dienst zur Dateisynchronisation. Verschlüsselung kann man mit Tools wie dem freien `encfs`[17] oder dem etwas umfangreicheren, proprietären `boxcryptor` nachrüsten. Was das Backend genau tut ist leider das Geheimnis von Dropbox — es ist nicht Open–Source. Mehr Details liefert die Arbeit von Herrn Piechula[1].

Die Server von Dropbox stehen in den Vereinigten Staaten von Amerika, was spätestens seit den Snowden–Enthüllungen Besorgnis um die Sicherheit der Daten weckt. Wie oben erwähnt, kann diese Problematik durch die Verschlüsselungssoftware `boxcryptor` abgemildert werden. Diese kostet aber zusätzlich und benötigt noch einen zusätzlichen zentralen Keyserver[18]. Ein weiterer Nachteil ist hier die Abhängigkeit von der Verfügbarkeit des Dienstes.

Technisch nachteilhaft bei vielen zentralen Diensten ist, dass die Datei »über den Pazifik« hinweg synchronisiert werden muss, nur um möglicherweise auf dem Arbeitsrechner »nebenan« anzukommen. Dropbox hat hier nachgerüstet, indem es nach Möglichkeit direkt über LAN synchronisiert[19]. Nichtsdestotrotz können Kunden nicht mehr synchronisieren, wenn der zentrale Dienst ausgefallen ist oder den Dienst eingestellt hat.

[15] Mehr Informationen unter https://nacl.cr.yp.to/secretbox.html
[16] Siehe https://en.wikipedia.org/wiki/Comparison_of_file_synchronization_software
[17] Mehr Informationen unter https://de.wikipedia.org/wiki/EncFS
[18] Mehr Informationen zum Keyserver unter https://www.boxcryptor.com/de/technischer-/%C3/%BCberblick/#anc09
[19] https://www.dropbox.com/de/help/137

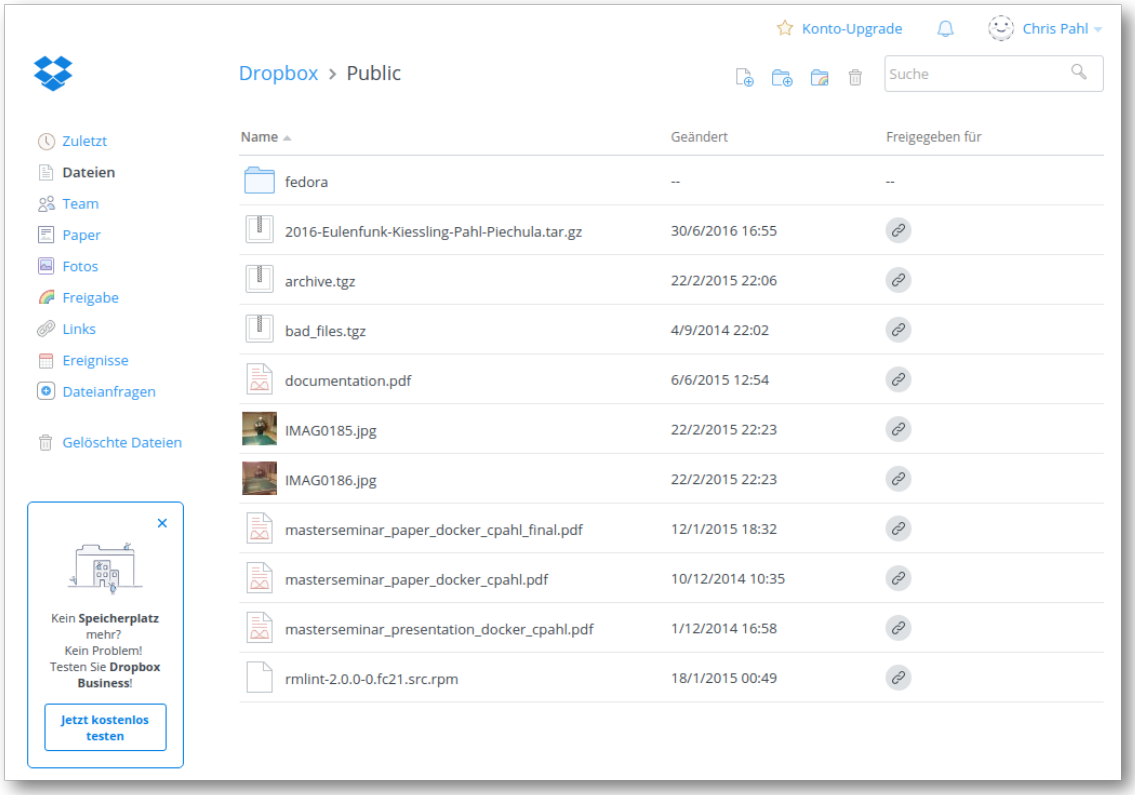


Abbildung 2.4.: Screenshot eines Dropbox–Accounts.

2.4.2. ownCloud / Nextcloud

Eine Alternative zu einem von einem Unternehmen bereitgestellten zentralen Dienst, ist die Nutzung einer eigenen »Private Cloud« mithilfe der Open–Source Lösung `ownCloud` (siehe Abb. 2.5, beziehungsweise dessen Fork `Nextcloud`). Nutzer installieren auf ihren Servern selbst eine `ownCloud`–Instanz und stellen ausreichend Speicherplatz bereit. Vorteilhaft ist also, dass die Daten auf den eigenen Servern liegen. Nachteilig hingegen, dass das zentrale Modell von Dropbox lediglich auf eigene Server übertragen wird. Einerseits ist `ownCloud` nicht so stark wie `brig` auf Sicherheit fokussiert, andererseits ist die Installation eines Serversystems für viele Nutzer eine große Hürde und somit zumindest für den Heimanwender nicht praktikabel.

2.4.3. Syncthing

Das 2013 veröffentlichte quelloffene `syncthing` (siehe Abb. 2.6) versucht diese zentrale Instanz zu vermeiden, indem die Daten jeweils von Teilnehmer zu Teilnehmer übertragen werden. Die Dateien werden in einem speziellen Ordner gelegt, der von `syncthing` überwacht wird. Nach der Installation wird eine einzigartige Client–ID generiert. Über eine Weboberfläche oder eine native Desktopanwendung kann konfiguriert werden, mit wem dieser Ordner geteilt werden soll, indem die Client–ID eines anderen Teilnehmers eingegeben wird.

Es ist allerdings kein vollständiges Peer–to–peer–Netzwerk: Geteilte Dateien liegen immer als vollständige Kopie bei allen Teilnehmern, welche die Datei haben. Alternativ ist nur die selektive Syn-

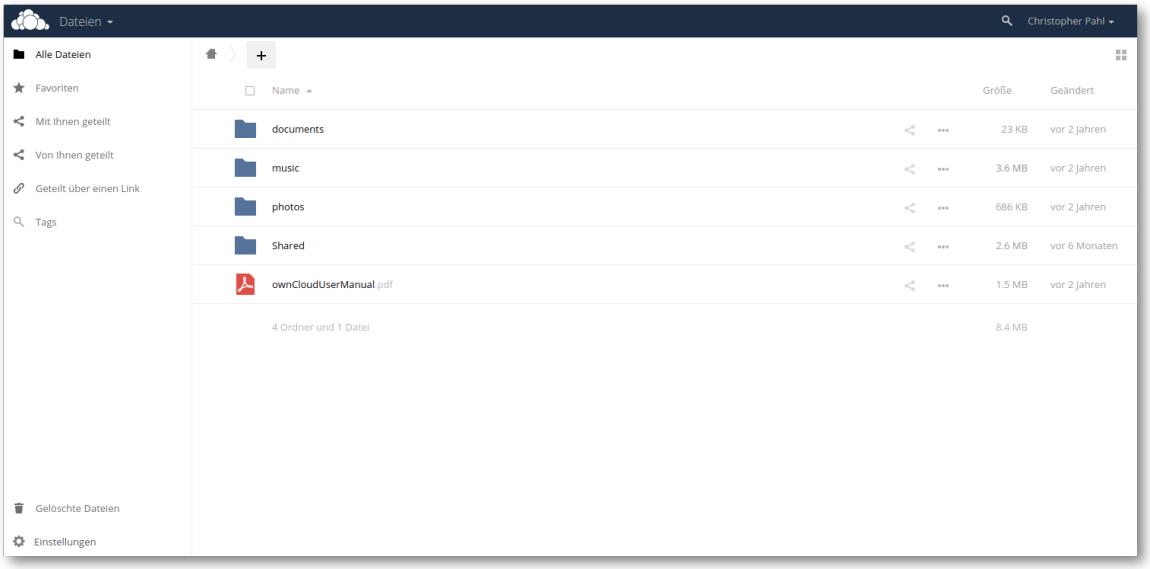


Abbildung 2.5.: Screenshot der ownCloud–Weboberfläche.

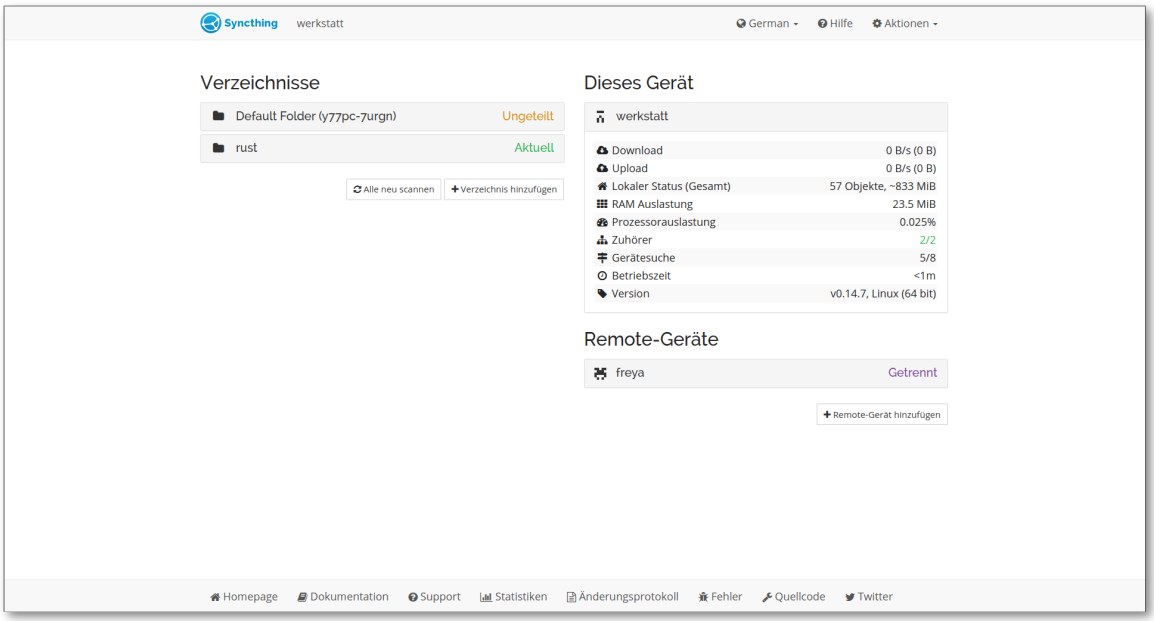


Abbildung 2.6.: Screenshot der Syncthing–Weboberfläche.

chronisation bestimmter Dateien möglich. Zwischen den Teilnehmern wird ein Protokoll mit dem Namen *Block Exchange Protocol*[4] etabliert. Dieses sorgt für eine sichere, differentielle und blockweise Übertragung.

Praktisch ist auch, dass `syncthing`–Instanzen mittels eines zentralen Discovery–Servers entdeckt werden. Nachteilig ist aber die fehlende Benutzerverwaltung: Man kann nicht festlegen von welchen Nutzern man Änderungen empfangen will und von welchen nicht. Eingesetzt wird `syncthing` zwar auch gerne von technisch versierten Nutzern, doch existiert auch für Neulinge ausreichend Dokumentation.

2.4.4. `resilio`

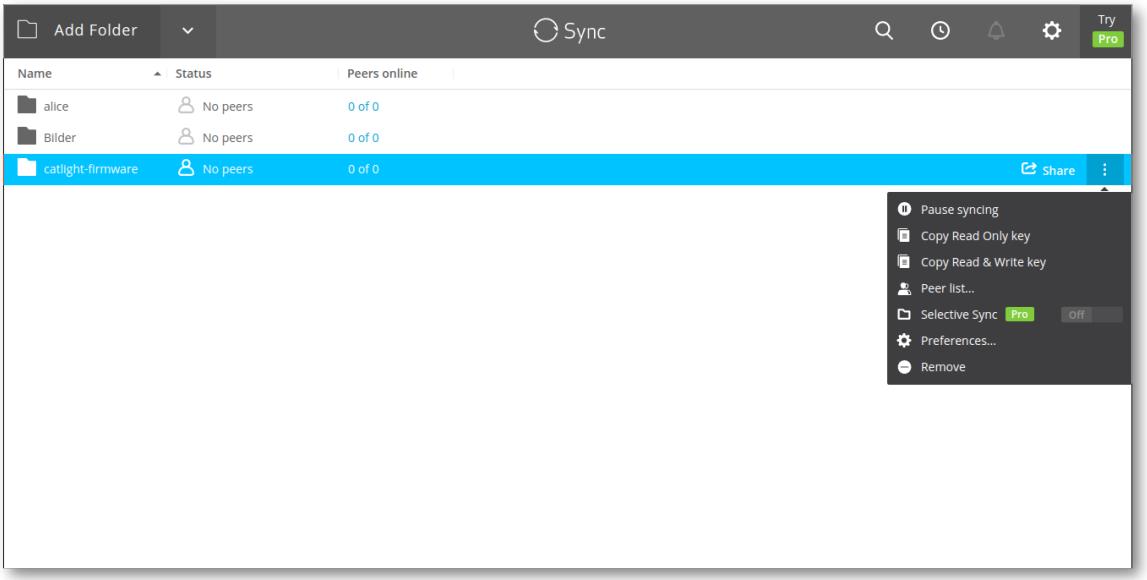


Abbildung 2.7.: Screenshot der `resilio`–Weboberfläche.

Das kommerzielle und proprietäre `resilio` (früher *Bittorrent Sync*) nutzt eine Modifikation[20] des bekannten und freien BitTorrent Protokoll zur Übertragung. Vom Feature–Umfang ist es in etwa vergleichbar mit `syncthing`. Das Anlegen von verschlüsselten Repositories ist möglich.

Genauere Aussagen über die verwendete Technik kann man aufgrund der geschlossenen Natur des Programms und der eher vagen Werbeprosa nicht treffen. Ähnlich zu `syncthing` ist allerdings, dass eine Versionsverwaltung nur mittels eines »Archivordners« vorhanden ist. Gelöschte Dateien werden in diesen Ordner verschoben und können von dort wiederhergestellt werden. Etwas mehr Details liefert der Vergleich des *Infinit*–Projekts.[21]

2.4.5. `git-annex`

Das 2010 erstmals von Joey Hess veröffentlichte `git-annex`[23] geht in vielerlei Hinsicht einen anderen Weg als die oben genannten Werkzeuge. Einerseits ist es in der funktionalen Programmiersprache

[20] Siehe auch: http://blog.bittorrent.com/2016/03/17/%CE%BCtp2-the-evolution-of-an-enterprise-grade-protocol
[21] https://infinit.sh/documentation/comparison/bsync
[22] Bildquelle: http://code.178.is/git-annex-is-magic/git-annex-assistant2.png
[23] Webpräsenz: https://git-annex.branchable.com/

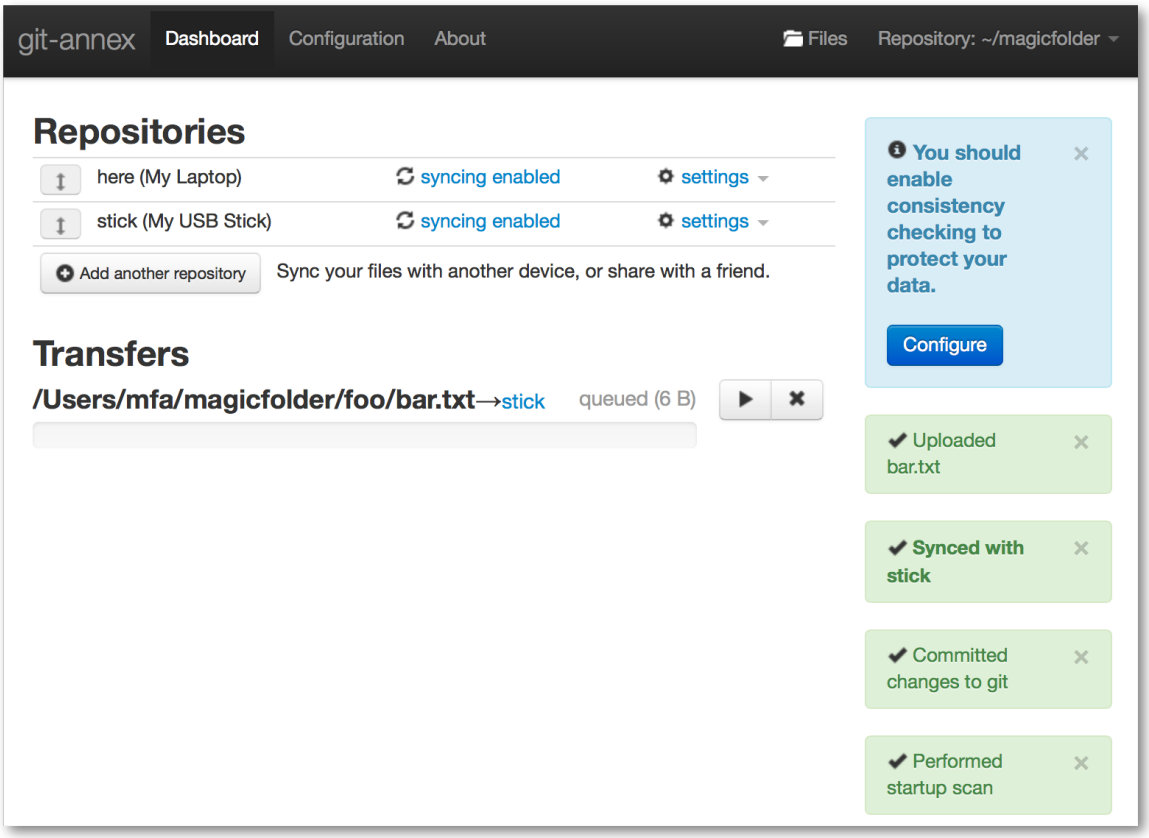


Abbildung 2.8.: Screenshot des `git-annex`–Assistenten[22].

Haskell geschrieben, andererseits nutzt es intern das Versionsverwaltungssystem `git`[5], um die Metadaten zu den Dateien abzuspeichern, die es verwaltet. Auch werden Dateien standardmäßig nicht automatisch synchronisiert, hier ist die Grundidee die Dateien selbst zu »pushen«, beziehungsweise zu »pullen«.

Dieser »Do-it-yourself« Ansatz ist sehr nützlich, um `git-annex` als Teil der eigenen Anwendung einzusetzen. Für den alltäglichen Gebrauch scheint es aber selbst für erfahrene Anwender zu kompliziert, um es praktikabel einzusetzen.

Trotzdem sollen zwei interessante Features genannt werden, welche auch für `brig` interessant sind:

- *Special Remotes:* »Datenablagen« bei denen `git-annex` nicht installiert sein muss. Damit können beliebige Cloud–Dienste als Speicher genutzt werden.
- *N-Copies:* Von wichtigen Dateien kann `git-annex` bis zu N Kopien speichern. Versucht man eine Kopie zu löschen, so verweigert `git-annex` dies.

2.4.6. Weitere Alternativen

Obwohl `brig` eine gewisse Ähnlichkeit mit verteilten Dateisystemen, wie *GlusterFS* hat, wurden diese in der Übersicht weggelassen — einerseits aus Gründen der Übersicht, andererseits weil diese andere Ziele verfolgen und von Heimanwendern kaum genutzt werden. Zudem ist der Vollständigkeit halber

auch *OpenPGP*[24] zu nennen, was viele Nutzer zum Verschlüsseln von E-Mails benutzen. Aber auch hier ist der größte Nachteil die für den Otto–Normal–Nutzer schwierige Einrichtung und Benutzung. Auch das freie Projekt `librevault`[25] wurde im Vergleich ausgelassen, da es sich noch im Alpha–Stadium befindet und bei einem Test reproduzierbar abstürzte.

2.4.7. Zusammenfassung

In Tabelle 2.1 und Tabelle 2.2 findet sich zusammenfassend eine Übersicht, mit den wichtigsten Unterscheidungsmerkmalen. Die Bewertung ist in Punkten wie *»Einfach nutzbar«* subjektiver Natur.

Tabelle 2.1.: Vergleich der Software aus technischer Sicht

	Dezentral	**Verschlüsselung im Client**	**Versionierung**
Dropbox/Boxcryptor	✗	Mit Boxcryptor	Rudimentär
`ownCloud`	✗	✗	Rudimentär
`syncthing`	✓	✓	Archivordner
`resilio`	✓	✓	Archivordner
`git-annex`	✓	✗	✓
`infinit`	✓	✗	✗
`brig` *(Prototyp)*	✓	✓	✓
`brig` *(Ziel)*	✓	✓	✓

Tabelle 2.2.: Vergleich der Software aus Nutzersicht

	FOSS	**Einfach nutzbar**	**Einfache Installation**
Dropbox/Boxcryptor	✗	✓	✓
`ownCloud`	✓	✓	✗
`syncthing`	✓	✓	✓
`resilio`	✗	✓	✓
`infinit`	✗	✗	✓
`git-annex`	✓	✗	✗
`brig` *(Prototyp)*	✓	✗	Auf Linux
`brig` *(Ziel)*	✓	✓	✓

Abschließend kann man sagen, dass `syncthing` dem Gedanken hinter `brig` am nächsten kommt. Der Hauptunterschied ist, dass die Basis hinter `brig` ein volles P2P–Netzwerk namens `ipfs` ist (dazu später mehr). Wie in den nächsten Kapiteln ersichtlich ist, eröffnet dieser Unterbau eine Reihe von Möglichkeiten, die `syncthing` nicht bieten kann[26].

[24] Siehe auch: https://de.wikipedia.org/wiki/OpenPGP
[25] Mehr Informationen hier: https://librevault.com
[26] Beispielsweise `git`–ähnliche Versionierung und die Möglichkeit auf alle Daten zuzugreifen, aber nur wenige physikalisch zu speichern. Hierzu später mehr.

2.5. Zielgruppen

Die primären Zielgruppen von `brig` sind Unternehmen und Heimanwender. Aufgrund der starken Ende-zu-Ende Verschlüsselung ist `brig` allerdings auch insbesondere für Berufsgruppen attraktiv, bei denen eine hohe Diskretion bezüglich Datenschutz gewahrt werden muss. Hier wären in erster Linie Journalisten, Anwälte, Ärzte mit Schweigepflicht und auch Aktivisten und politisch verfolgte Minderheiten zu nennen.

Unternehmen: Unternehmen können `brig` nutzen, um ihre Daten und Dokumente intern zu verwalten und zwischen Mitarbeitern zu teilen. Besonders sicherheitskritische Dateien entgehen so der Lagerung in Cloud–Services oder der Gefahr von Kopien auf potenziell unsicheren Mitarbeiter–Endgeräten. Größere Unternehmen verwalten dabei oft ein Rechenzentrum in dem firmeninterne Dokumente gespeichert werden. Von den Nutzern werden diese dann meist mittels Diensten wie *ownCloud*[27] »händisch« heruntergeladen. In diesem Fall könnte man `brig` im Rechenzentrum und auf allen Endgeräten installieren. Das Rechenzentrum würde die Datei mit tiefer Versionierung vorhalten. Endanwender würden alle Daten sehen, aber auf ihrem Gerät nur die Daten tatsächlich speichern, die sie auch benötigen. Hat beispielsweise ein Kollege im selben Büro die Datei bereits vorliegen, kann `brig` diese dann direkt transparent vom Endgerät des Kollegen holen. Das »intelligente Routing« erlaubt den Einsatz von `brig` auf Smartphones, Tablets und anderen speicherplatzlimitierten Geräten. Nutzer, die eine physikalische Kopie der Datei auf ihrem Gerät haben wollen, können das entsprechende Dokument »pinnen«. Ist ein Außendienstmitarbeiter beispielsweise im Zug unterwegs, kann er vorher ein benötigtes Dokument pinnen, damit `brig` die Datei persistent verfügbar macht.

Privatanwender: Privatanwender können `brig` für ihren Datenbestand aus Fotos, Filmen, Musik und sonstigen Dokumenten nutzen. Ein typischer Anwendungsfall wäre dabei ein Network–Attached-Storage–Server (NAS), der alle Dateien mit niedriger Versionierung speichert. Endgeräte, wie Laptops und Smartphones, würden dann ebenfalls `brig` nutzen, aber mit deutlich geringeren Speicherquotas (maximales Speicherlimit), so dass nur die aktuell benötigten Dateien physikalisch auf dem Gerät vorhanden sind. Die anderen Dateien lagern im Netz und können transparent von `brig` von anderen verfügbaren Knoten geholt werden.

Plattform: Da `brig` auch komplett automatisiert und ohne Interaktion nutzbar ist, kann es auch als Plattform für andere Anwendungen genutzt werden, die Dateien sicher austauschen und synchronisieren müssen. Eine Anwendung in der Industrie 4.0 wäre beispielsweise die Synchronisierung von Konfigurationsdateien im gesamten Netzwerk.

Einsatz im öffentlichen Bereich: Aufgrund der Ende-zu-Ende Verschlüsselung und einfachen Usability ist eine Nutzung an Schulen, Universitäten sowie auch in Behörden zum Dokumentenaustausch denkbar. Vorteilhaft wäre für die jeweiligen Institutionen hierbei vor allem, dass man sich aufgrund des Open–Source–Modells an keinen Hersteller bindet (Stichwort: *Vendor Lock-In*) und keine behördlichen Daten in der Cloud landen. Eine praktische Anwendung im universitärem Bereich wäre die Verteilung von Studienunterlagen an die Studenten. Mangels einer Standardlösung ist es heutzutage schwierig Dokumente sicher mit Behörden auszutauschen. `brig` könnte hier einen Standard etablieren und in Zukunft als eine Plattform dienen, um beispielsweise medizinische Unterlagen mit einem

[27] Siehe auch https://owncloud.org, bzw. dessen Fork *Nextcloud* https://nextcloud.com

Krankenhaus auszutauschen.

2.6. Einsatzszenarien

Basierend auf den vorgestellten Nutzergruppen lassen sich einige konkrete Einsatzszenarien ableiten:

Synchronisationslösung: Spiegelung von zwei oder mehr Ordnern und das Teilen derselben zwischen ein oder mehreren Nutzern. Ein häufiger Anwendungsfall ist dabei die Synchronisation zwischen mehreren Geräten eines einzigen Nutzers. Eine selektive Synchronisation bestimmter Ordner ist vorerst nicht vorgesehen.

Transferlösung: Veröffentlichen von Dateien nach Außen mittels eines *Gateway* über den Browser. Eine beliebige Anzahl an anonymen Teilnehmern können die Datei herunterladen.

Versionsverwaltung: Alle Modifikationen an den bekannten Dateien werden aufgezeichnet. Bis zu einer bestimmten Tiefe können Dateien wiederhergestellt werden.

Backup- und Archivierungslösung: Es ist möglich Knoten so zu konfigurieren, dass alle Dateien gepinned werden. Ein solcher Knoten kann dann anderen Teilnehmern automatisch als Archiv für alte Dateien dienen.

Verschlüsselter Safe: Da alle Dateien verschlüsselt sind, müssen sie beim Zugriff der Software erst entschlüsselt werden. Da die entschlüsselten Daten nur im Hauptspeicher vorgehalten werden, ist nach Beenden der Software kein Zugriff mehr möglich.

Es gibt natürlich auch einige Einsatzzwecke, für die `brig` weniger geeignet ist. Diese werden in Kapitel 8 beleuchtet, da die dortige Argumentation teilweise ein Verständnis von der internen Architektur benötigt.

2.7. Annahmen während der Konzeption

Das Design von `brig` basiert auf einigen Annahmen, die im Voraus getroffen werden mussten:

Durchschnittliche Netzwerkkonfiguration: Für den Prototypen wird ein normales Heimnetzwerk mit mehren Computern angenommen, welche typischerweise hinter einem NAT liegen. Diese sollen sich mit anderen Computern in anderen Heimnetzwerken über das Internet austauschen können.

Durchschnittlicher Arbeitsrechner: Das Design wurde nicht auf leistungsschwache Hardware ausgerichtet. Ausgegangen wird von einem »normalen« Arbeitsrechner. Normal wird hier definiert durch Vorhandensein eines typischen Mehrkernprozessors aus dem Jahr 2008 oder später und mindestens 2 Gigabyte Arbeitsspeicher. Der Internetanschluss sollte ein Download von mindestens 4 Mbit/s[28] besitzen und ein Upload von 1 Mbit/s.

Stabilität von `ipfs`: Es wird angenommen, dass `ipfs` stetig weiterentwickelt wird und im momentanen Zustand keine gravierenden Sicherheitsmängel enthält. Zudem wird angenommen, dass es für die Zwecke von `brig` ausreichend hohe Performanz bietet.

[28] Der Durchschnitt im Jahr 2016 beträgt bereits etwa 14 Mbit/s. Quelle: Statista, [6].

Keine Kollision der Prüfsummen: `brig` kann (genau wie `ipfs`) Dateien nicht auseinanderhalten, die einen unterschiedlichen Inhalt besitzen, aber die selbe Prüfsumme erzeugen. Auch wenn dieser Fall in der Theorie eintreten kann, so ist dieser extrem schwer zu erreichen. Der von `ipfs` standardmäßig verwendete Algorithmus ist *sha256*[29], welcher eine Prüfsumme von 256 Bit Länge liefert. Wie in Gleichung (2.1) gezeigt, müssten trotz des Geburtstagsparadoxons[7] unpraktikabel viele Prüfsummen erzeugt werden, um eine Kollisionswahrscheinlichkeit von 0.1% zu erreichen, selbst wenn man sehr optimistisch annimmt, dass die Berechnung einer einzigen Prüfsumme nur eine Pikosekunde dauert.

$$(\frac{1}{1000} \times 2^{\frac{256}{2}}) \times 10^{-12}s \simeq 10^{35.5} \times 10^{-12}s \simeq 10^{15}\text{Jahre} \tag{2.1}$$

[29] Ein Prüfsummenalgorithmus der SHA-2 Familie. Siehe auch: https://de.wikipedia.org/wiki/SHA-2

3 Anforderungen

Im Folgenden wird auf die Anforderungen eingegangen, welche `brig` in Zukunft erfüllen soll. Diese sind weitreichender als der Umfang der aktuellen Implementierung. Die Anforderungen lassen sich in drei Kategorien unterteilen:

- **Anforderungen an die Integrität:** `brig` muss die Daten, die es speichert, versionieren, auf Integrität prüfen können und korrekt wiedergeben.
- **Anforderungen an die Sicherheit:** Alle Daten, die `brig` anvertraut werden, sollten sowohl bei der Speicherung auf der Festplatte als auch bei der Übertragung zwischen Partnern verschlüsselt werden. Die Implementierung der Sicherheitstechniken sollte transparent von Nutzern und Experten nachvollzogen werden können.
- **Anforderungen an die Usability:** Die Software soll möglichst einfach zu nutzen und zu installieren sein. Der Nutzer soll `brig` auf den populärsten Betriebssystemen nutzen können und auch Daten mit Nutzern anderer Betriebssysteme austauschen können.

Die Kategorien beinhalten einzelne, konkretere Anforderungen, die im Folgenden aufgelistet und erklärt werden. Dabei wird jeweils im ersten Paragraphen die eigentliche Anforderung formuliert und danach kurz beispielhaft erklärt. Ob und wie die Anforderung letztlich erfüllt wurde, wird in Kapitel 8 betrachtet.

Nicht jede Anforderung kann dabei voll umgesetzt werden. Teils überschneiden oder widersprechen sich Anforderungen an die Sicherheit und an die Effizienz, da beispielsweise verschlüsselte Speicherung mehr Prozessor–Ressourcen in Anspruch nimmt. Auch ist hohe Usability bei gleichzeitig hohen Sicherheitsanforderungen schwierig umzusetzen. Die Neueingabe eines Passworts bei jedem Zugriff mag sicherer sein, aber eben kaum benutzerfreundlich. Daher muss bei der Erfüllung der Anforderungen eine Priorisierung erfolgen. Im Zweifel wurde sich beim Entwurf von `brig` primär für die Usability entschieden. Zwar kann ein sehr sicheres System den Nutzer beschützen, doch wird der Nutzer es ungern einsetzen wollen, wenn es aufwendig zu bedienen ist. Das heißt allerdings keineswegs, dass `brig` »per Entwurf« unsicher ist. Es wurde darauf geachtet, dass Sicherheitstechniken den Benutzer möglichst wenig im Weg stehen und eher in den Hintergrund treten. Rob Pike hat diesen Punkt überspitzt, aber prägnant dargestellt:

> *Weak security that's easy to use will help more people than strong security that's hard to use. Door locks are a good example.*
>
> — Rob Pike ([8] S.24)

Die untenstehenden Anforderungen sind teilweise an die Eigenschaften des verteilten Dateisystems *Infinit* (beschrieben in [3], siehe S.39) angelehnt und an die Ausrichtung von `brig` angepasst worden.

3.1. Anforderungen an die Integrität

Entkopplung von Metadaten und tatsächlichen Daten: Statt einem zentralen Dienst, soll `brig` die Basis eines dezentralen Netzwerkes bilden. Dabei stellt jeder Teilnehmer einen Knoten in diesem Netzwerk dar. Nutzer des Netzwerkes können Dateien untereinander synchronisieren. Dabei muss nicht zwangsweise die gesamte Datei übertragen werden, jeder Nutzer verwaltet lediglich eine Liste der Metadaten der Dateien, die jeder Teilnehmer besitzt. Durch diese Entkopplung ist es möglich, bestimmte Dateien »on–demand« und für den Nutzer transparent zu übertragen.

Der Hauptvorteil einer dezentralen Architektur ist die erhöhte Ausfallsicherheit (kein *Single–Point–of–Failure*) und der Fakt, dass das Netzwerk durch seine Nutzer entsteht und keine eigene Infrastruktur benötigt. `brig` funktioniert daher als *Overlay–Netzwerk* (Siehe [2], S.8) über das Internet.

Pinning: Der Nutzer soll Kontrolle darüber haben, welche Dateien er lokal auf seinem Rechner speichert und welche er von anderen Teilnehmern dynamisch empfangen will. Dazu wird das Konzept des »Pinnings« und der »Quota« eingeführt. Ein Nutzer kann eine Datei manuell *pinnen*, um sie auf seinem lokalen Rechner zu behalten oder um `brig` anzuweisen, sie aus dem Netzwerk zu holen und lokal zwischenzulagern. Dateien, die `brig` explizit hinzugefügt wurden, werden implizit mit einem *Pin* versehen. Die *Quota* hingegen beschreibt ein Limit an Bytes, die lokal zwischengespeichert werden dürfen. Wird dies überschritten, so werden Daten gelöscht, die keinen Pin haben.

Das manuelle Pinnen von Dateien ist insbesondere nützlich, wenn eine bestimmte Datei zu einem Zeitpunkt ohne Internetzugang benötigt wird. Ein Beispiel wäre ein Zugpendler, der ein Dokument auf dem Weg zur Arbeit editieren möchte. Er kann dieses vorher *pinnen*, um es lokal auf seinem Laptop zu lagern.

Langlebigkeit: Daten, die `brig` anvertraut werden, müssen solange ohne Veränderung und Datenverlust gespeichert werden bis kein Nutzer mehr diese Datei benötigt.

Dabei ist zu beachten, dass diese Anforderung nur mit einer gewissen Wahrscheinlichkeit erfüllt werden kann, da heutige Hardware nicht die Integrität der Daten gewährleisten kann. So können beispielsweise Bitfehler[1] bei der Verarbeitung im Hauptspeicher oder konventionelle Festplatten mit beschädigten Platten die geschriebenen Daten verändern. Ist die Datei nur einmal gespeichert worden, kann sie von Softwareseite aus nicht mehr fehlerfrei hergestellt werden. Um diese Fehlerquelle zu verkleinern sollte eine Möglichkeit zur redundanten Speicherung geschaffen werden, bei der eine minimale Anzahl von Kopien einer Datei konfiguriert werden kann.

Verfügbarkeit: Alle Daten die `brig` verwaltet sollen stets erreichbar sein und bleiben. In der Praxis ist dies natürlich nur möglich, wenn alle Netzwerkteilnehmer ohne Unterbrechung zur Verfügung stehen oder wenn alle Dateien lokal zwischengelagert worden sind.

Oft sind viele Nutzer zu unterschiedlichen Zeiten online oder leben in komplett verschiedenen Zeitzonen. Aufgrund der Zeitverschiebung wäre eine Zusammenarbeit zwischen einem chinesischen und einem deutschen Nutzer schwierig. Eine mögliche Lösung wäre die Einrichtung eines automatisierten Knoten der ständig verfügbar ist. Statt Dateien direkt miteinander zu teilen, könnten Nutzer diesen

[1] Auch als *Bitrot* bekannt, siehe https://en.wikipedia.org/wiki/Data_degradation

Knoten als Zwischenlager benutzen. Falls nötig, soll es also auch möglich sein den Vorteil eines zentralen Ansatzes (also seine permanente Erreichbarkeit) mit `brig` zu kombinieren.

Integrität: Es muss sichergestellt werden, dass absichtliche oder unabsichtliche Veränderungen an den Daten festgestellt werden können.

Unabsichtliche Änderungen können wie oben beschrieben beispielsweise durch fehlerhafte Hardware geschehen. Absichtliche Änderungen können durch Angriffe von außen passieren, bei denen gezielt Dateien von einem Angreifer manipuliert werden. Als Beispiel könnte man an einen Schüler denken, welcher unbemerkt seine Noten in der Datenbank seiner Schule manipulieren will. Aus diesem Grund sollte das Dateiformat von `brig` mittels *Message Authentication Codes* (MACs) sicherstellen können, dass die gespeicherten Daten den ursprünglichen Daten entsprechen.

3.2. Anforderungen an die Sicherheit

Verschlüsselte Speicherung: Die Daten sollten verschlüsselt auf der Festplatte abgelegt werden und nur bei Bedarf wieder entschlüsselt werden. Kryptografische Schlüssel sollten aus denselben Gründen nicht unverschlüsselt auf der Platte, sondern nur im Hauptspeicher abgelegt werden.

Wie in Kapitel 2 beleuchtet wurde, speichern die meisten Dienste und Anwendungen zum Dateiaustausch ihre Dateien in keiner verschlüsselten Form. Es gibt allerdings eine Reihe von Angriffsszenarien (siehe auch [1]), die durch eine Vollverschlüsselung der Daten verhindert werden können.

Verschlüsselte Übertragung: Bei der Synchronisation zwischen Teilnehmern sollte der gesamte Verkehr ebenfalls verschlüsselt erfolgen. Nicht nur die Dateien selbst, sondern auch die dazugehörigen Metadaten sollen Ende–zu–Ende verschlüsselt werden.

Die Verschlüsselung der Metadaten erscheint vor allem im Lichte der Enthüllungen zur NSA–Affäre geboten[2]. Eine Ende–zu–Ende Verschlüsselung ist in diesem Fall vor allem deswegen wichtig, weil der Datenverkehr auch über andere, ansonsten unbeteiligte, Knoten im Netzwerk gehen kann.

Authentifizierung: `brig` sollte die Möglichkeit bieten zu überprüfen, ob Synchronisationspartner wirklich diejenigen sind, die sie vorgeben zu sein. Dabei muss zwischen der initialen Authentifizierung und der fortlaufenden Authentifizierung unterschieden werden. Bei der initialen Authentifizierung wird neben einigen Sicherheitsfragen ein Fingerprint des Kommunikationspartners übertragen, welcher bei der fortlaufenden Authentifizierung auf Änderung überprüft wird.

Mit welchen Partnern synchronisiert werden soll und wie vertrauenswürdig diese sind kann `brig` nicht selbstständig ermessen. Man kann allerdings dem Nutzer Hilfsmittel geben, um die Identität des Gegenüber zu überprüfen. So könnten Werkzeuge angeboten werden, mithilfe deren der Nutzer dem potenziellen Partner eine Frage (mit vordefinierter Antwort) schicken kann, die dieser dann beantworten muss. Alternativ können sich beide Partner vorher auf einem separaten Kanal auf ein gemeinsames Geheimnis einigen, welches dann über `brig` ausgetauscht und überprüft werden kann. Diese beiden Möglichkeiten sind ähneln der OTR–Implementierung des Instant-Messanger Pidgin[3].

[2] Siehe auch: https://de.wikipedia.org/wiki/Globale_%C3%9Cberwachungs-_und_Spionageaff%C3%A4re
[3] Webseite: https://www.pidgin.im

Identität: Jeder Benutzer des Netzwerks muss eine öffentliche Identität besitzen, welche ihn eindeutig identifiziert. Gekoppelt mit der öffentlichen Identität soll jeder Nutzer ein überprüfbares Geheimnis kennen, mithilfe dessen er sich gegenüber anderen authentifizieren kann. Zusätzlich dazu sollte es einen menschenlesbaren Nutzernamen für jeden Teilnehmer geben. Dieser sollte zur öffentlichen Identität des jeweiligen Nutzers auflösbar sein. Eine Registrierung bei einer zentralen Stelle soll nicht benötigt werden.

Transparenz: Die Implementierung aller oben genannten Sicherheitsfeatures muss für Anwender und Entwickler nachvollziehbar und verständlich sein. Durch die Öffnung des gesamten Quelltextes können Entwickler den Code auf Fehler überprüfen. Normale Anwender können die Arbeit von Herrn Piechula (siehe [1]) lesen, um für die Themantik der Sicherheit sensibilisiert zu werden und ein Überblick über die Sicherheit von `brig` zu bekommen. Dort wird auch das Entwicklungsmodell besprochen, welches helfen soll, sichere Software zu entwickeln.

3.3. Anforderungen an die Usability

Anmerkung: In Kapitel 7 werden weitere Anforderungen zur Usability in Bezug auf eine grafische Oberfläche definiert. Da diese nicht für die Gesamtheit der Software relevant sind, werden sie hier ausgelassen.

Automatische Versionierung: Die Dateien die `brig` verwaltet, sollen automatisch versioniert werden. Die Versionierung soll dabei in Form von *Checkpoints* bei jeder Dateiänderung erfolgen. Mehrere Checkpoints können manuell oder per *Timer* in einem zusammenhängenden *Commit* zusammengefasst werden. Die Menge an Dateien, die in alter Version vorhanden sind, sollen durch eine Speicher-Quota geregelt werden, die nicht überschritten werden darf. Wird dieses Limit überschritten, so werden die ältesten Dateien von der lokalen Maschine gelöscht. Die jeweiligen Checkpoints sind aber noch vorhanden und der darin referenzierte Stand kann von anderen Teilnehmern aus dem Netzwerk geholt werden, falls verfügbar.

Nutzer tendieren oft dazu mehrere Kopien einer Datei unter verschiedenen Orten als Backup anzulegen. Leider artet dies erfahrungsgemäß in der Praxis oft dazu aus, dass Dateinamen wie `FINAL-rev2.pdf` oder `FINAL-rev7.comments.pdf` entstehen. Daher wäre für viele Nutzer eine automatisierte und robuste Versionierung wünschenswert.

Portabilität: `brig` soll in möglichst portabler Weise implementiert werden, um die zunehmende Fragmentierung des Betriebssystemmarkts[9] zu berücksichtigen. Es sollen neben den populärsten Systemen wie Windows, macOS und GNU/Linux auf lange Sicht auch mobile Plattformen wie Android unterstützt werden.

Einfache Installation: `brig` sollte möglichst einfach und ohne Vorkenntnisse installierbar sein. Zur Installation gehört dabei nicht nur die Beschaffung der Software und deren eigentliche Installation, sondern auch die initiale Konfiguration. Die Erfahrungen des Autors haben gezeigt, dass Nutzer oft eine einfach zu installierende Software bevorzugen, obwohl eine schwerer zu installierende Software, ihr Problem möglicherweise besser löst.

Keine künstlichen Limitierungen: Mit `brig` sollten die gleichen für den Nutzer gewohnten Ope-

rationen und Limitierungen gelten, wie bei einem normalen Dateisystem. Als Datei wird in diesem Kontext ein Datenstrom verstanden, der unter einem bestimmten Pfad im Dateisystem ausgelesen oder beschrieben werden kann. Ihm zugeordnet sind Metadaten, wie Größe, Änderungsdatum und Zugriffsdatum. Dateien sollen kopiert, verschoben und gelöscht werden können. Zudem sollten keine Limitierungen der Pfadlänge oder der Dateigröße durch `brig` erfolgen. Auch soll keine bestimmte Enkodierung des Pfadnamens forciert werden.

Generalität: Die Nutzung von Techniken, die den Nutzerstamm auf bestimmte Plattformen einschränkt oder den Kauf zusätzlicher, spezieller Hardware benötigt, ist nicht erlaubt. Beispielsweise der Einsatz von plattformspezifischen Dateisystemen wie btrfs[4] oder ZFS[5] zur Speicherung entfällt daher. Auch darf nicht vorausgesetzt werden, dass alle Nutzer `brig` verwenden, da dies ein Lock–in wie bei anderen Produkten bedeuten würde.

Ein häufiger Anwendungsfall ist ein Nutzer, der ein bestimmtes Dokument anderen Nutzern zu Verfügung stellen möchte. Optimalerweise müssen dabei die Empfänger des Dokuments keine weitere Software installiert haben, sondern können die Datei einfach mittels eines Hyperlinks in ihrem Browser herunterladen. Zentrale Dienste können dies relativ einfach leisten, indem sie einen Webservice anbieten, welcher die Datei von einer zentralen Stelle herunterladbar macht. Ein dezentrales Netzwerk wie `brig` muss hingegen *Gateways* anbieten, also eine handvoll Dienste, welche zwischen den »normalen Internet« und dem `brig`–Netzwerk vermitteln (siehe Abb. 3.1). Nutzer, welche die Dateien verteilen wollen, können ein solches Gateway selbst betreiben oder können ein von Freiwilligen betriebenes Gateway benutzen.

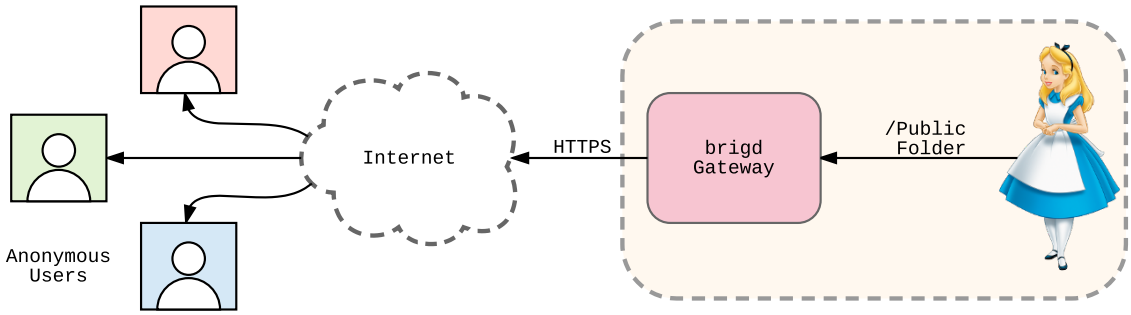


Abbildung 3.1.: Schematischer Aufbau eines HTTPS–Gateways.

Stabilität: Die Software muss bei normaler Benutzung ohne Abstürze und offensichtliche Fehler funktionieren. Eine umfangreiche Testsuite soll die Fehlerquote der Software minimieren, quantisierbar machen und die Weiterentwicklung erleichtern. Spätestens nach der Veröffentlichung der Software, sollten auch Regressionstests[6] das erneute Auftreten von bereits reparierten Fehlern vermeiden.

Effizienz: Die Geschwindigkeit der Software auf durchschnittlicher Hardware (siehe Abschnitt 2.7) soll schnell genug sein, um dem Anwender ein flüssiges Arbeiten ermöglichen zu können. Die Geschwindigkeit sollte durch eine Benchmarksuite messbar gemacht werden und bei jedem neuen Release mit dem Vorgänger verglichen werden.

[4] Siehe auch: https://de.wikipedia.org/wiki/Btrfs
[5] Siehe auch: https://de.wikipedia.org/wiki/ZFS_(Dateisystem)
[6] Siehe auch: https://de.wikipedia.org/wiki/Regressionstest

4 Grundlagen

Für das Verständnis der Architektur von `brig` ist die Erklärung einiger Internas von `ipfs` und dem freien Versionsverwaltungssystem `git` nötig. Diese werden im Folgenden gegeben.

4.1. `ipfs`: Das *Interplanetary Filesystem*

Anstatt das »Rad neu zu erfinden«, setzt `brig` auf das relativ junge *Interplanetary Filesystem* (kurz `ipfs`), welches von Juan Benet und seinen Mitentwicklern unter der MIT–Lizenz in der Programmiersprache Go entwickelt wird (siehe auch das Whitepaper[10]). Im Gegensatz zu den meisten anderen verfügbaren Peer–to–Peer Netzwerken kann `ipfs` als Software–Bibliothek genutzt werden. Dies ermöglicht es `brig` als, vergleichsweise dünne Schicht, zwischen Benutzer und `ipfs` zu fungieren (wie in Abb. 4.1 dargestellt).

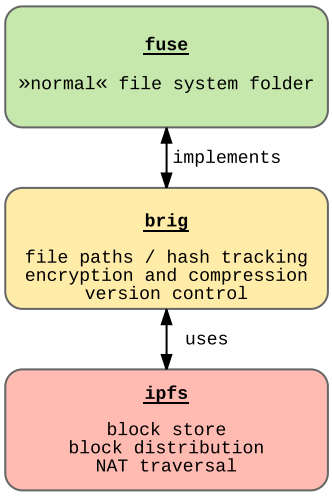


Abbildung 4.1.: Zusammenhang zwischen `ipfs`, `brig` und FUSE.

`Ipfs` stellt dabei ein *Content Addressed Network* (kurz *CAN*[1]) dar. Dabei wird eine Datei, die in das Netzwerk gelegt wird nicht mittels eines Dateinamen angesprochen, sondern mittels einer Prüfsumme, die durch eine vorher festgelegte Hashfunktion berechnet wird. Andere Teilnehmer im Netzwerk können mittels dieser Prüfsumme die Datei lokalisieren und empfangen. Anders als bei einer HTTP–URL (*Unified Resource Locator*) steckt in der Prüfsumme einer Datei also nicht nur die Lokation der Datei, sondern sie dient auch als eindeutiges Identifikationsmerkmal (ähnlich eines Pfads) und gleicht daher eher einem Magnet Link[2] als einer URL. Vereinfacht gesagt ist es nun die Hauptaufgabe von `brig` dem Nutzer die gewohnte Abstraktionsschicht eines Dateisystems zu geben, während im Hintergrund jede Datei zu einer Prüfsumme aufgelöst wird.

Im Vergleich zu zentralen Ansätzen können Dateien intelligent geroutet werden und müssen nicht

[1] Siehe auch: https://en.wikipedia.org/wiki/Content_addressable_network
[2] Mehr Informationen unter https://de.wikipedia.org/wiki/Magnet-Link

physikalisch auf allen Geräten verfügbar sein. Eine Datei kann »im Netzwerk liegen«. Greift der Nutzer über ihre Prüfsumme darauf zu, wird sie vom *CAN* intelligent aus dem Netzwerk geholt, sofern sie lokal nicht vorhanden ist. Dabei wird die Datei typischerweise in kleine Blöcke unterteilt, welche einzeln verteilt und geholt werden können. Daher müssen beispielsweise bei einem Netzwerkfehler nur alle Blöcke heruntergeladen werden, die noch fehlen.

Technisch basiert `ipfs` auf der Distributed–Hashtable *Kademlia* (vgl. [11] und [2], S. 247), welches mit den Erkenntnissen aus den Arbeiten *CoralDHST*[12] (Ansatz um das Routing zu optimieren) und *S/Kademlia*[13] (Ansatz um das Netzwerk gegen Angriffe zu schützen) erweitert und abgesichert wurde. *S/Kademlia* verlangt dabei, dass jeder Knoten im Netzwerk über ein Schlüsselpaar verfügt, bestehend aus einem öffentlichen und privaten Schlüssel. Die Prüfsumme des öffentlichen Schlüssels dient dabei als einzigartige Identifikation des Knotens und der private Schlüssel dient als Geheimnis mit dem ein Knoten seine Identität nachweisen kann. Diese Kernfunktionalitäten sind bei `ipfs` in einer separaten Bibliothek namens `libp2p`[3] untergebracht, welche auch von anderen Programmen genutzt werden können.

4.1.1. Eigenschaften des *Interplanetary Filesystems*

Im Folgenden werden die Eigenschaften von `ipfs` kurz vorgestellt, welche von `brig` genutzt werden. Einige interessante Features wie beispielsweise das *Interplanetary Naming System* (IPNS) werden dabei ausgelassen, da sie für `brig` aktuell keine praktische Bedeutung haben.

Weltweites Netzwerk: Standardmäßig bilden alle `ipfs`–Knoten ein zusammenhängendes, weltweites Netzwerk. `ipfs` verbindet sich beim Start mit einigen, wohlbekannten *Bootstrap–Nodes*, deren Adressen mit der Software mitgeliefert werden. Diese können dann wiederum den neuen Knoten an ihnen bekannte, passendere Knoten vermitteln. Die Menge der so entstandenen verbundenen Knoten nennt `ipfs` den *Swarm* (dt. Schwarm). Ein Nachbarknoten wird auch *Peer* genannt.

Falls gewünscht, kann allerdings auch ein abgeschottetes Subnetz erstellt werden. Dazu ist es lediglich nötig, die *Bootstrap*–Nodes durch Knoten auszutauschen, die man selbst kontrolliert. Unternehmen könnten diesen Ansatz wählen, falls ihr Netzwerk komplett von der Außenwelt abgeschottet sein soll. Wie in Kapitel 5 beleuchtet wird, ist eine Abschottung des Netzwerks rein aus Sicherheitsgründen nicht zwingend nötig.

Operation mit Prüfsummen: `ipfs` arbeitet nicht mit herkömmlichen Dateipfaden, sondern nur mit der Prüfsumme einer Datei. Im folgenden Beispiel wird eine Fotodatei mittels der `ipfs`–Kommandozeile in das Netzwerk gelegt[4]:

```
$ ipfs add my-photo.png
QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG
```

Wird eine Datei modifiziert, so muss sie neu mittels `ipfs add` hinzugefügt werden und wird in dieser Version unter einer anderen Prüfsumme erreichbar sein. Im Gegensatz zu normalen Dateisystemen kann es keinen allgemeinen Einstiegspunkt (wie das Wurzelverzeichnis »/«) geben. Die

[3] Mehr Informationen in der Dokumentation unter: https://github.com/ipfs/specs/tree/master/libp2p

[4] Voraussetzung hierfür ist allerdings, dass der `ipfs`–Daemon vorher gestartet wurde und ein Repository mittels `ipfs init` erzeugt wurde.

Prüfsumme eines Verzeichnisses definiert sich in ipfs durch die Prüfsummen seiner Inhalte. Das Wurzelverzeichnis hätte also nach jeder Modifikation eine andere Prüfsumme.

ipfs nutzt dabei ein spezielles Format um Prüfsummen zu repräsentieren[5]. Die ersten zwei Bytes einer Prüfsumme repräsentieren dabei den verwendeten Algorithmus und die Länge der darauf folgenden, eigentlichen Prüfsumme. Die entstandene Byte–Sequenz wird dann mittels base58[6] enkodiert, um sie menschenlesbar zu machen. Da der momentane Standardalgorithmus sha256 ist, beginnt eine von ipfs generierte Prüfsumme stets mit »Qm«. Abbildung Abb. 4.2 zeigt dafür ein Beispiel.

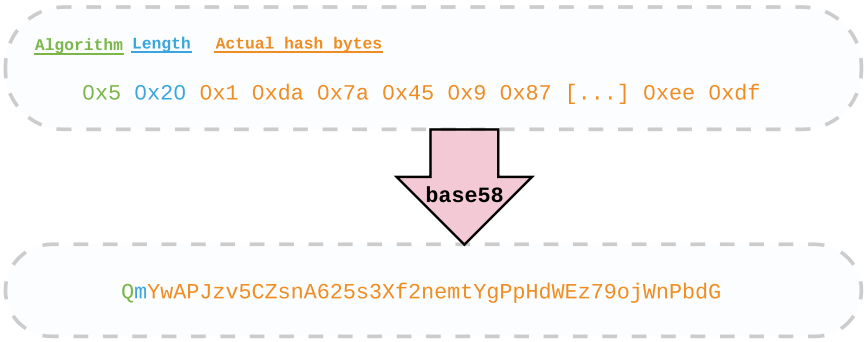


Abbildung 4.2.: Layout der ipfs Prüfsumme.

Auf einem anderen Computer, mit laufenden ipfs–Daemon, ist das Empfangen der Datei möglich, indem die Prüfsumme an das Kommando ipfs cat gegeben wird. Dabei wird für den Nutzer transparent über die DHT ein Peer ausfindig gemacht, der die Datei anbieten kann und der Inhalt von diesem bezogen:

```
$ ipfs cat QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG > my-photo.png
```

Public–Key Infrastructure: Jeder Knoten im ipfs–Netzwerk besitzt ein RSA–Schlüsselpaar, welches beim Anlegen des Repositories erzeugt wird. Mittels einer Prüfsumme wird aus dem öffentlichen Schlüssel eine Identität berechnet (Id $= H_{sha256}(K_{Public})$). Diese kann dann dazu genutzt werden, einen Knoten eindeutig zu identifizieren und andere Nutzer im Netzwerk nachzuschlagen und deren öffentlichen Schlüssel zu empfangen:

```
# Nachschlagen des öffentlichen Schlüssels eines zufälligen Bootstrap-Nodes:
$ ipfs id QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ
{
  "ID": "QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ",
  "PublicKey": "CAASpgIwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEK[...]",
  ...
}
```

Der öffentliche Schlüssel kann dazu genutzt werden, mit einem Peer mittels asymmetrischer Verschlüsselung eine verschlüsselte Verbindung aufzubauen (siehe [1]). Von brig wird dieses Konzept

[5] Mehr Informationen unter: https://github.com/multiformats/multihash
[6] https://de.wikipedia.org/wiki/Base58

weiterhin genutzt, um eine Liste vertrauenswürdiger Knoten zu verwalten. Jeder Peer muss bei Verbindungsaufbau nachweisen, dass er den zum öffentlichen Schlüssel passenden privaten Schlüssel besitzt (für Details siehe [1]).

Pinning und Caching: Das Konzept von `ipfs` basiert darauf, dass Knoten nur das speichern, woran sie auch interessiert sind. Daten, die von Außen zum eigenen Knoten übertragen worden sind werden nur kurzfristig zwischengelagert. Nach einiger Zeit bereinigt der eingebaute Garbage–Collector die Daten im *Cache* von `ipfs`.[7]

Werden Daten allerdings über den Knoten selbst hinzugefügt, so bekommen sie automatisch einen *Pin* (dt. Stecknadel). *Gepinnte* Daten werden automatisch vom *Garbage-Collector* ignoriert und beliebig lange vorgehalten, bis sie wieder *unpinned* werden. Möchte ein Nutzer sicher sein, dass die Datei im lokalen Speicher bleibt, so kann er sie manuell pinnen:

```
$ ipfs pin add QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG
```

Wenn die Dateien nicht mehr lokal benötigt werden, können sie *unpinned* werden:

```
$ ipfs pin rm QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG
```

Flexibler Netzwerkstack: Einer der größten Vorteile von `ipfs` ist, dass es auch über NAT–Grenzen hinweg funktioniert. Da aufgrund von *UDP–Hole–Punching* kein *TCP* genutzt werden kann, wird *UDP* genutzt. Um die Garantien zu erhalten, die *TCP* bezüglich der Paketzustellung gibt, nutzt `ipfs` das Anwendungs–Protokoll *UDT*. Insgesamt implementiert `ipfs` also einige Techniken, um, im Gegensatz zu den meisten theoretischen Ansätzen, eine leichte Usability zu gewährleisten. Speziell wäre hier zu vermeiden, dass ein Anwender die Einstellungen seines Routers ändern muss, um `brig` zu nutzen.

In Einzelfällen kann es trotzdem dazu kommen, dass die von `ipfs` verwendeten Ports durch eine besonders in Unternehmen übliche Firewall blockiert werden. Dies kann nötigenfalls aber vom zuständigen Administrator geändert werden.

Übermittlung zwischen Internet und `ipfs`: Ein Client/Server–Betrieb lässt sich mithilfe der `ipfs`–*Gateways* emulieren. Gateways sind zentrale, wohlbekannte Dienste, die zwischen dem »normalen Internet« und dem `ipfs` Netzwerk mittels HTTP vermitteln. Die Datei `my-photo.png` aus dem obigen Beispiel kann von anderen Nutzern bequem über den Browser heruntergeladen werden:

```
$ export PHOTO_HASH=QmPtoEEMMnbTSmzr28UEJFvmsD2dW88nbbCyyTrQgA9JR9
$ curl https://gateway.ipfs.io/ipfs/$PHOTO_HASH > my-photo.png
```

Auf dem Gateway läuft dabei ein Webserver, der die gleiche Aufgabe hat wie »`ipfs cat`«, aber statt auf der Kommandozeile die Daten auf eine HTTP–Verbindung ausgibt. Standardmäßig wird bei jedem Aufruf von `ipfs daemon` ein Gateway auf der Adresse http://localhost:8080 gestartet.

4.2. Datenmodell von `ipfs`

Wie bereits beschrieben ist `brig` ein »Frontend«, welches `ipfs` zum Speichern und Teilen von Dokumenten nutzt. Die Dokumente werden dabei einzig und allein über ihre Prüfsumme (`QmXYZ...`)

[7] Der Garbage–Collector kann auch manuell mittels `ipfs repo gc` von der Kommandozeile aufgerufen werden.

referenziert. Aus architektonischer Sicht kann man `ipfs` als eine verteilte Datenbank sehen, die vier simple Operationen beherrscht:

- $Put(\text{Stream}) \rightarrow$ Hash: Speichert einen endlichen Datenstrom in der Datenbank und liefert die Prüfsumme als Ergebnis zurück.
- $Get(\text{Hash}) \rightarrow$ Stream: Holt einen endlichen Datenstrom aus der Datenbank der durch seine Prüfsumme referenziert wurde und gibt ihn aus.
- $Pin(\text{Hash}, \text{Count})$: Pinnt einen Datenstrom wenn Count größer 0 ist oder unpinnt ihn wenn er negativ ist. Im Falle von 0 wird nichts getan. In jedem Fall wird der neue Status zurückgeliefert.
- $Cleanup$: Lässt einen »Garbage–Collector«[8] laufen, der Datenströme aus dem lokalen Speicher löscht, die nicht gepinned wurden.

Das Besondere ist, dass die $Get()$ Operation von jedem verbundenen Knoten ausgeführt werden kann, wodurch die Nutzung von `ipfs` als verteilte Datenbank möglich wird. Die oben geschilderte Sicht ist rein die Art und Weise in der `ipfs` von `brig` benutzt wird. Die Möglichkeiten, die `ipfs` bietet, sind tatsächlich sehr viel weitreichender als »nur« eine Datenbank bereitzustellen. Intern hat es ein mächtiges Datenmodell, das viele Relationen wie eine Verzeichnisstruktur, Versionsverwaltung, ein alternatives World–Wide–Web oder gar eine Blockchain[9] gut abbilden kann: Der *Merkle–DAG* (Direkter azyklischer Graph), im Folgenden kurz *MDAG* oder *Graph* genannt. Diese Struktur ist eine Erweiterung des Merkle–Trees[14], bei der ein Knoten mehr als einen Elternknoten haben kann.

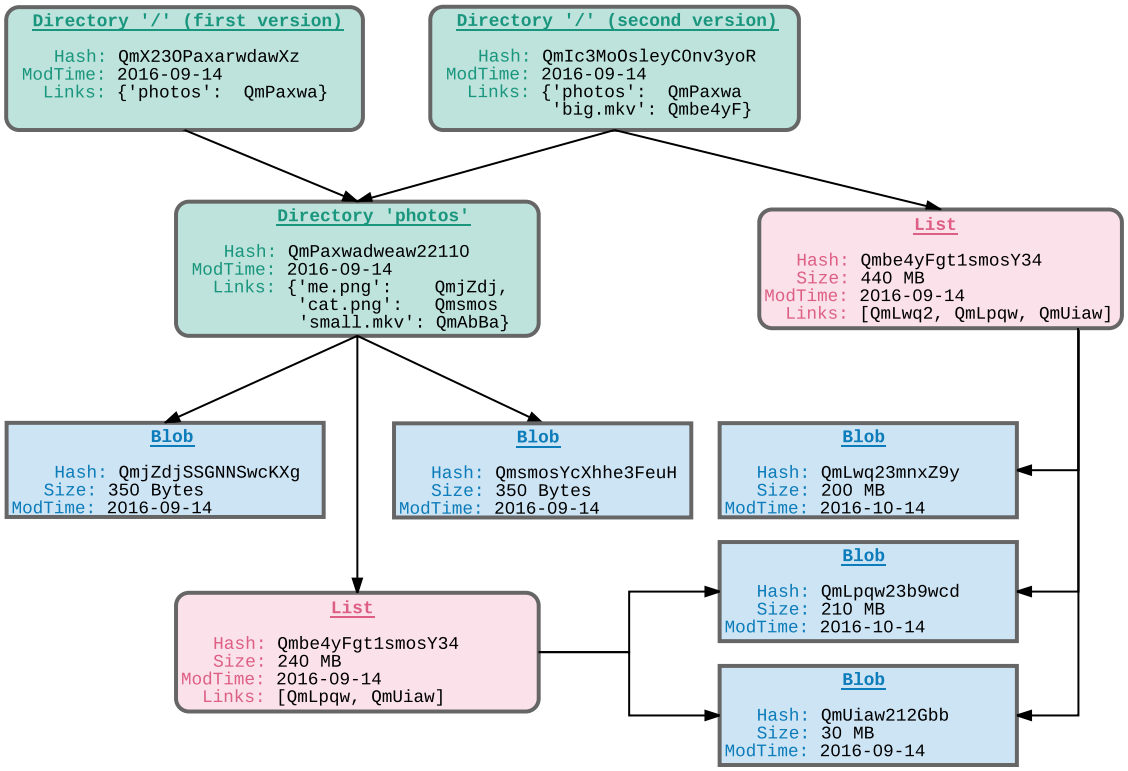


Abbildung 4.3.: Beispielhafter MDAG der eine Verzeichnisstruktur abbildet.

[8] Siehe auch https://de.wikipedia.org/wiki/Garbage_Collection
[9] Siehe auch die Erklärung hier: https://medium.com/@ConsenSys/an-introduction-to-ipfs-9bba4860abd0#.t6mcryb1r

In Abb. 4.3 ist ein beispielhafter Graph gezeigt, der eine Verzeichnishierarchie modelliert. Die gezeigten Attributnamen entsprechen den ipfs–Internas. Gerichtet ist der Graph deswegen, weil es keine Schleifen und keine Rückkanten zu den Elternknoten geben darf. Jeder Knoten wird durch eine Prüfsumme referenziert und kann wiederum mehrere andere Knoten über weitere Prüfsümmen referenzieren. Im Beispiel sieht man zwei Wurzelverzeichnisse, bei denen das erste ein Unterverzeichnis /photos enthält, welches wiederum drei einzelne Dateien (cat.png, me.png und small.mkv) enthält. Das zweite Wurzelverzeichnis beinhaltet ebenfalls dieses, referenziert als zusätzliche Datei aber noch eine größere Datei namens big.mkv. Die Besonderheit ist dabei, dass die Dateien jeweils in einzelne Blöcke (blobs) zerlegt werden, die automatisch dedupliziert abgespeichert werden. In der Grafik sieht man das dadurch, dass big.mkv bereits aus zwei Blöcken von small.mkv besteht und der zweite Wurzelknoten auf /photos referenziert, ohne dessen Inhalt zu kopieren.

Im Datenmodell von ipfs ([10]) gibt es vier unterschiedliche Strukturen:

- blob: Ein Datensatz mit definierter Größe und Prüfsumme. Wird teilweise auch *Chunk* genannt.
- list: Eine geordnete Liste von blobs oder weiteren lists. Wird benutzt um große Dateien in kleine, deduplizierbare Teile herunterzubrechen.
- tree: Eine Abbildung von Dateinamen zu Prüfsummen. Modelliert ein Verzeichnis, das blobs, lists oder andere trees beinhalten kann. Die Prüfsumme ergibt sich aus den Kindern.
- commit: Ein Snapshot eines der drei obigen Strukturen. In der Grafik nicht gezeigt, da diese Datenstrukutur noch nicht finalisiert ist[10].

Wenn ipfs bereits ein Datenmodell hat, welches Verzeichnisse abbilden kann, ist es eine berechtigte Frage, warum brig ein eigenes Datenmodell implementiert und nicht das vorhandene als Basis verwendet. Der Grund dafür liegt in der bereits erwähnten Entkopplung von Daten und Metadaten. Würden die Dateien und Verzeichnisse direkt in ipfs abgebildet, so wäre diese Teilung nicht mehr gegeben, da trotzdem alle Daten in einem gemeinsamen Speicher liegen. Dies hätte zur Folge, dass ein Angreifer zwar nicht die verschlüsselten Daten lesen könnte, aber problemlos die Verzeichnisstruktur betrachten könnte, sobald er die Prüfsumme des Wurzelknotens hat. Dies würde den Sicherheitsversprechen von brig widersprechen. Abgesehen davon wurde ein eigenes Datenmodell entwickelt, um mehr Freiheiten beim Design zu haben.

Zusammengefasst lässt sich also sagen, dass ipfs in dieser Arbeit als Content–Adressed–Storage–Datenbank verwendet wird, die sich im Hintergrund um die Speicherung von Datenströmen und deren Unterteilung in kleine Blöcke mittels *Chunking* kümmert. Die Aufteilung geschieht dabei entweder simpel, indem die Datei in gleichgröße Blöcke unterteilt wird, oder indem ein intelligenter Algorithmus wie Rabin–Karp–Chunking[15] angewandt wird.

4.3. Datenmodell von git

Der interne Aufbau von brig ist relativ stark von den Internas des freien Versionsverwaltungssystem git inspiriert. Deshalb werden im Folgenden immer wieder Parallelen zwischen den beiden Systemen gezogen, um die jeweiligen Unterschiede aufzuzeigen und zu erklären warum brig letztlich einige wichtige Differenzen aus architektonischer Sicht aufweist. Was die Usability angeht, soll allerdings

[10] Diskussion der Entwickler hier: https://github.com/ipfs/notes/issues/23

aufgrund der relativ unterschiedlichen Ziele kein Vergleich gezogen werden.

Im Folgenden ist ein gewisses Grundwissen über git nützlich. Es wird bei Unklarheiten das Buch »*Git — Verteilte Versionsverwaltung für Code und Dokumente*[5]« empfohlen. Alternativ bietet auch die offizielle Projektdokumentation[11] einen sehr guten Überblick. Aus Platzgründen wird an dieser Stelle über eine gesonderte Einführung verzichtet, da es diese in ausreichender Menge frei verfügbar gibt.

Kurz beschrieben sind beide Projekte »*stupid content tracker*«[12], die Änderungen an tatsächlichen Dateien auf Metadaten abbilden, welche in einer dafür geeigneten Datenbank abgelegt werden. Die eigentlichen Daten werden dabei nicht mittels eines Pfades abgespeichert, sondern werden durch eine Prüfsumme referenziert (im Falle von git mittels sha1). Im Kern lösen beide Programme also Pfade in Prüfsummen auf und umgekehrt. Um diese Auflösung so einfach und effizient wie möglich zu machen, nutzt git ein ausgeklügeltes Datenmodell, mit dem sich Änderungen abbilden lassen. Dabei werden, anders als bei anderen Versionsverwaltungssystemen (wie Subversion), Differenzen »on-the-fly« berechnet und nicht zusätzlich abgespeichert, daher die Bezeichnung »stupid«. Abgespeichert werden, wie in Abb. 4.4 gezeigt, nur vier verschiedene *Objekte*:

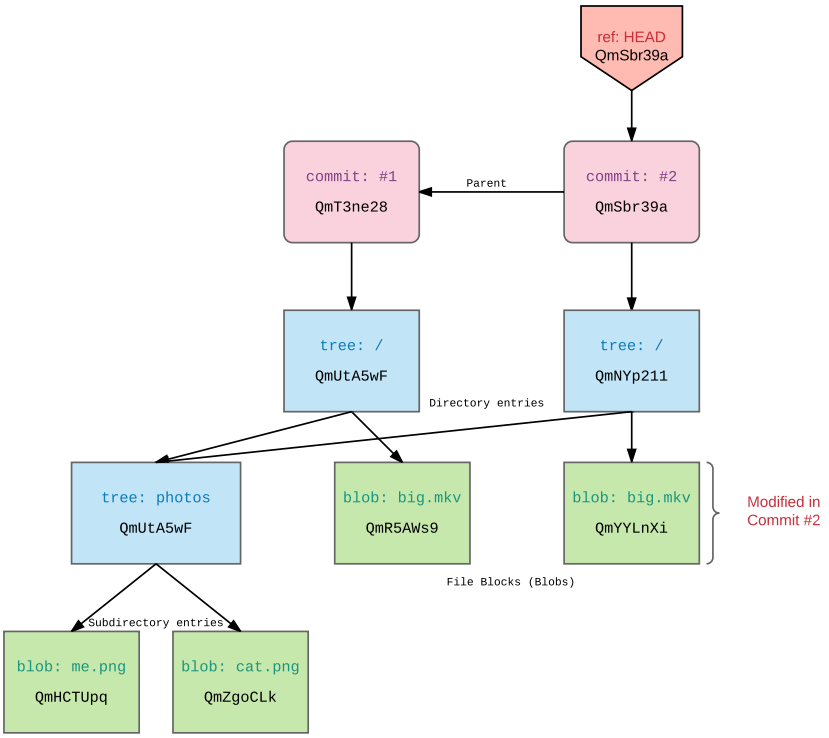


Abbildung 4.4.: Vereinfachte Darstellung des Datenmodells von git.

- **Blob:** Speichert Daten einer bestimmten Größe komprimiert ab und assoziiert diese mit einer sha1–Prüfsumme des unkomprimierten Dateiinhaltes.
- **Tree:** Speichert *Blobs* oder weitere *Trees*, modelliert also eine Art »Verzeichnis«. Seine Prüfsumme ergibt sich, indem eine Prüfsumme aus den Prüfsummen der Kinder gebildet wird. Zusammen mit *Blobs* lässt sich bereits ein »unixoides Dateisystem« modellieren, bei dem alle

[11] Offizielle Projektdokumentation von git: https://git-scm.com/doc
[12] Zitat von Linus Torvalds. Siehe auch: https://git-scm.com/docs/git.html

Dateien von einem Wurzelknoten (ein *Tree* ohne Vorgänger) aus mittels eines Pfades erreichbar sind.

- **Commit:** Ein *Commit* speichert den Zustand des »Dateisystems«, indem es seinen Wurzelknoten referenziert. Zudem hat ein *Commit* mindestens einen Vorgänger (meist *Parent* genannt, kann beim initialen *Commit* leer sein) und speichert eine vom Nutzer verfasste Änderungszusammenfassung ab, sowie den Namen des Nutzers. Seine Prüfsumme ergibt sich indem eine Konkatenation der Wurzelprüfsumme, der Vorgängerprüfsumme, des Nutzernamen und der Commit–Nachricht[13].
- **Ref:** Eine Referenz auf einen bestimmten *Commit*. Er speichert lediglich dessen Prüfsumme und wird von `git` separat zu den eigentlichen Objekten gespeichert. In Abb. 4.4 verweist beispielsweise die Referenz `HEAD` stets auf den aktuellsten *Commit*.

Die ersten drei Objekte werden in einem MDAG untereinander in Relation gesetzt. Diese Struktur ergibt sich dadurch, dass bei Änderung einer Datei in `git` sich sämtliche Prüfsummen der Verzeichnisse darüber ändern. In Abbildung Abb. 4.4 wurde im zweiten Commit die Datei `big.mkv` verändert (Prüfsumme ändert sich von *QmR5AWs9* zu *QmYYLnXi*). Als direkte Konsequenz ändert sich die Prüfsumme des darüber liegenden Verzeichnisses, in diesem Fall das Wurzelverzeichnis »/«. Bemerkenswert ist hier aber, dass das neue »/«–Verzeichnis trotzdem auf das `/photos`–Verzeichnis des vorherigen *Commits* verlinkt, da dieses sich in der Zwischenzeit nicht geändert hat.

Jede Änderung bedingt daher eine Veränderung der Prüfsumme des »/«–Verzeichnisses. Daher sichert dies die Integrität aller darin enthaltenen Dateien ab. Aufgrund dessen kann ein darüber liegender *Commit* einfach ein *Wurzelverzeichnis* referenzieren, um eine Momentaufnahme aller Dateien zu erzeugen. Jeder *Commit* lässt in seine eigene Prüfsumme zudem die Prüfsumme seines Vorgängers einfließen, weshalb jegliche (absichtliche oder versehentliche) Modifikation der von `git` gespeicherten Daten aufgedeckt werden kann.

Möchte `git` nun die Unterschiede zwischen zwei Dateiständen in zwei verschiedenen Commits anzeigen, so geht es folgendermaßen vor:

1) Löse die Prüfsummen der beiden zu untersuchenden *Commits* auf.
2) Löse die Prüfsummen der darin enthaltenen Wurzelverzeichnisse auf.
3) Traversiere in beiden Wurzelverzeichnisse zum gewünschten *Blob*.
4) Lade beide *Blobs* und wende einen Algorithmus an, der Differenzen findet (z. B. `diff` von Unix).
5) Gebe Differenzen aus.

Dies ist ein signifikanter Unterschied zu zentralen Versionsverwaltungssystemen wie `svn`, die jeweils die aktuellste Datei ganz und ein oder mehrere »Reverse-Diff« abspeichern. Mithilfe des *Reverse-Diff* ist es möglich, die alten Stände wiederherzustellen. Obwohl das auf den ersten Blick wie ein Vorteil von `svn` wirkt, so nutzt dieses in der Praxis deutlich mehr Speicherplatz für ein Repository[14] und ist signifikant langsamer als `git`, insbesondere da Netzwerkzugriffe nötig sind, während `git` lokal arbeitet. Insbesondere beim Erstellen von *Commits* und dem Wiederherstellen alter Stände ist `git` durch sein Datenmodell erstaunlich schnell. Tatsächlich speichert `git` auch nicht jeden *Blob* einzeln, sondern fasst diese gelegentlich zu sogenannten *Packfiles* zusammen, welche vergleichbar mit einem

[13] Mehr Details unter: https://gist.github.com/masak/2415865
[14] https://git.wiki.kernel.org/index.php/GitSvnComparison#Smaller%20Space%20Requirements

indizierten, komprimierten Archiv mehrerer Objekte sind[15].

Zusammengefasst hat `git` also aus architektonischer Sicht einige positive Eigenschaften:

- Objekte werden vollautomatisch und ohne weiteren Aufwand dedupliziert abgespeichert.
- Das Datenmodell ist minimalistisch gehalten und relativ leicht verständlich.
- Nicht alle Objekte müssen beim Start von `git` geladen werden. Lediglich die benötigten Objekte werden von `git` geladen, was den Startvorgang beschleunigt.
- Das Bilden einer dezentralen Architektur liegt nahe, da das Datenmodell immer alle Objekte beinhalten muss. Jeder Knoten hat also dieselben Informationen.
- Alle Dateien liegen in einem separaten `.git`–Verzeichnis und alle darin enthaltenen Internas sind durch die gute Dokumentation gut verständlich und nötigenfalls reparierbar. Zudem ist das Arbeitsverzeichnis ein ganz normales Verzeichnis, in dem der Benutzer arbeiten kann ohne von `git` gestört zu werden.
- Die gespeicherten Daten sind durch kryptografische Prüfsummen gegen Veränderungen geschützt. Ein potentieller Angreifer müsste ein *Blob* generieren, der die von ihm gewünschten Daten enthält *und* die gleiche Prüfsumme, wie der bereits vorhandene *Blob* erzeugt. Obwohl `sha1` nicht mehr empfohlen wird[16], wäre das ein sehr rechenintensiver Angriff.

Aus Sicht des Autors hat `git` aus architektonischer Sicht einige kleinere Schwächen:

1) **Prüfsummenalgorithmus nicht veränderbar:** Ein MDAG–basiertes Versionsverwaltungssystem muss eine Abwägung zwischen der Prüfsummenlänge (länger ist typischerweise rechenaufwendiger, braucht mehr Speicher und ist unhandlicher für den Benutzer) und der Kollisionsresistenz der kryptografischen Prüfsumme treffen. Tritt trotzdem eine Kollision auf, so können Daten überschrieben werden[17]. Unabsichtliche Kollisionen sind sehr unwahrscheinlich. Mit steigender Rechenleistungen wird die Berechnung einer absichtlichen Kollision aber denkbar. Leider kann `git` den genutzten Prüfsummenalgorithmus (`sha1`) nicht mehr ohne hohen Aufwand ändern[18]. Bei `brig` ist dies möglich, da das Prüfsummenformat von `ipfs` die Länge und Art des Algorithmus in der Prüfsumme selbst abspeichert.
2) **Keine nativen Renames:** `git` behandelt das Verschieben einer Datei als eine Sequenz aus dem Löschen und anschließendem Hinzufügen der Datei[19]. Der Nachteil dabei ist, dass `git` dem Nutzer die Umbenennung nicht mehr als solche präsentiert, was für diesen verwirrend sein kann wenn er nicht sieht, dass die Datei anderswo neu hinzugefügt wurde. Neuere `git` Versionen nutzen Heuristiken, um Umbenennungen zu finden (Beispiel: Pfad wurde gelöscht, Prüfsumme der Datei tauchte aber anderswo auf). Diese können zwar nicht alle Fälle abdecken (umbenannt, dann modifiziert) leisten aber in der Praxis gute Dienste.
3) **Probleme mit großen Dateien:** Da `git` für die Verwaltung von Quelltextdateien entwickelt wurde, ist es nicht auf die Verwaltung großer Dateien ausgelegt. Jede Datei muss einmal im `.git`–Verzeichnis und einmal im Arbeitsverzeichnis gespeichert werden, was den Speicherverbrauch mindestens verdoppelt. Da Differenzen zwischen Binärdateien nur wenig Aussagekraft

[15] Siehe auch: https://git-scm.com/book/be/v2/Git-Internals-Packfiles
[16] Siehe unter anderem: https://www.schneier.com/blog/archives/2005/02/sha1_broken.html
[17] Siehe auch: http://ericsink.com/vcbe/html/cryptographic_hashes.html
[18] Mehr zum Thema unter: https://lwn.net/Articles/370907
[19] https://git.wiki.kernel.org/index.php/GitFaq#Why_does_Git_not_.22track.22_renames.3F

haben (da Differenz–Algorithmen normalerweise zeilenbasiert arbeiten) wird bei jeder Modifikation jeweils noch eine Kopie angelegt. Nutzer, die ein solches Repository »*clonen*« (also sich eine eigene Arbeitskopie besorgen wollen), müssen diese Kopien lokal bei sich speichern. Werkzeuge wie `git-annex` versuchen das Problem zu lösen, indem sie statt den Dateien, nur symbolische Links versionieren, die zu den tatsächlichen Dateien zeigen[20]. Symbolische Links sind allerdings wenig portabel.

4) **Kein Tracking von leeren Verzeichnissen:** Es können keine leeren Verzeichnisse zu `git` hinzugefügt werden. Damit ein Verzeichnis von `git` verfolgt werden kann, muss sich mindestens eine Datei darin befinden. Das ist weniger eine Einschränkung des Datenmodells von `git`, als viel mehr ein kleiner Designfehler[21] in der Implementierung, der bisher als zu unwichtig galt, um korrigiert zu werden.
5) **Keine eigene Historie pro Datei:** Es gibt nur eine gesamte *Historie*, die durch die Verkettung von *Commits* erzeugt wird. Bei einem Befehl wie `git log <filename>` (Zeige alle Commits, in denen `<filename>` verändert wurde) müssen alle *Commits* betrachtet werden, auch wenn `<filename>` nur in wenigen davon tatsächlich etwas geändert wurde. Eine mögliche Lösung wäre das Anlegen einer Historie für einzelne Dateien.

Zusammengefasst lässt sich sagen, dass `git` ein extrem flexibles und schnelles Werkzeug für die Verwaltung von Quelltext und kleinen Dateien ist. Weniger geeignet ist es für eine allgemeine Dateisynchronisationssoftware, die auch große Dokumente effizient behandeln können muss.

[20] https://git-annex.branchable.com/direct_mode
[21] https://git.wiki.kernel.org/index.php/GitFaq#Can_I_add_empty_directories.3F

5 Architektur

In diesem Kapitel wird die grundlegende Architektur von `brig` erklärt. Dabei wird vor allem das »Kernstück« beleuchtet: Das zugrundeliegende Datenmodell in dem alle Metadaten abgespeichert und in Relation gesetzt werden. Dazu wird auf die zuvor erklärten Internas von `ipfs` und `git` eingegangen.

Basierend darauf werden die umgebenden Komponenten beschrieben, die um den Kern von `brig` gelagert sind. Am Ende des Kapitels werden zudem noch einmal alle Einzelkomponenten in einer Übersicht gezeigt. Mögliche Erweiterungen werden in Kapitel 8 (*Evaluation*) diskutiert. Die technische Umsetzung des Prototypen hingegen wird in Kapitel 6 (*Implementierung*) besprochen.

5.1. Datenmodell von `brig`

Die Einsatzziele von `brig` und `git` unterscheiden sich: `git` ist primär eine Versionsverwaltugssoftware, mit der man auch synchronisieren kann. `brig` kann man hingegen eher als eine Synchronisationssoftware sehen, die auch Versionierung beherrscht. Aus diesem Grund wurde das Datenmodell von `git` für den Einsatz in `brig` angepasst und teilweise vereinfacht. Die Hauptunterschiede sind dabei wie folgt:

- **Strikte Trennung zwischen Daten und Metadaten:** Metadaten werden von `brig`'s Datenmodell verwaltet, während die eigentlichen Daten lediglich per Prüfsumme referenziert und von `ipfs` gespeichert werden. So gesehen ist `brig` ein Versionierungsaufsatz für `ipfs`.
- **Lineare Versionshistorie:** Jeder *Commit* hat maximal einen Vorgänger und exakt einen Nachfolger. Dies macht die Benutzung von *Branches*[1] unmöglich, bei der ein *Commit* zwei Nachfolger haben kann, beziehungsweise sind auch keine Merge–Commits möglich, die zwei Vorgänger besitzen. Diese Vereinfachung ist nicht von der Architektur vorgegeben und könnte nachgerüstet werden. Allerdings hat die Benutzung dieses Features »Verwirrungspotenzial«[2] für gewöhnliche Nutzer, die gedanklich eher von einer linearen Historie ihrer Dokumente ausgehen.
- **Synchronisationspartner müssen keine gemeinsame Historie haben:**[3] Es wird bei `brig` davon ausgegangen, dass unterschiedliche Dokumentensammlungen miteinander synchronisiert werden sollen, während bei `git` davon ausgegangen wird, dass eine einzelne Dokumentensammlung immer wieder modifiziert und zusammengeführt wird. Haben die Partner keine gemeinsame Historie, wird einfach angenommen, dass alle Dokumente synchronisiert werden müssen. Aus diesen Grund kennt `brig` auch keine `clone` und `pull`–Operation. Diese werden durch »`brig sync <with>`« ersetzt.

Abb. 5.1 zeigt das oben verwendete Beispiel in `brig`'s Datenmodell. Es werden prinzipiell die gleichen Objekttypen verwendet, die auch `git` verwendet:

[1] *Branches* dienen bei `git` dazu, um einzelne Features oder Fixes separat entwickeln zu können.
[2] So ist es bei `git` relativ einfach möglich in den sogenannten *Detached HEAD* Modus zu kommen, in dem durchaus Daten verloren gehen können. Siehe auch: http://gitfaq.org/articles/what-is-a-detached-head.html
[3] Streng genommen ist dies bei `git` auch nicht nötig, allerdings sehr unüblich.

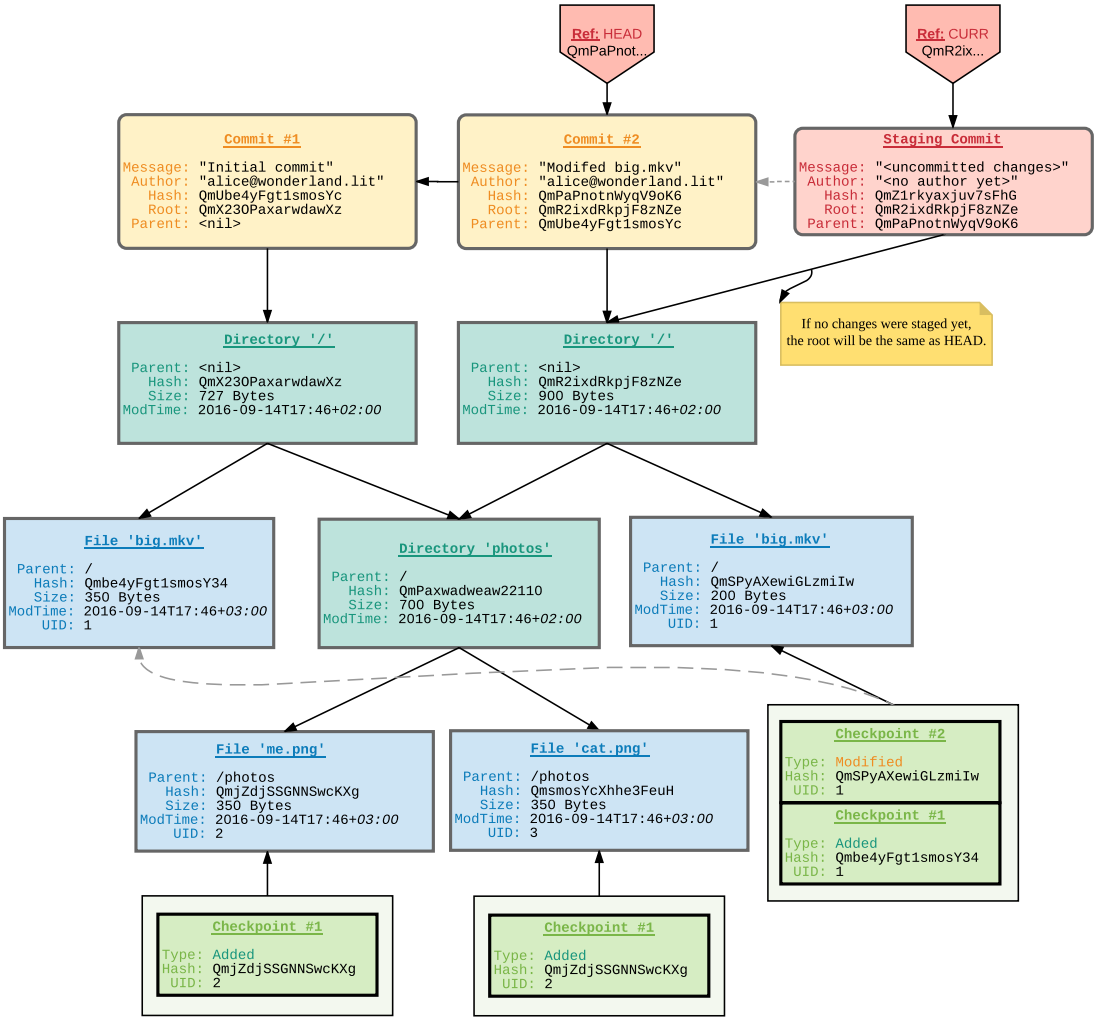


Abbildung 5.1.: Das Datenmodell von brig. Checkpoints von Verzeichnissen wurden ausgelassen.

- **File:** Speichert die Metadaten einer einzelnen, regulären Datei. Zu den Metadaten gehört die aktuelle Prüfsumme, die Dateigröße, der letzte Änderungszeitpunkt und der kryptografische Schlüssel mit dem die Datei verschlüsselt ist. Anders als ein *Blob* speichert ein *File* die Daten nicht selbst, sondern referenziert diese nur im `ipfs`–Backend.
- **Directory:** Speichert wie ein *Tree* einzelne *Files* und weitere *Directories*. Die Prüfsumme des Verzeichnisses $H_{directory}$ ergibt sich auch hier aus der XOR–Verknüpfung ($\oplus$) der Prüfsumme des Pfades H_{path} mit den Prüfsummen der direkten Nachfahren x:

$$H_{directory}(x) = \begin{cases} H_{path} & \text{für } x = () \\ x_1 \oplus f(x_{(x_2,\ldots,x_n)}) & \text{sonst} \end{cases}$$

  Die Verwendung der XOR–Verknüpfung hat dabei den Vorteil, dass sie selbstinvers und kommutativ ist. Wendet man sie also zweimal an, so erhält man das neutrale Element 0. Analog dazu führt die Anwendung auf ein vorheriges Ergebnis wieder zur ursprünglichen Eingabe:

$$x \oplus x = 0 \text{ (Auslöschung)}$$

$$y = y \oplus x \oplus x = x \oplus y \oplus x = x \oplus x \oplus y \text{ (Kommutativität)}$$

  Diese Eigenschaft kann man sich beim Löschen einer Datei zunutze machen, indem die Prüfsumme jedes darüberliegenden Verzeichnisses mit der Prüfsumme der zu löschenden Datei XOR–genommen wird. Der resultierende Graph hat die gleiche Prüfsumme wie vor dem Einfügen der Datei.
- **Commits:** Analog zu `git`; dienen aber bei `brig` nicht nur der logischen Kapselung von mehreren Änderungen, sondern werden auch automatisiert von der Software nach einem bestimmten Zeitintervall erstellt. Daher ist ihr Zweck eher mit den *Snapshots* vieler Backup–Programme vergleichbar, welche dem Nutzer einen Sicherungspunkt zu einem bestimmten Zeitpunkt in der Vergangenheit bieten. Als Metadaten speichert er als Referenz die Prüfsumme des Wurzelverzeichnisses, eine Commit–Nachricht sowie dessen Autor und eine Referenz auf den Vorgänger. Aus diesen Metadaten wird durch Konkatenation derselben eine weitere Prüfsumme errechnet, die den Commit selbst eindeutig referenziert. In diese Prüfsumme ist nicht nur die Integrität des aktuellen Standes gesichert, sondern auch aller Vorgänger.
- **Refs:** Analog zu `git` dienen sie dazu, bestimmten *Commits* einen Namen zu geben. Es gibt zwei vordefinierte Referenzen, welche von `brig` aktualisiert werden: `HEAD`, welche auf den letzten vollständigen *Commit* zeigt und `CURR`, welche auf den aktuellen Commit zeigt (meist dem *Staging Commit*, dazu später mehr). Da es keine Branches gibt, ist eine Unterscheidung zwischen *Refs* und *Tags* wie bei `git` nicht mehr nötig.

Directories und *Files* speichern zudem zwei weitere gemeinsame Attribute:

- **Ihren eigenen Namen und den vollen Pfad des darüber liegenden Knoten.** Zusammen ergibt dieser den vollen Pfad der Datei oder des Verzeichnisses. Dieser Pfad ist nötig, um den jeweiligen Elternknoten zu erreichen. In einem gerichteten, azyklischen Graphen darf es keine

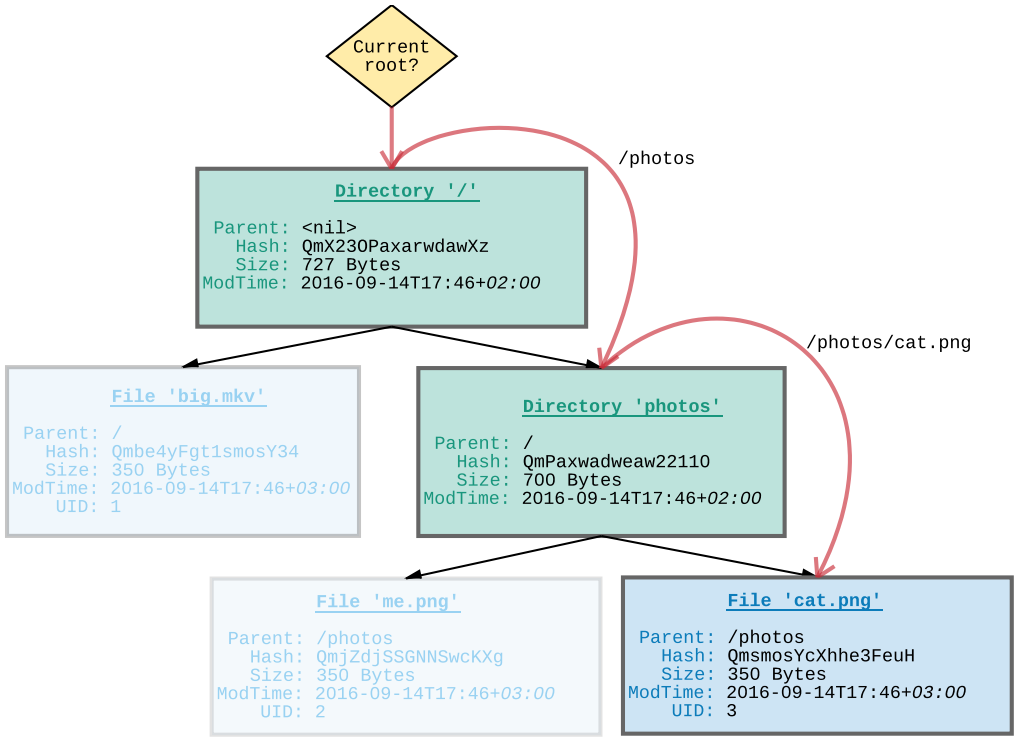


Abbildung 5.2.: Jeder Knoten muss von dem aktuellen Wurzelverzeichnis aus neu aufgelöst werden, selbst wenn nur der Elternknoten gesucht wird.

Rückkanten nach »oben« geben, deswegen scheidet die direkte Referenzierung des Elternknotens mittels seiner Prüfsumme aus. Wie in Abb. 5.2 gezeigt ist es daher nötig, beispielsweise den Elternknoten eines beliebigen Knotens vom aktuellen Wurzelknoten neu aufzulösen.

- **Eine eindeutige Nummer** (*Unique Identifier*, `UID`), welche die Datei oder das Verzeichnis eindeutig kennzeichnet. Diese Nummer bleibt auch bei Modifikation und Verschieben der Datei gleich. Neben der Prüfsumme (referenziert einen bestimmten Inhalt) und dem Pfad (referenziert eine bestimmte Lokation) ist die Nummer ein weiterer Weg eine Datei zu referenzieren (referenziert ein veränderliches »Dokument«) und ist grob mit dem Konzept einer *Inode-Nummer* bei Dateisystemen[4] vergleichbar.

Davon abgesehen fällt auf, dass zwei zusätzliche Strukturen eingeführt wurden:

- **Checkpoints:** Jeder Datei ist über ihre `UID` eine Historie von mehreren, sogenannten *Checkpoints* zugeordnet. Jeder Einzelne dieser Checkpoints beschreibt eine atomare Änderung an der Datei. Da keine partiellen Änderungen[5] möglich sind, müssen nur vier verschiedene Operation unterschieden werden: `ADD` (Datei wurde initial oder erneut hinzugefügt), `MODIFY` (Prüfsumme hat sich verändert), `MOVE` (Pfad hat sich verändert) und `REMOVE` (Datei wurde entfernt). Eine beispielhafte Historie findet sich in Abb. 5.3. Werden mehrere Checkpoints eingepflegt, die den gleichen Typen haben (beispielsweise mehrere `MODIFY`–Operationen), so wird nur die letzte `MODIFY`–Operation in der Historie abgespeichert. Jeder Checkpoint kennt den Zustand der Datei zum Zeitpunkt der Modifikation, sowie einige Metadaten wie einen Zeitstempel, der

[4] Siehe auch: https://de.wikipedia.org/wiki/Inode
[5] Es wird nicht zwischen der Änderung eines einzelnen Bytes oder der gesamten Datei unterschieden wie bei `git`.

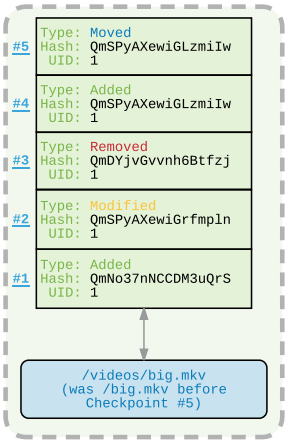


Abbildung 5.3.: Jede Datei und jedes Verzeichnis besitzt eine Liste von Checkpoints.

Dateigröße, dem Änderungstyp, dem Vorgänger und dem Urheber der Änderung. Der Vorteil einer dateiabhängigen Historie ist die Möglichkeit, umbenannte Dateien zu erkennen, sowie Dateien zu erkennen, die gelöscht und dann wieder hinzugefügt worden sind. Ein weiterer Vorteil ist, dass zur Ausgabe der Historie einer Datei, nur die *Checkpoints* betrachtet werden müssen. Es muss nicht wie bei `git` jeder Commit betrachtet werden, um nachzusehen ob eine Änderung an einer bestimmten Datei stattgefunden hat.

- **Staging–Commit:** Es existiert immer ein sogenannter *Staging–Commit*. Dieser beinhaltet alle Knoten im MDAG, die seit dem letzten »vollwertigen« Commit modifiziert worden sind. Abb. 5.4 zeigt den Staging–Bereich von `git` und `brig` im Vergleich. Im Falle von `git` handelt es sich um eine eigene, vom eigentlichen Graphen unabhängige, Datenstruktur, in die der Nutzer mittels `git add` explizit Dokumente aus dem Arbeitsverzeichnis hinzufügt. Bei `brig` hingegen gibt es kein Arbeitsverzeichnis und daher keine Unterscheidung zwischen »Unstaged Files« und »Staged Files«. Die Daten kommen entweder von einer externen Datei, welche mit `brig stage <filename>` dem Staging–Bereich hinzugefügt wurde, oder die Datei wurde direkt im FUSE–Dateisystem von `brig` modifiziert. In beiden Fällen wird die neue oder modifizierte Datei in den *Staging–Commit* eingegliedert, welcher aus diesem Grund eine veränderliche Prüfsumme aufweist und nach jeder inhaltlichen Modifikation auf ein anderes Wurzelverzeichnis verweist.

Da ein *Commit* nur einen Vorgänger haben kann, muss ein anderer Mechanismus eingeführt werden, um die Synchronisation zwischen zwei Partnern festzuhalten. Bei `git` wird dies mittels eines sogenannten *Merge–Commit* gelöst, welcher aus den Änderungen des Synchronisationspartners besteht. Hier wird das Konzept eines *Merge–Points* eingeführt. Innerhalb eines *Commit* ist das ein spezieller Marker, der festhält mit wem synchronisiert wurde und welchen Stand er zu diesem Zeitpunkt hatte. Bei einer späteren Synchronisation muss daher lediglich der Stand zwischen dem aktuellen *Commit* (»CURR«) und dem letzten Merge–Point verglichen werden. Basierend auf diesem Vergleich wird ein neuer *Commit* (der *Merge–Commit*) erstellt, der alle (möglicherweise nach der Konfliktauflösung zusammengeführten) Änderungen des Gegenübers enthält und als neuer *Merge–Point* dient.

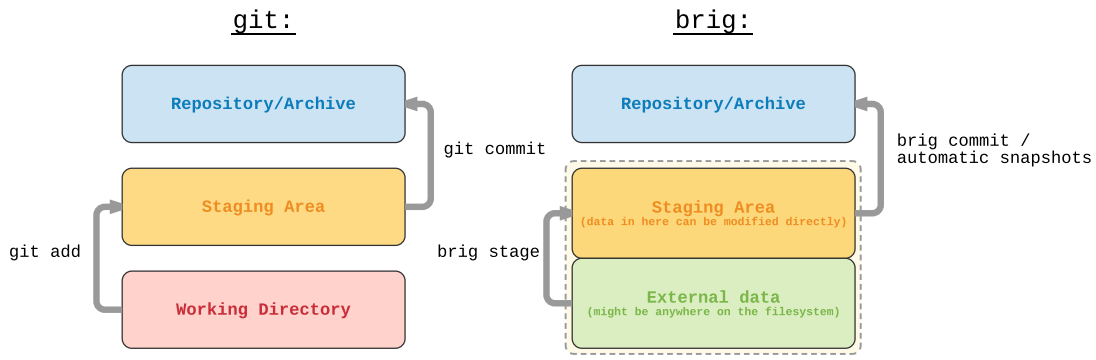


Abbildung 5.4.: Der Staging–Bereich im Vergleich zwischen `git` und `brig`

5.1.1. Operationen auf dem Datenmodell

Die Gesamtheit aller *Files*, *Directories*, *Commits*, *Checkpoints* und *Refs* wird im Folgenden als *Store* bezeichnet. Da ein *Store* nur aus Metadaten besteht, ist er selbst leicht auf andere Geräte übertragbar. Er kapselt den Objektgraphen und kümmert sich um die Verwaltung der Objekte. Basierend auf dem Store werden insgesamt elf verschiedene atomare Operationen implementiert, die jeweils den aktuellen Graphen nehmen und einen neuen und veränderten Graphen erzeugen.

Es gibt sechs Operationen, die die Benutzung des Graphen als gewöhnliches Dateisystem ermöglichen:

`STAGE`: Fügt ein Dokument dem Staging–Bereich hinzu oder aktualisiert die Version eines vorhandenen Dokuments. Der Pfad entscheidet dabei wo das Dokument eingefügt wird, beziehungsweise welches existierende Dokument modifiziert wird. Abb. 5.5 zeigt die Operationen, die zum Einfügen einer Datei notwendig sind. Als Vorarbeit muss allerdings erst die gesamte Datei gelesen werden und in das `ipfs`–Backend eingefügt werden. Die Datei wird zudem gepinnt. Als Ergebnis dieses Teilprozesses wird die Größe und Prüfsumme der verschlüsselten und komprimierten Datei zurückgeliefert. Handelt es sich bei dem hinzuzufügenden Objekt um ein Verzeichnis, wird der gezeigte Prozess für jede darin enthaltene Datei wiederholt.

`REMOVE`: Entfernt eine vorhandene Datei aus dem Staging–Bereich. Der Pin der Datei oder des Verzeichnisses und all seiner Kinder wird entfernt. Die gelöschten Daten werden möglicherweise beim nächsten Durchgang der $Cleanup$ Operation aus dem lokalen Speicher von `ipfs` entfernt. Die Prüfsumme der entfernten Datei wird aus den darüber liegenden Verzeichnissen herausgerechnet. Handelt es sich dabei um ein Verzeichnis, wird der Prozess *nicht* rekursiv für jedes Unterobjekt ausgeführt. Es genügt die Prüfsumme des zu löschenden Verzeichnisses aus den Eltern mittels der XOR–Operation herauszurechnen und die Kante zu dem Elternknoten zu kappen.

`LIST`: Entspricht konzeptuell dem Unix–Werkzeug `ls`. Besucht alle Knoten unter einem bestimmten Pfad rekursiv (breadth-first) und gibt diese aus.

`MKDIR`: Erstellt ein neues, leeres Verzeichnis. Die initiale Prüfsumme des neuen Verzeichnisses ergibt sich aus dem Pfad des neuen Verzeichnisses. Diese wird in den Elternknoten eingerechnet. Die Referenz auf das Wurzelverzeichnis wird im Staging–Commit angepasst. Eventuell müssen noch dazwischenliegende Verzeichnisse erstellt werden. Diese werden einzeln von oben nach unten mit

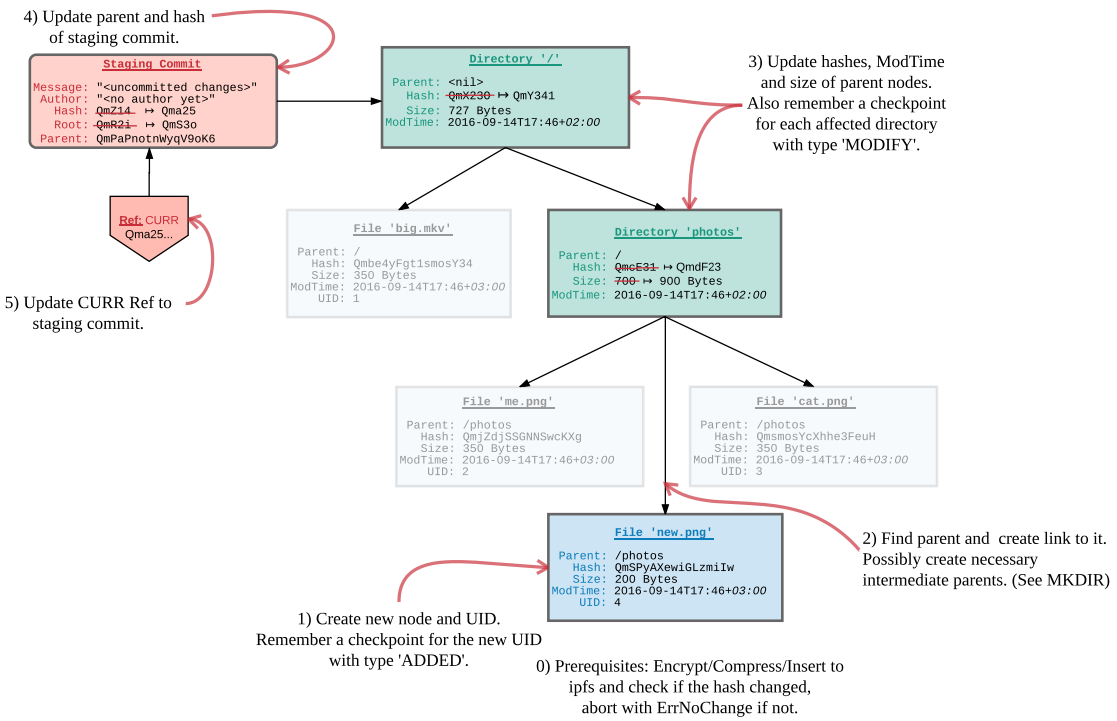


Abbildung 5.5.: Die Abfolge der STAGE-Operation im Detail

den eben beschriebenen Prozess erstellt.

MOVE: Verschiebt eine Quelldatei oder Verzeichnis zu einem Zielpfad. Es muss eine Fallunterscheidung getroffen werden, je nachdem ob und welcher Knoten im Zielpfad vorhanden ist:

1) *Ziel existiert noch nicht:* Quelldaten werden zum neuen Pfad verschoben.
2) *Ziel existiert und ist eine Datei:* Vorgang wird abgebrochen, es sei denn die Aktion wird »forciert«.
3) *Ziel existiert und ist ein Verzeichnis:* Quelldaten werden direkt unter das Zielverzeichnis verschoben, sofern darunter noch kein Verzeichnis mit diesem Namen existiert. Andernfalls wird die Aktion mit einem Fehler abgebrochen.

In jedem Fall entspricht diese Operation technisch dem, möglicherweise mehrfachen, sequentiellen Ausführen der Operationen REMOVE und ADD. Im Unterschied dazu ist sie im Ganzen atomar und erstellt einen Checkpoint mit dem Typen MOVED für alle verschobenen Knoten.

CAT: Gibt ein Dokument als einen Datenstrom aus. Der Name lehnt sich dabei an das Unix–Tool `cat` an, welches ebenfalls Dateien ausgeben kann. Es wird lediglich wie in Abb. 5.2 gezeigt der gesuchte Knoten per Pfad aufgelöst und die darin enthaltene Prüfsumme wird vom `ipfs`–Backend aufgelöst. Die ankommenden Daten werden entschlüsselt und dekomprimiert bevor sie dem Nutzer präsentiert werden.

Neben den oben stehenden Operationen, gibt es noch fünf weitere, die zur Versionskontrolle dienen und in dieser Form normalerweise nicht von Dateisystemen implementiert werden:

UNSTAGE: Entfernt ein Dokument aus dem Staging–Bereich und setzt den Stand auf den zuletzt bekannten Wert zurück (also der Stand der in HEAD präsent war). Die Prüfsumme des entfernten Dokumentes wird aus den Elternknoten herausgerechnet und dafür die alte Prüfsumme wieder eingerechnet.

Anmerkung: Die Benennung der Operationen STAGE, UNSTAGE und REMOVE ist anders als bei den semantisch gleichen git–Werkzeugen add, reset und rm. Die Benennung nach dem git–Schema ist irreführend, da `git add` nicht nur neue Dateien hinzufügt, sondern auch modifizierte Dateien aktualisiert. Zudem ist `git add` nicht das Gegenteil von `git rm`[6], wie man vom Namen annehmen könnte. Das eigentliche Gegenteil ist `git reset`. Eine mögliche Alternative zu `brig stage` wäre vermutlich auch `brig track`, beziehungsweise `brig untrack` statt `brig rm`.

COMMIT: Erstellt einen neuen Commit, basierend auf dem Inhalt des *Staging–Commits* (siehe auch Abb. 5.6 für eine Veranschaulichung). Dazu werden die Prüfsummen des aktuellen und des Wurzelverzeichnisses im letzten Commit (HEAD) verglichen. Unterscheiden sie sich nicht, wird abgebrochen, da keine Veränderung vorliegt. Im Anschluss wird der *Staging–Commit* finalisiert, indem die angegebene *Commit–Message* und der Autor in den Metadaten des Commits gesetzt werden. Basierend darauf wird die finale Prüfsumme berechnet und der entstandene Commit abgespeichert. Ein neuer *Staging-Commit* wird erstellt, welcher im unveränderten Zustand auf das selbe Wurzelverzeichnis zeigt wie sein Vorgänger. Zuletzt werden die Referenzen von HEAD und CURR jeweils um einen Platz nach vorne verschoben.

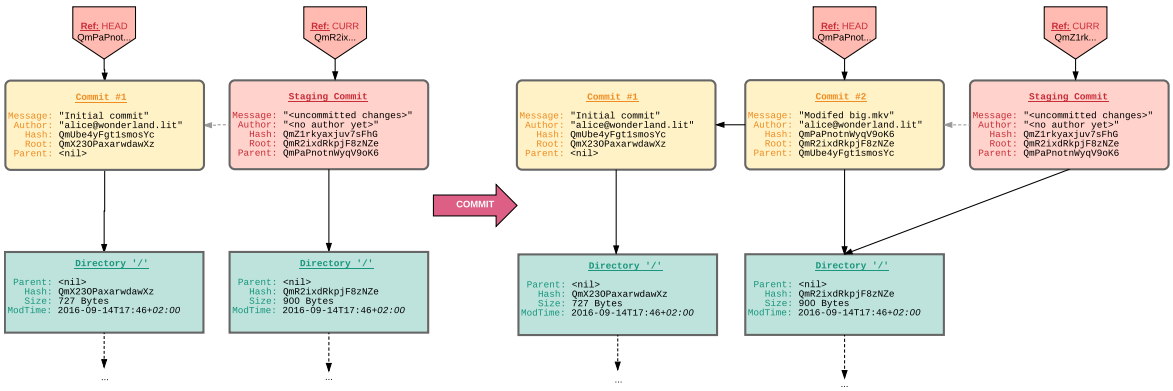


Abbildung 5.6.: Die Abfolge der COMMIT–Operation im Detail. Links der vorige Stand, rechts der Stand nach der COMMIT–Operation.

CHECKOUT: Stellt einen alten Stand wieder her. Dabei kann die Operation eine alte Datei oder ein altes Verzeichnis basierend auf der alten Prüfsumme oder den Stand eines gesamten, in der Vergangenheit liegenden, Commits wiederherstellen.

Im Gegensatz zu `git` ist es allerdings nicht vorgesehen, in der Versionshistorie »herumzuspringen«. Soll ein alter *Commit* wiederhergestellt werden, so wird der Staging–Commit so verändert, dass er dem gewünschten, alten Stand entspricht (siehe auch Abbildung Abb. 5.7). Das Verhalten von `brig` entspricht an dieser Stelle also nicht dem Namensvetter `git checkout` sondern eher dem wiederholten Anwenden von `git revert` zwischen dem aktuellen und dem Nachfolger des gewünschten

[6] Selbstkritik des git–Projekts: https://git.wiki.kernel.org/index.php/GitFaq#Why_is_.22git_rm.22_not_the_inverse_of_.22git_add.22.3F

Commits.

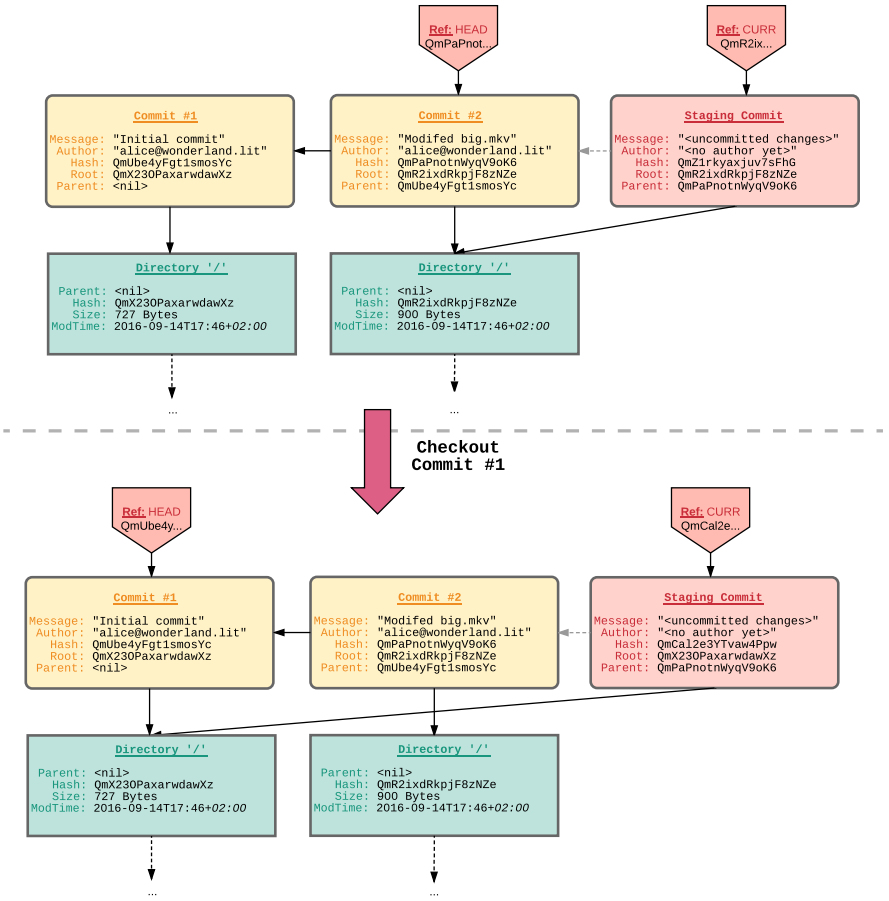


Abbildung 5.7.: Die Abfolge der CHECKOUT–Operation im Detail.

Begründet ist dieses Verhalten darin, dass kein sogenannter »Detached HEAD«–Zustand entstehen soll, da dieser für den Nutzer irreführend sein kann. Dieser Zustand kann in `git` erreicht werden, indem man in einen früheren *Commit* springt ohne einen neuen *Branch* davon abzuzweigen. Der HEAD zeigt dann nicht mehr auf einen benannten Branch, sondern auf die Prüfsumme des neuen Commits, der vom Nutzer nur noch durch die Kenntnis derselben erreichbar ist. Macht man in diesem Zustand Änderungen, ist es möglich die geänderten Daten zu verlieren[7]. Um das zu vermeiden, hält `brig` die Historie stets linear und unveränderlich. Dies stellt keine Einschränkung der Architektur an sich dar.

LOG/HISTORY: Zeigt alle Commits, bis auf den Staging–Commit. Begonnen wird die Ausgabe mit HEAD und beendet wird sie mit dem initialen Commit. Alternativ kann auch die Historie eines einzelnen Verzeichnisses oder einer Datei angezeigt werden. Dabei werden statt Commits alle Checkpoints dieser Datei, beginnend mit dem Aktuellsten, ausgegeben.

STATUS: Zeigt den Inhalt des aktuellen Staging–Commits (analog zu `git status`) und damit aller geänderten Dateien und Verzeichnisse im Vergleich zu HEAD. Es gibt keine eigene DIFF–Operation, da es keine partiellen Differenzen gibt. Eine Übersicht der Änderung erhält man durch Anwendung der STATUS und HISTORY–Operationen.

[7] So ist es bei `git` relativ einfach möglich in den sogenannten *Detached HEAD* Modus zu kommen, in dem durchaus Daten verloren gehen können. Siehe auch: http://gitfaq.org/articles/what-is-a-detached-head.html

5.2. Synchronisation

Ähnlich wie `git` speichert `brig` für jeden Nutzer seinen zuletzt bekannten *Store* ab. Mithilfe dieser Informationen können dann Synchronisationsentscheidungen größtenteils automatisiert getroffen werden. Welche Stores dabei lokal zwischengespeichert werden, entscheiden die Einträge der sogenannten *Remote–Liste*.

5.2.1. Die Remote–Liste

Jeder Teilnehmer mit dem synchronisiert werden soll, muss zuerst in eine spezielle Liste von `brig` eingetragen werden, damit dieser dem System bekannt wird. Dies ist vergleichbar mit der Liste die `git remote -v` erzeugt: Eine Zuordnung eines menschenlesbaren Namen zu einer eindeutigen Referenz zum Synchronisationspartner. Im Falle von `git` ist das eine URL, bei `brig` handelt es sich um die öffentliche Identität des Partners, also einer Prüfsumme. Wie später gezeigt wird, ist dieses explizite Hinzufügen des Partners eine Authentifizierungsmaßnahme, die bewusst eingefügt wurde. Unter Abschnitt 5.4.3 wird das Konzept genauer erläutert. Dadurch, dass nur mit authentifizierten Knoten Verbindungen aufgebaut werden, bildet `brig` ein *Private-Peer-to-Peer*–Netzwerk[8] auf Basis von `ipfs`.

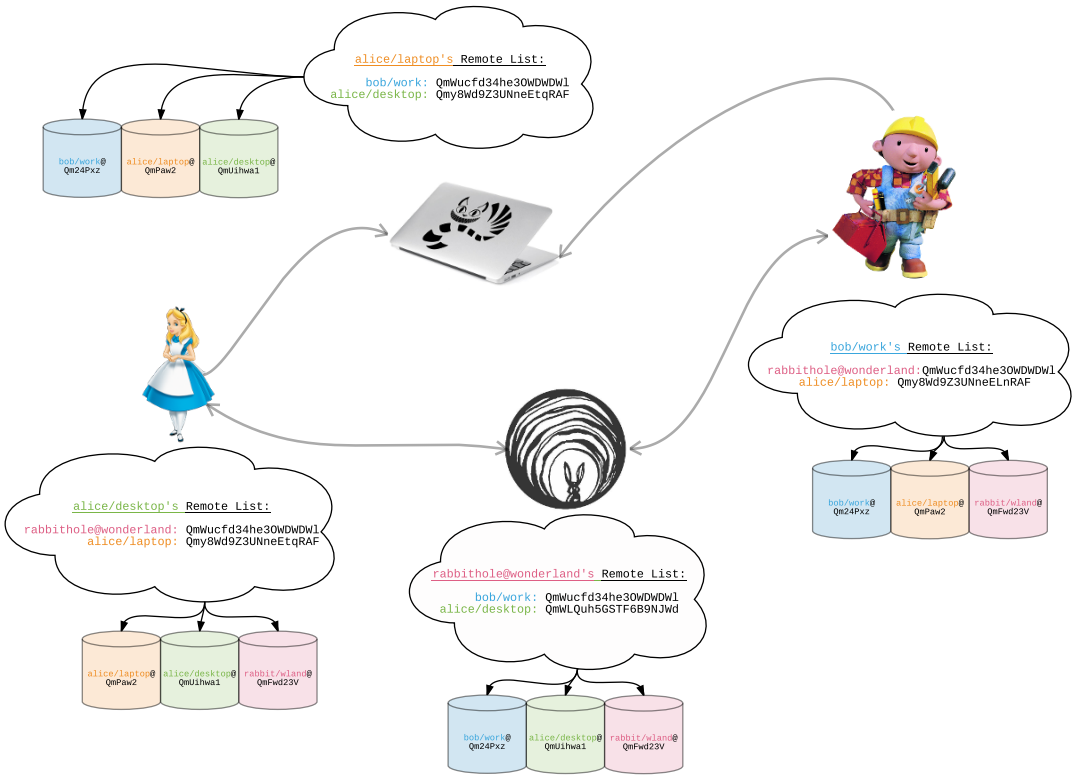


Abbildung 5.8.: Remote–Liste von vier Repositories und verschiendenen Synchronisationsrichtungen.

Wie in Abb. 5.8 gezeigt wird, können alle Knoten miteinander synchronisieren, die sich gegenseitig in die Liste eingetragen haben, da von diesen jeweils der zuletzt bekannte Store übertragen wurde. Die Synchronisation ist dabei, wie ein `git pull`, nicht bidirektional. Lediglich die eigenen Daten

[8] Siehe auch: https://en.wikipedia.org/wiki/Private_peer-to-peer

werden mit den Fremddaten zusammengeführt. Es gibt kein Äquivalent zu `git push`, welches die eigenen Daten zu einem Partner überträgt. Jeder Partner entscheidet selbst, mit welchen anderen Teilnehmern er synchronisiert, ohne dass seine eigenen Daten überschrieben werden können. In der Grafik wird zudem ein spezieller Anwendungsfall gezeigt: Das Repository `rabbithole@wonderland` ist eine gemeinsame Datenablage für zwei Parteien, die stets online verfügbar ist[9]. Dieses kann durch ein Skript automatisiert stets die Änderungen aller bekannten Teilnehmer synchronisieren und auch weitergeben, wenn der eigentliche Nutzer gerade offline ist. Dieses Vorgehen bietet sich vor allem dann an, wenn aufgrund der Zeitverschiebung zwei Nutzer selten zur selben Zeit online sind.

5.2.2. Synchronisation einzelner Dateien

In seiner einfachsten Form nimmt ein Synchronisationsalgorithmus als Eingabe die Metadaten zweier Dateien von zwei Synchronisationspartnern. Als Ausgabe trifft der Algorithmus auf dieser Basis eine der folgenden Entscheidungen:

1) Die Datei existiert nur bei Partner A.
2) Die Datei existiert nur bei Partner B.
3) Die Datei existiert bei beiden und ist gleich.
4) Die Datei existiert bei beiden und ist verschieden.

Je nach Entscheidung kann für diese Datei eine entsprechende Aktion ausgeführt werden:

1) Die Datei muss zu Partner B übertragen werden (falls bidirektionale Synchronisation gewünscht ist).
2) Die Datei muss zu Partner A übertragen werden.
3) Es muss nichts weiter gemacht werden.
4) Konfliktsituation: Auflösung nötig.

Bis auf den vierten Schritt ist die Implementierung trivial und kann von einem Computer erledigt werden. Das Kriterium, ob die Datei gleich ist, kann entweder durch einen direkten Vergleich der Daten gelöst werden (aufwendig) oder durch den Vergleich der Prüfsummen beider Dateien (schnell, aber vernachlässigbares Restrisiko durch Kollision). Manche Werkzeuge wie `rsync` setzen sogar auf Heuristiken, indem sie in der Standardkonfiguration aus Geschwindigkeitsgründen nur das Änderungsdatum und die Dateigröße vergleichen.

Für die Konfliktsituation hingegen kann es keine perfekte, allumfassende Lösung geben, da die optimale Lösung von der jeweiligen Datei und der Absicht des Nutzers abhängt. Bei Quelltext–Dateien möchte der Anwender vermutlich, dass beide Stände möglichst automatisch zusammengeführt werden, bei großen Videodateien ist das vermutlich nicht seine Absicht. Selbst wenn die Dateien nicht automatisch zusammengeführt werden sollen (englisch »to merge«), ist fraglich was mit der Konfliktdatei des Partners geschehen soll. Soll die eigene oder die fremde Version behalten werden? Dazwischen sind auch weitere Lösungen denkbar, wie das Anlegen einer Konfliktdatei (`photo.png:conflict-by-bob-2015-10-04_14:45`), so wie es beispielsweise Dropbox macht.[10] Alternativ könnte der Nutzer auch bei jedem Konflikt befragt werden. Dies wäre allerdings im Falle von

[9] Typischerweise würde ein solches Repository in einem Rechenzentrum liegen, oder auf einem privaten Server.
[10] Siehe https://www.dropbox.com/help/36

`brig` nach Meinung des Autors der Usability stark abträglich.

Im Falle von `brig` müssen nur die Änderungen von ganzen Dateien betrachtet werden, aber keine partiellen Änderungen darin. Eine Änderung der ganzen Datei kann dabei durch folgende Aktionen des Nutzers entstehen:

1) Der Dateiinhalt wurde modifiziert, ergo muss sich die Prüfsumme geändert haben (`MODIFY`).
2) Die Datei wurde verschoben, ergo muss sich der Pfad geändert haben (`MOVE`).
3) Die Datei wurde gelöscht, ergo ist sie im *Staging-Commit* nicht mehr vorhanden (`REMOVE`).
4) Die Datei wurde (initial oder erneut nach einem `REMOVE`) hinzugefügt (`ADD`).

Der vierte Zustand (`ADD`) ist dabei der Initialisierungszustand. Nicht alle dieser Zustände führen dabei automatisch zu Konflikten. So sollte beispielsweise ein guter Algorithmus kein Problem erkennen, wenn ein Partner die Datei modifiziert und der andere sie nicht verändert, sondern lediglich umbenennt. Eine Synchronisation der entsprechenden Datei sollte den neuen Inhalt mit dem neuen Dateipfad zusammenführen. Tabelle 5.1 zeigt welche Operationen zu Konflikten führen und welche verträglich sind. Die einzelnen Möglichkeiten sind dabei wie folgt:

- »✗«: Die beiden Aktionen sind nicht miteinander verträglich, es sei denn ihre Prüfsummen sind gleich. Sind die Prüfsummen gleich, heißt das, dass die exakt gleiche Änderung auf beiden Seiten gemacht wurde.
- »✎«: Die Aktion ist prinzipiell verträglich, hängt aber von der Konfiguration ab. Entweder wird die Löschung oder die Umbenennung vom Gegenüber propagiert (Standard) oder die eigene Datei wird behalten.
- »✓«: Die beiden Aktionen sind verträglich.

Tabelle 5.1.: Verträglichkeit der atomaren Operationen untereinander für die Partner **A** und **B**.

A/B	ADD	REMOVE	MODIFY	MOVE
ADD	✗	✎	✗	✗
REMOVE	✎	✓	✎	✎
MODIFY	✗	✎	✗	✓
MOVE	✗	✎	✓	✗

Zusammenfassend wird der in Listing 5.2.2 gezeigte Pseudo-Code von beiden Teilenehmern ausgeführt, um zwei Dateien synchron zu halten. Unten stehender Go-Pseudocode ist eine modifizierte Version aus Russ Cox' Arbeit »File Synchronization with Vector Time Pairs«[16], welcher für `brig` angepasst wurde. Die Funktionen `HasConflictingChanges()` und `ResolveConflict()` prüfen dabei die Verträglichkeit mithilfe von Tabelle 5.1.

```
// historyA ist die Historie der eigenen Datei A.
// historyB ist die Historie der fremden Datei B mit gleichem Pfad.
func sync(historyA, historyB History) Result {
    if historyA.Equal(historyB) {
        // Keine weitere Aktion nötig.
```

```
        return NoConflict
    }

    // Prüfe, ob historyA mit den Checkpoints von historyB beginnt.
    if historyA.IsPrefix(historyB) {
        // B hängt A hinterher.
        return NoConflict
    }

    if historyB.IsPrefix(historyA) {
        // A hängt B hinterher. Kopiere B zu A.
        copy(B, A)
        return NoConflict
    }

    if root := historyA.FindCommonRoot(historyB); root != nil {
        // A und B haben trotzdem eine gemeinsame Historie,
        // haben sich aber auseinanderentwickelt.
        if !historyA.HasConflictingChanges(historyB, root) {
            // Die Änderungen sind verträglich und
            // können automatisch aufgelöst werden.
            ResolveConflict(historyA, historyB, root)
            return NoConflict
        }
    }

    // Keine gemeinsame Historie.
    // -> Nicht automatisch zusammenführbar.
    // -> Eine Konfliktstrategie muss angewandt werden.
    return Conflict
}
```

5.2.3. Synchronisation von Verzeichnissen

Die naive Herangehensweise wäre, den obigen Algorithmus für jede Datei im Verzeichnis zu wiederholen[^SYNC_ONLY_FILE]. Der beispielhafte Verzeichnisbaum in Abb. 5.9 zeigt allerdings bereits ein Problem dabei: Die Menge an Pfaden, die Alice besitzt wird sich selten ganz mit denen decken, die Bob besitzt. So kann natürlich Alice Pfade besitzen, die Bob nicht hat und umgekehrt. Im Beispiel synchronisiert Alice mit Bob. Das heißt, Alice möchte die Änderungen von Bob empfangen.

Man könnte also das »naive« Konzept weiterführen und die Menge der zu synchronisierenden Pfade in drei Untermengen unterteilten. Jede dieser Untermengen hätte dann eine unterschiedliche Semantik:

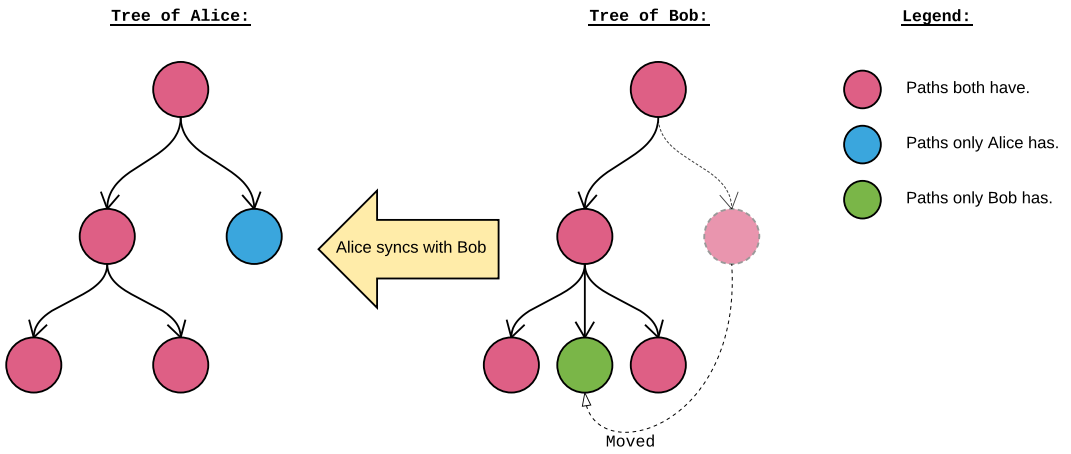


Abbildung 5.9.: Unterteilung der zu synchronisierenden Pfade in drei Gruppen.

- Pfade die beide haben ($X = Paths_A \bigcap Paths_B$): Konfliktpotenzial. Führe obigen Algorithmus für jede Datei aus. Pfade die nur Alice hat ($Y = Paths_A \setminus Paths_B$): Brauchen keine weitere Behandlung. - Pfade die nur Bob hat ($Z = Paths_B \setminus Paths_A$): Müssen nur hinzugefügt werden.

Wie in Abb. 5.9 angedeutet, sind diese Mengen allerdings schwerer zu bestimmen als durch eine simple Vereinigung, beziehungsweise Differenz. Zwei Beispiele verdeutlichen dies:

- Löscht Bob eine Datei, während Alice sie nicht verändert, würde der Pfad trotzdem in der Menge Y landen. Dies hätte zur Folge, dass die Löschung nicht zu Alice propagiert wird.
- Verschiebt Bob eine Datei zu einem neuen Pfad, muss dieser neue Pfad trotzdem mit dem alten Pfad von Alice verglichen werden, um die Umbenennung zusammenzuführen. In Abb. 5.9 sollte also der blaue Knoten mit dem grünen verglichen werden.

Es muss also eine Abbildungsfunktion gefunden werden, die jedem Pfad von Alice einen Pfad von Bob zuordnet. Die Wertemenge dieser Funktion entspricht der Menge X, also aller Pfade die einer speziellen Konfliktauflösung bedürfen. Die Menge Z (also alle Pfade die Bob hat, aber Alice nicht) ergibt sich dann einfach durch $Z = Paths_B \setminus X$. Für die Abbildung der Pfade von Alice zu Bob's Pfaden funktioniert die Abbildungsfunktion folgendermaßen:

1) Aus Bob's Store werden alle Knoten gesammelt, die sich seit dem letzten gemeinsamen Merge–Point verändert haben. Falls es noch keinen gemeinsamen Merge–Point gab, werden alle Knoten angenommen.
2) Aus Bob's Store wird für jeden Knoten die Historie (= Liste aller Checkpoints) seit dem letzten Merge–Point gesammelt, oder die gesamte Historie (= alle Checkpoints) falls es noch keinen Merge–Point gab.
3) Es wird eine Abbildung (als assoziatives Array) erstellt, die alle bekannten Pfade von Bob der jeweiligen Historie zuordnen, in dem der Pfad vorkommt. Mehr als ein Pfad kann dabei auf die gleiche Historie zeigen, wenn Verschiebungen vorkamen. Innerhalb dieser Spanne gelöschte Dateien sind in der Abbildung unter ihrem zuletzt bekannten Pfad zu finden.
4) Für alle Pfade, die Alice momentan besitzt (Alle Pfade unter `HEAD`), wird der Algorithmus in

Listing 5.2.3 ausgeführt. Dieser ordnet jedem Pfad von Alice, einem Pfad von Bob zu oder meldet, dass er kein passendes Gegenstück finden konnte.

```
// Ein assoziatives Array mit dem Pfad zu der Historie
// seit dem letzten gemeinsamen Merge-Point.
type PathToHistory map[string]History

// BobMapping enthält alle Pfade;
// also auch Pfade die entfernt wurden (unter ihrem letzten Namen)
// Wurden Pfade verschoben, so enthält das Mapping auch alle Zwischenschritte.
func MapPath(HistA History, BobMapping PathToLastHistory) (string, error) {
    // Iteriere über alle Zwischenpfade, die `HistA` hatte.
    // In den meisten Fällen (ohne Verschiebungen) also nur ein einziger.
    for _, path := range HistA.AllPaths() {
        HistB, ok := BobMapping[path]

        // Diesen Pfad hatte Bob nicht.
        if !ok {
            continue
        }

        // Erfolg! Gebe den aktuellsten Pfad von Bob zurück.
        // Also der Pfad an dem die Datei zuletzt bei Bob war,
        // beziehungsweise der Pfad des aktuellsten Checkpoints.
        return HistB.MostCurrentPath(), nil
    }

    // Bob hat diesen Pfad nirgends.
    // -> Es muss ein Pfad sein, den nur Alice hat.
    return "", ErrNoMappingFound
}
```

Das Ergebnis dieses Vorgehens ist eine Abbildung aller Pfade von Alice zu den Pfaden von Bob. Damit wurde eine eindeutige Zuordnung erreicht und die einzelnen Dateien können mit dem Algorithmus unter Abschnitt 5.2.2 synchronisiert werden. Die Dateien, die Bob zusätzlich hat (aber Alice nicht) können nun leicht ermittelt werden, indem geprüft wird welche von Bob's Pfaden noch nicht in der errechneten Wertemenge der Abbildung vorkommen. Diese Pfade können dann in einem zweiten Schritt dem Stand von Alice hinzugefügt werden.

Darüber hinaus gibt es noch einen Spezialfall, der vor der eigentlichen Synchronisation abgeprüft werden muss. Hat einer der beiden Partner keine Änderungen gemacht und haben beide Partner eine gemeinsame Historie, kann der Stand »vorgespult« werden. Das heißt, alle Änderungen der Gegenseite können direkt übernommen werden. Dieses Vorgehen ist bei `git` auch als *Fast-Forward-Merge* bekannt (`git merge --ff`). Anders als bei `git` wird bei `brig` allerdings immer ein Merge-Point

erstellt, weswegen dies nur eine algorithmische Optimierung darstellt.

5.2.4. Austausch der Metadaten

Um die Metadaten nun tatsächlich synchronisieren zu können, muss ein Protokoll etabliert werden, mit dem zwei Partner ihren Store über das Netzwerk austauschen können. Im Folgenden wird diese Operation, analog zum gleichnamigen `git`–Kommando[11], `brig fetch` genannt.

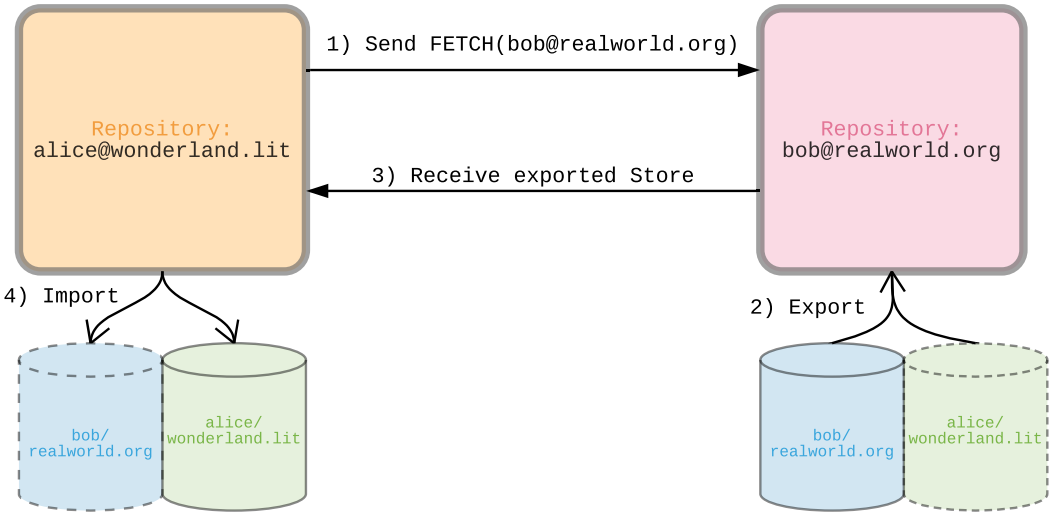


Abbildung 5.10.: Das Protokoll das bei der `FETCH`–Operation ausgeführt wird.

Wie in Abb. 5.10 gezeigt, besteht das Protokoll aus drei Teilen:

- Alice schickt eine `FETCH`–Anfrage zu Bob, der den Namen des zu holenden Stores enthält. Im Beispiel ist dies Bob's eigener Store, `bob@realworld.org`.
- Falls Alice in Bob's Remote–Liste steht, wandelt Bob seinen eigenen Store in eine exportierbare Form um, die aus einer großen serialisierten Nachricht[12] besteht, die alle notwendigen Daten enthält.
- Die serialisierte Form des Stores wird über den Transfer–Layer von `brig` (siehe Abschnitt 5.4.2) zurück an `alice@wonderland.lit` geschickt.
- Alice importiert die serialisierte Form in einen neuen, leeren Store und speichert das Ergebnis in der Liste der Stores ihrer Kommunikationspartner. Eine Synchronisation der beiden Metadatensätze kann nun lokal bei Alice erfolgen.

Aus Zeitgründen ist dieses Protokoll momentan noch sehr einfach gehalten und beherrscht keine differentiellen Übertragungen. Da hier nur Metadaten übertragen werden sollte das nur bedingt ein Problem sein. In der Tat müssten aber nur die Commits seit dem letzten gemeinsamen Merge–Point übertragen werden.

Auch sind zum momentanen Stand noch keine *Live–Updates* möglich. Hierfür müssten sich die einzelnen Knoten bei jeder Änderung kleine *Update–Pakete* schicken, welche einen einzelnen *Checkpoint*

[11] https://git-scm.com/book/be/v2/Git-Internals-Transfer-Protocols
[12] Die Form des serialisierten Export–Formats ist nicht weiter interessant und kann im Anhang Anhang B eingesehen werden (Message: *Store*).

beinhalten würden. Diese Checkpoints müssten dann jeweils in den aktuellen Staging–Bereich eingepflegt werden. Dadurch wären Änderungen in »Echtzeit« auf anderen Knoten verfügbar. Aus Zeitgründen wird an dieser Stelle aber nur auf diese Möglichkeit verwiesen; eine konzeptuelle Implementierung hierzu steht noch aus.

5.2.5. Abgrenzung zu anderen Synchronisationswerkzeugen

In der Fachliteratur (vgl. unter anderem [16]) findet sich zudem die Unterscheidung zwischen *informierter* und *uninformierter* Synchronisation. Der Hauptunterschied ist, dass bei ersterer die Änderungshistorie jeder Datei als zusätzliche Eingabe zur Verfügung steht. Auf dieser Basis können dann intelligentere Entscheidungen bezüglich der Konflikterkennung getroffen werden. Insbesondere können dadurch aber leichter die Differenzen zwischen den einzelnen Ständen ausgemacht werden: Für jede Datei muss dabei lediglich die in Listing 5.2.2 gezeigte Sequenz abgelaufen werden, die von beiden Synchronisationspartnern unabhängig ausgeführt werden muss. Werkzeuge wie `rsync` oder `unison` betreiben eine *uninformierte Synchronisation*. Sie müssen bei jedem Programmlauf Metadaten über beide Verzeichnisse sammeln und darauf arbeiten.

5.2.6. Speicherquoten

Werden immer mehr Modifikationen gespeichert, so steigt der Speicherverbrauch immer weiter an, da ohne ein Differenzmechanismus jede Datei pro Version einmal voll abgespeichert werden muss. Die Anzahl der Objekte die dabei gespeichert werden können, hängt von dem verfügbaren Speicherplatz ab. Sehr alte Versionen werden dabei typischerweise nicht mehr benötigt und können bei Platzbedarf gelöscht werden. Diese Aufgabe wird derzeit nicht von `brig` selbst übernommen, sondern vom `ipfs`–Backend. Dieses unterstützt mit dem Befehl `ipfs gc` eine Bereinigung von Objekten, die keinen Pin mehr haben. Zudem kann `brig` den Konfigurationswert `Datastore.StorageMax` von `ipfs` auf eine maximale Höhe (minus einen kleinen Puffer für `brig`–eigene Dateien) setzen. Wird dieser überschritten, geht der Garbage–Collector aggressiver vor und löscht nicht gepinnte Objekte sofort. In der momentanen Architektur und Implementierung sind allerdings zu diesem Zeitpunkt noch keine Speicherquoten vorhanden.

Eine Möglichkeit den Speicherverbrauch zu reduzieren, wäre die Einführung von *Packfiles*, wie `git` sie implementiert[13]. Diese komprimieren nicht eine einzelne Datei, sondern packen mehrere Objekte in ein zusammengehöriges Archiv. Dies kann die Kompressionsrate stark erhöhen wenn viele ähnliche Dateien (beispielsweise viele subtil verschiedene Versionen der gleichen Datei) zusammen gepackt werden. Nachteilig sind die langsameren Zugriffszeiten. Eine Implementierung dieser Lösung müsste zwischen eigentlichem Datenmodell und dem `ipfs`–Backend eine weitere Schicht einschieben, welche transparent und intelligent passende Dateien in ein Archiv verpackt und umgekehrt auch wieder entpacken kann. Diese Idee wird in Abschnitt 8.5.1 noch einmal aufgegriffen.

5.3. Architekturübersicht

Um den eigentlichen Kern des Store sind alle anderen Funktionalitäten gelagert. Abb. 5.11 zeigt diese in einer Übersicht. Die einzelnen Unterdienste werden im Folgenden besprochen.

[13] Mehr Details zur `git`–Implementierung hier: https://git-scm.com/book/uz/v2/Git-Internals-Packfiles

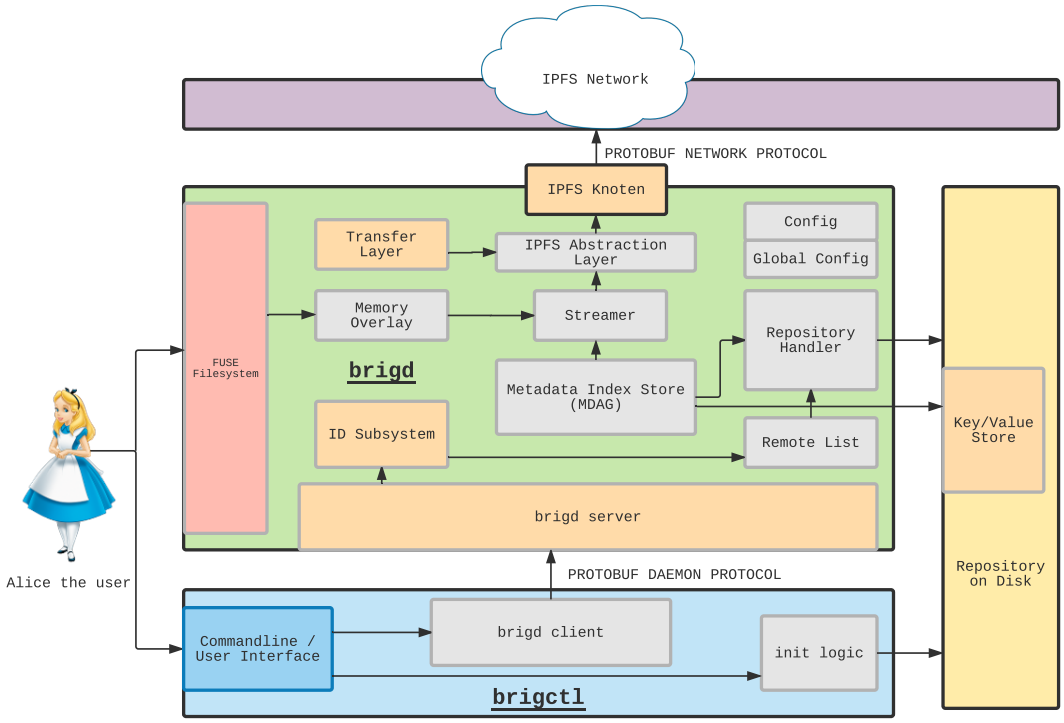


Abbildung 5.11.: Übersicht über die Architektur von `brig`.

5.3.1. Lokale Aufteilung in Client und Daemon

`brig` ist architektonisch in einem langlebigen Daemon–Prozess und einem kurzlebigen Kontroll–Prozess aufgeteilt, welche im Folgenden jeweils `brigd` und `brigctl` genannt werden[14]. Beide Prozesse kommunizieren dabei über das Netzwerk mit einem speziellen Protokoll, welches auf einen Serialisierungsmechanismus von Google namens *Protobuf*[15] basiert. Dabei wird basierend auf einer textuellen Beschreibung des Protokolls (einer `.proto`–Datei mit eigener Syntax) Quelltext in der gewünschten Zielsprache generiert. Dieser Quelltext ist dann in der Lage, Datenstrukturen von der Zielsprache in ein serialisiertes Format zu überführen, beziehungsweise dieses wieder einzulesen. Als Format steht dabei wahlweise eine speichereffiziente, binäre Repräsentation der Daten zur Verfügung, oder eine menschenlesbare Darstellung als JSON–Dokument.

Nötig ist die Aufteilung vor allem, da `brigd` im Hintergrund als Netzwerkdienst laufen muss, um Anfragen von Außen verarbeiten zu können. Auch läuft `ipfs` im selben Prozess wie `brigd` und muss daher stets erreichbar sein. Abgesehen davon ist es aus Effizienzgründen förderlich, wenn nicht bei jedem eingetippten Kommando das gesamte Repository geladen werden muss. Auch ist es durch die Trennung möglich, dass `brigd` auch von anderen Programmiersprachen und Prozessen auf dem selben Rechner aus gesteuert werden kann. Verbindungen von außen sollten aus Sicherheitsgründen nicht angenommen werden. Unter unixoiden Betriebssystemen wäre eine Alternative zu normalen Netzwerksockets die Nutzung von Unix–Domain–Sockets[16]. Diese sind als Datei im Dateisystem erreichbar und können daher mit entsprechenden Zugriffsrechten nur von bestimmten Nutzern

[14] Tatsächlich gibt es derzeit keine ausführbaren Dateien mit diesen Namen. Die Bezeichnungen `brigctl` und `brigd` dienen lediglich der Veranschaulichung.
[15] Mehr Informationen unter: https://developers.google.com/protocol-buffers
[16] Siehe auch: https://en.wikipedia.org/wiki/Unix_domain_socket

benutzt werden.

5.3.2. brigctl: Aufbau und Aufgabe

Zusammengefasst ist `brigctl` eine »Fernbedienung« für `brigd`, welche im Moment exklusiv von der Kommandozeile aus bedient wird. In den meisten Fällen verbindet sich der Kommando–Prozess `brigctl` beim Start zu `brigd`, sendet ein mittels *Protobuf* serialisiertes Kommando und wartet auf die dazugehörige Antwort welche dann deserialisiert wird. Nachdem die empfangene Antwort, je nach Art, ausgewertet wurde, beendet sich der Prozess wieder.

Protobuf Protokoll: Das Protokoll ist dabei so aufgebaut, dass für jede Aufgabe, die `brigd` erledigen soll ein separates Kommando existiert. Neben einer allgemeinen Typbezeichnung, können auch vom Kommando abhängige optionale und erforderliche Parameter enthalten sein. Ein gekürzter Auszug aus der Protokollspezifikation veranschaulicht dies in Listing 5.3.2.

```
enum MessageType {
    ADD = 0;
    // ...
}

message Command {
    // Typ des Kommandos
    required MessageType command_type = 1;

    message AddCmd {
        // Absoluter Pfad zur Datei auf der Festplatte des Nutzers.
        required string file_path = 1;

        // Pfad innerhalb von brig (/photos/me.png)
        required string repo_path = 2;

        // Füge Verzeichnisse rekursiv hinzu? (Standard: Ja)
        optional bool recursive = 3;
    }
    // ... weitere Subkommandos ...

    // Falls der Typ ADD war, lese von 'add_commando'
    optional AddCmd add_command = 2;
    // ... weitere Kommandoeinträge ...
}
```

Analog dazu kann `brigd` mit einer *Response* auf ein *Command* antworten. In Listing 5.3.2 wird beispielhaft die Antwortspezifikation (`OnlineStatusResp`) auf ein `OnlineStatusCmd`–Kommando gezeigt, welches prüft, ob `brigd` Verbindungen von Außen annimmt.

```
message Response {
    // Typ der Antwort
    required MessageType response_type = 1;

    // Wahr, falls es keine Fehlerantwort ist
    required bool success = 2;

    // Bei einem Fehler wird ein optionale Fehlerbeschreibung angegeben.
    optional string error = 3;

    // Detaillierter Fehlercode (noch nicht benutzt)
    optional id errno = 4;

    message OnlineStatusResp {
        required bool is_online = 1;
    }
    // ... Mehr Unterantworten ...

    optional OnlineStatusResp online_status_resp = 5;
    // ... Mehr Antworteinträge ...
}
```

Neben der Kommunikation mit `brigd` muss `brigctl` noch drei andere Aufgaben erledigen:

- **Initiales Anlegen eines Repositories:** Bevor `brigd` gestartet werden kann, muss die in Abb. 6.5 gezeigte Verzeichnisstruktur angelegt werden.
- **Bereitstellung des User–Interfaces:** Das zugrundeliegende Protokoll wird so gut es geht vom Nutzer versteckt und Fehlermeldungen müssen möglichst gut beschrieben werden.
- **Autostart von `brigd`:** Damit der Nutzer nicht explizit `brigd` starten muss, sollte der Daemon–Prozess automatisch im Hintergrund gestartet werden, falls er noch nicht erreichbar ist. Dies setzt `brigctl` um, indem es dem Nutzer nach dem Passwort zum Entsperren eines Repositories fragt und das Passwort beim Start an `brigd` weitergibt, damit der Daemon–Prozess das Repository entsperren kann.

5.3.3. `brigd`: Aufbau und Aufgabe

Der Daemon–Prozess implementiert alle Kernfunktionalitäten. Die einzelnen Komponenten werden in Abschnitt 5.4 beschrieben.

Als Netzwerkdienst muss `brigd` auf einem bestimmten Port (momentan standardmäßig Port `6666` auf `127.0.0.1`) auf Anfragen warten. Es werden keine Anfragen von Außen angenommen, da über diese lokale Verbindung fast alle sicherheitskritischen Informationen ausgelesen werden können. Für den Fall, dass ein Angreifer den lokalen Netzwerkverkehr mitlesen kann wird der gesamte Netzwerkverkehr zwischen `brigctl` und `brigd` mit AES256 verschlüsselt. Der Schlüssel wird beim Verbindungsaufbau mittels Diffie–Hellmann ausgetauscht. Die Details des Protokolls werden in [1]

beschrieben.

Die Anzahl der gleichzeitig offenen Verbindungen wird auf ein Maximum von `50` limitiert und Verbindungen werden nach Inaktivität mit einer Zeitüberschreitung von 10 Sekunden automatisch getrennt. Diese Limitierungen soll verhindern, dass fehlerhafte Clients den Hintergrundprozess zu stark auslasten.

Im selben Prozess wie `brigd` läuft auch der `ipfs`–Daemon und nutzt dabei standardmäßig den Port `4001`, um sich mit dem Netzwerk zu verbinden. Nachteilig an diesem Vorgehen ist, dass ein Absturz oder eine Sicherheitslücke in `ipfs` auch `brigd` betreffen kann. Längerfristig sollten beide Prozesse möglichst getrennt werden, auch wenn dies aus Effizienzgründen nachteilig ist.

5.4. Einzelkomponenten

Im Folgenden werden die einzelnen Komponenten von `brigd` aus architektonischer Sicht erläutert. Genauere Angaben zu Implementierungsdetails, insbesondere zum FUSE–Dateisystem, folgen im nächsten Kapitel.

5.4.1. Dateiströme

Im `ipfs`–Backend werden nur verschlüsselte und zuvor komprimierte Datenströme gespeichert. Verschlüsselung ist bei `brig` nicht optional. Hat ein Angreifer die Prüfsumme einer Datei erbeutet, so kann er die Datei aus dem `ipfs`–Netzwerk empfangen. Solange die Datei aber verschlüsselt ist, so wird der Angreifer alleine mit den verschlüsselten Daten ohne den dazugehörigen Schlüssel nichts anfangen können. In der Tat unterstützt er das `ipfs`–Netzwerk sogar, da der Knoten des Angreifers auch wieder seine Bandbreite zum Upload anbieten muss, da der Knoten sonst ausgebremst wird. Aus diesem Grund ist es aus Sicherheitsperspektive keine Notwendigkeit, `brig` in einem abgeschotteten Netzwerk zu betreiben. Standardmäßig verbindet sich `brig` mit dem weltweiten `ipfs`–Netzwerk, indem es die standardmäßig eingetragenen Bootstrap–Knoten kontaktiert.

Nachteilig an einer »zwangsweisen« Verschlüsselung ist, dass die Deduplizierungsfähigkeit von `ipfs` ausgeschaltet wird. Wird die selbe Datei mit zwei unterschiedlichen Schlüsseln verschlüsselt, so werden die resultierenden Daten (bis auf ihre Größe) keine Ähnlichkeit besitzen, sind also kaum deduplizierbar. Trotzdem ist die Unterteilung in Blöcke durch `ipfs` sinnvoll, da dadurch bereits heruntergeladene Blöcke nicht ein zweites Mal besorgt werden müssen. So lässt sich der Download von großen Dateien unterbrechbar und wieder fortsetzbar gestalten.

Eine mögliche Lösung wäre ein Verfahren namens *Convergent Encryption*[17]. Dabei wird der Schlüssel der zu verschlüsselnden Datei aus der Prüfsumme derselben Datei abgeleitet. Dies hat den Vorteil, dass gleiche Dateien auch den gleichen (deduplizierbaren) Ciphertext generieren. Der Nachteil ist, dass ein Angreifer feststellen kann, ob jemand eine Datei (beispielsweise Inhalte mit urhebergeschützten Inhalten) besitzt. Im Protoypen werden die Dateischlüssel daher zufällig generiert, was die Deduplizierungsfunktion von `ipfs` momentan ausschaltet. Dies hat auch zur Folge, dass die Synchronisation von zwei unabhängig hinzugefügten, aber sonst gleichen Dateien zwangsweise dazu führt, dass diese unterschiedlich sind, da auf beiden Seiten jeweils ein anderer Schlüssel generiert wird. Die Vor- und Nachteile dieses Verfahrens wird weiter in [1] diskutiert.

5.4.1.1. Verschlüsselung

Für `brig` wurde ein eigenes Containerformat für verschlüsselte Daten eingeführt, welches wahlfreien Zugriff auf beliebige Bereiche der verschlüsselten Datei erlaubt, ohne die gesamte Datei entschlüsseln zu müssen. Dies ist eine wichtige Eigenschaft für die Implementierung des FUSE–Dateisystems und ermöglicht zudem aus technischer Sicht das Streaming von großen, verschlüsselten Dateien wie Videos. Zudem kann das Format durch den Einsatz von *Authenticated Encryption* (*AE*, [18]) die Integrität der verschlüsselten Daten sichern.

Es werden lediglich reguläre Dateien verschlüsselt. Verzeichnisse existieren nur als Metadaten und werden nicht von `ipfs` gespeichert. Die Details und Entscheidungen zum Design des Formats werden in [1] dargestellt.

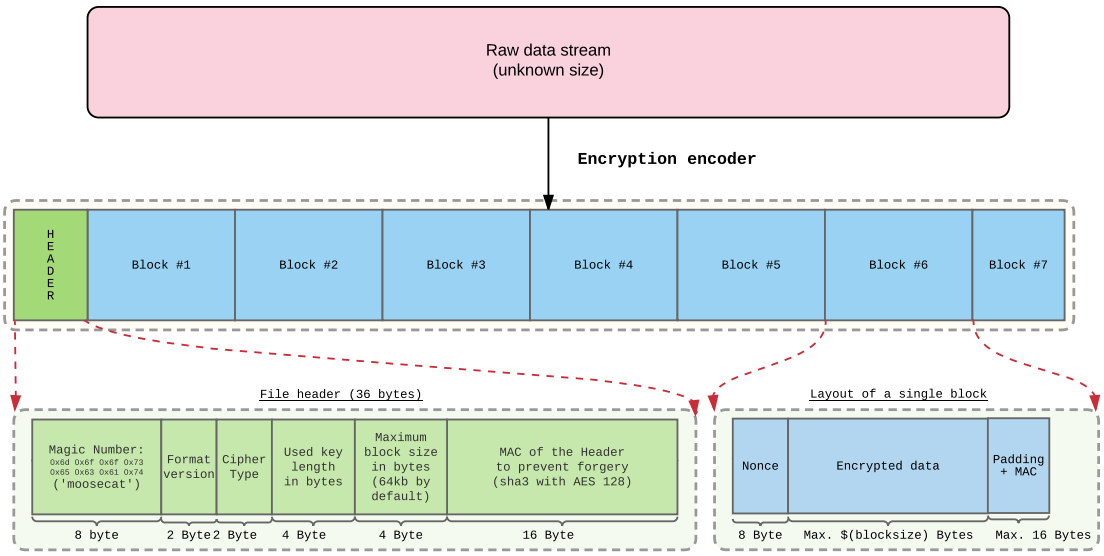


Abbildung 5.12.: Aufbau des Verschlüsselungs–Dateiformats.

Enkodierung: Abb. 5.12 zeigt den Aufbau des Formats. Ein roher Datenstrom (dessen Länge nicht bekannt sein muss) wird an den Enkodierer gegeben. Als weitere Eingabe muss ein Algorithmus ausgewählt werden und ein entsprechend dimensionierter, symmetrischer Schlüssel mitgegeben werden. Werden die ersten Daten geschrieben, so schreibt der Kodierer zuerst einen 36–Byte großen Header. In diesem finden sich folgende Felder:

- Eine *Magic–Number*[17] (8 Byte, ASCII–Repräsentation von `moosecat`) zur schnellen Identifikation einer von `brig` geschriebenen Datei.
- Die *Versionsnummer* (2 Byte) des vorliegenden Formats. Standardmäßig »`0x01`«. Sollten Änderungen am Format nötig sein, so müssen nur die ersten 10 Byte beibehalten werden und die Versionsnummer inkrementiert werden. Für die jeweilige Version kann dann ein passender Dekodierer genutzt werden.
- Die verwendete *Blockchiffre* (siehe [1]) (2 Byte) zur Verschlüsselung. Standardmäßig wird *ChaCha20/Poly1305* (siehe [19]) eingesetzt, aber es kann auch AES (siehe [20], S. 116 ff.) mit 256 Bit Schlüssellänge im Galois–Counter–Modus (GCM, siehe [1]) verwendet werden.

[17] Siehe auch: https://de.wikipedia.org/wiki/Magische_Zahl_(Informatik)

- Die *Länge* (4 Byte) des verwendeten Schlüssels in Bytes.
- Die *maximale Blockgröße* (4 Byte) der nachfolgenden Blöcke in Bytes.
- Ein *Message–Authentication–Code (MAC, siehe auch [20], S. 205 ff.)* (16 Byte) der die Integrität des Headers sicherstellt.

Nachdem der Header geschrieben wurde, sammelt der Enkodierer in einem internen Puffer ausreichend viele Daten, um einen zusammenhängenden Block zu schreiben (standardmäßig 64 Kilobyte). Ist diese Datenmenge erreicht, wird der Inhalt des Puffers verschlüsselt und ein kompletter Block ausgegeben. Dieser enthält folgende Felder:

- Eine *Nonce* (siehe auch [20], S. 263) (8 Byte). Diese eindeutige Nummer wird bei jedem geschriebenen Block inkrementiert und stellt daher die Blocknummer dar. Sie wird benutzt, um die Reihenfolge des geschriebenen Datenstroms zu validieren und wird zudem als öffentlich bekannte Eingabe für den Verschlüsselungsalgorithmus benutzt.
- Die eigentlichen, verschlüsselten Daten. Diese sind maximal so lang wie die maximale Blockgröße, können aber im Falle des letzten Blockes kleiner sein.
- Am Ende kann, je nach Algorithmus, ein gewisse Überlänge durch *Padding* entstehen. Zudem wird an jeden Block eine weitere MAC angehängt, welche die Integrität der Nonce und der nachfolgenden, verschlüsselten Daten sicherstellt.

So wird blockweise weiter verfahren, bis alle Daten des Ursprungsdatenstroms aufgebraucht worden sind. Der letzte Block darf als einziger kleiner als die maximale Blockgröße sein. Der resultierende Datenstrom ist etwas größer als der Eingabedatenstrom. Seine Größe lässt sich wie in Gleichung (5.1) gezeigt mithilfe der Eingabegröße s der Datei in Bytes und der Blockgröße b berechnen:

$$f_{\text{size}}(s) = 36 + s + \left\lceil \frac{s}{b} \right\rceil \times (8 + 16) \tag{5.1}$$

Was den Speicherplatz angeht, hält sich der »Overhead« in Grenzen. Zwar wächst eine fast leere Datei von 20 Byte Originalgröße auf 80 Byte nach der Verschlüsselung, aber bereits eine 20 Megabyte große Datei wächst nur noch um zusätzliche 7.5 Kilobyte ($+0.03\%$).

Dekodierung: Beim Lesen der Datei wird zuerst der Header ausgelesen und auf Korrektheit geprüft. Korrekt ist er wenn eine Magic–Number vorhanden ist, alle restlichen Felder erlaubte Werte haben und die Integrität des Headers bis Byte 20 durch die darauffolgende MAC überprüft werden konnte. Konnte die Integrität nicht überprüft werden, wurden entweder Daten im Header verändert oder ein falscher Schlüssel wurde übergeben.

Jeder zu lesende Block wird im Anschluss komplett in einen Puffer gelesen. Die Nonce wird ausgelesen und dem Entschlüsselungsalgorithmus als Eingabge neben dem eigentlichen Datenblock und dem Schlüssel mitgegeben. Dieser überprüft ob die Integrität des Datenblocks korrekt ist und entschlüsselt diesen im Erfolgsfall. Anhand der Position im Datenstrom wird zudem überprüft ob die Blocknummer zum Wert in der Nonce passt. Stimmen diese nicht überein, wird die Entschlüsselung verweigert, da ein Angreifer möglicherweise die Reihenfolge der Blöcke hätte vertauschen können.

Wahlfreier Zugriff: Wurde der Header bereits gelesen, so kann ein beliebiger Block im Datenstrom gelesen werden, sofern der unterliegende Datenstrom wahlfreien Zugriff (also die Anwendung von

Seek()) erlaubt. Die Anfangsposition des zu lesenden Blocks kann mit Gleichung (5.2) berechnet werden, wobei o der Offset im unverschlüsselten Datenstrom ist.

$$f_{\text{offset}}(o) = 36 + \left\lceil \frac{o}{b} \right\rceil \times (8 + 16 + b) \tag{5.2}$$

Der Block an dieser Stelle muss komplett gelesen und entschlüsselt werden, auch wenn nur wenige Bytes innerhalb des Blocks angefragt werden. Da typischerweise die Blöcke aber fortlaufend gelesen werden, ist das aus Sicht des Autors ein vernachlässigbares Problem.

Die einzelnen Blocks des vorgestellten Formats ähneln der *Secretbox* der freien NaCL–Bibliothek[18]. Diese erlaubt allerdings keinen wahlfreien Zugriff. Abgesehen handelt es sich um eine Neuentwicklung, die auch außerhalb von `brig` eingesetzt werden kann.

5.4.1.2. Kompression

Bevor Datenströme verschlüsselt werden, werden diese von `brig` auch komprimiert[19]. Auch hier wurde ein eigenes Containerformat entworfen, welches in Abb. 5.13 gezeigt wird.

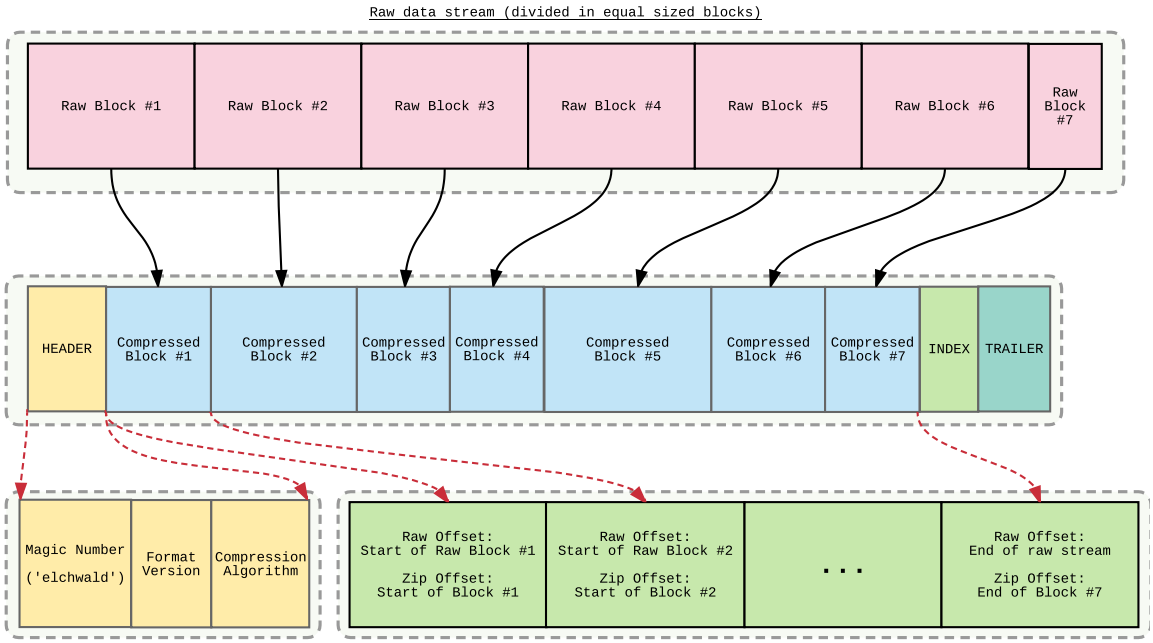


Abbildung 5.13.: Aufbau des Kompressions–Dateiformats.

Nötig war dieser Schritt auch hier wieder, weil kein geeignetes Format gefunden werden konnte, welches wahlfreien Zugriff im komprimierten Datenstrom zulässt, ohne dass dabei die ganze Datei entpackt werden muss.

Enkodierung: Der Eingabedatenstrom wird in gleich große Blöcke unterteilt (standardmäßig maxi-

[18] Siehe auch: https://nacl.cr.yp.to/secretbox.html

[19] Rein technisch ist es auch andersherum möglich, aber aufgrund der prinzipbedingten, hohen Entropie von verschlüsselten Texten wären in dieser Reihenfolge die Kompressionsraten sehr gering.

mal 64KB), wobei nur der letzte Block kleiner sein darf. Nachdem der Header geschrieben wurde, folgt jeder Eingabeblock als komprimierter Block mit variabler Länge. Am Schluss wird ein Index geschrieben, der beschreibt welcher Eingabeblock mit welchem komprimierten Block korrespondiert. Der Index kann nur am Ende geschrieben werden, da die genauen Offsets innerhalb dieses Indexes erst nach dem Komprimieren bekannt sind. Für eine effiziente Nutzung dieses Formats ist es also nötig, dass der Datenstrom einen effizienten, wahlfreien Zugriff am Ende der Datei bietet. Glücklicherweise unterstützt dies `ipfs`. Datenströme wie `stdin` unter Unix unterstützen allerdings keinen wahlfreien Zugriff, weshalb das vorgestellte Format für solche Anwendungsfälle eher ungeeignet ist.

Der Index besteht aus zwei Teilen: Aus dem eigentlichen Index und einem sogenannten »Trailer«, der die Größe des Indexes enthält. Zusätzlich enthält dieser Trailer noch die verwendete Blockgröße, in die der unkomprimierte Datenstrom unterteilt wurde. Der eigentliche Index besteht aus einer Liste von 64–Bit Offset–Paaren. Jedes Paar enthält einmal den unkomprimierten und einmal den komprimierten Offset eines Blocks als Absolutwert gemessen vom Anfang des Datenstroms. Am Ende wird ein zusätzliches Paar eingefügt, welches zu keinem realen Block verweist. Dieses letzte Paar beschreibt die Größe des unkomprimierten und komprimierten Datenstroms.

Der vorangestellte Header enthält alle Daten, die definitiv vor der Kompression des ersten Blockes vorhanden sind:

- Eine *Magic–Number* (8 Byte, ASCII Repräsentation von `elchwald`). Wie beim Verschlüsselungsformat dient diese zur schnellen Erkennung dieses Formats.
- Eine *Formatversion* (2 Byte, momentan »`0x01`«). Kann analog zum Verschlüsselungsformat bei Änderungen inkrementiert werden.
- Der verwendete *Algorithmustyp* (2 Byte, standardmäßig *Snappy*). Folgende Algorithmen werden momentan unterstützt:
  - Snappy[21] (sehr schneller Algorithmus mit akzeptabler Kompressionsrate)
  - LZ4[22] (etwas langsamer, aber deutlich höhere Kompressionsrate)
  - None (gar keine Kompression, Index wird trotzdem geschrieben)

  Weitere Algorithmen wie *Brotli*[23] können problemlos hinzugefügt werden, allerdings gab es zu diesen Zeitpunkten noch keine vernünftig nutzbaren Bibliotheken.

Dekodierung: Bevor der erste Block dekodiert werden kann muss sowohl der Header als auch der Index geladen werden. Dazu müssen die ersten 12 Bytes des Datenstroms gelesen werden. Im Anschluss muss fast an das Ende (Ende minus 12 Byte) des Datenstroms gesprungen werden, um dort den Trailer zu lesen. Mit der darin enthaltenen Größe des Indexes kann die Anfangsposition des Indexes bestimmt werden (Ende minus 12 Byte minus Indexgröße). Alle Offset–Paare im Index werden in eine sortierte Liste geladen. Die Blockgröße eines komprimierten/unkomprimierten Blocks an der Stelle ergibt sich dabei aus der Differenz des Offset–Paars an der Stelle $n + 1$ und seines Vorgängers an der Stelle n. Mithilfe der Blockgröße kann ein entsprechend dimensioniertes Stück vom komprimierten Datenstrom gelesen und dekomprimiert werden.

Wahlfreier Zugriff: Um auf einen beliebigen Offset o im unkomprimierten Datenstrom zuzugreifen, muss dieser zunächst in den komprimierten Offset übersetzt werden. Dazu muss mittels binärer Suche

im Index der passende Anfang des unkomprimierten Blocks gefunden werden. Wurde der passende Block bestimmt, ist auch der Anfangsoffset im komprimierten Datenstrom bekannt. Dadurch kann der entsprechende Block ganz geladen und dekomprimiert werden. Innerhalb der dekomprimierten Daten kann dann vom Anfangsoffset a noch zum Zieloffset $o - a$ gesprungen werden.

5.4.2. Transfer–Layer

Damit Metadaten ausgetauscht werden können, ist ein sicherer Steuerkanal nötig, der unabhängig vom Datenkanal ist, über den die eigentlichen Daten ausgetauscht werden. Über diesen muss ein *Remote-Procedure-Call* (RPC[20]) ähnliches Protokoll implementiert werden, damit ein Teilnehmer Anfragen an einen anderen stellen kann.

Die Basis dieses sicheren Steuerkanals wird von `ipfs` gestellt. Dabei wird kein zusätzlicher Netzwerkport für den RPC–Dienst in Anspruch genommen, da alle Kommunikation über den selben Kanal laufen, wie die eigentliche Datenübertragung. Es findet also eine Art »Multiplexing« statt.

Dies wird durch das fortgeschrittene Netzwerkmodell von `ipfs` möglich[21], welches in Abb. 5.14 gezeigt wird. Nutzer des gezeigten Netzwerkstacks können eigene Protokolle registrieren, die mittels eines *Muxing-Protokolls* namens *Multistream*[22] in einer einzigen, gemeinsamen physikalischen Verbindung zusammengefasst werden. Der sogenannte *Swarm* hält eine Verbindung zu allen zu ihm verbundenen Peers und macht es so möglich jeden Netzwerkpartner von der Protokollebene aus über seine Peer–ID anzusprechen. Der eigentliche Verbindungsaufbau geschieht dann, wie in Abschnitt 4.1.1 beschrieben auch über NAT–Grenzen hinweg.

Im Falle von `brig` wird ein eigenes Protokoll registriert, um mit anderen Teilnehmern zu kommunizieren. Dieses ist ähnlich aufgebaut wie das Protokoll zwischen Daemon und Client (siehe Abschnitt 5.3.2), unterstützt aber andere Anfragen und hat erhöhte Sicherheitsanforderungen. Eine genauere Beschreibung des Protokolls wird in [1] gegeben, hier werden nur kurz die wichtigsten Eigenschaften genannt:

- Authentifizierung mittels Remote–Liste bei jedem Verbindungsaufbau.
- Anfragen und Antworten werden als Protobuf–Nachricht enkodiert. Die eigentliche Protokolldefinition kann in Anhang B.2 eingesehen werden.
- Kompression der gesendeten Nachrichten mittels Snappy.
- Zusätzliche Verschlüsselung der Verbindung, mittels eines via *Elliptic Curve Diffie Hellman* ausgetauschten Schlüssels.
- Senden von »Broadcast«–Nachrichten zu allen bekannten, verbundenen Teilnehmern. Es wird keine Antwort auf Broadcast–Nachrichten erwartet. Diese sind daher eher für Status–Updates geeignet.

Im momentanen Zustand wird nur eine einzige Anfrage unterstützt. Dies ist die in Abschnitt 5.2.4 beschriebene `FETCH`–Anfrage. Zukünftig ist die Einführung weiterer Anfragen geplant. Um beispielsweise Echtzeit–Synchronisation zu unterstützen, müssten zwei weitere Nachrichten eingeführt

[20] Siehe auch: https://de.wikipedia.org/wiki/Remote_Procedure_Call
[21] Implementiert als eigene Bibliothek »libp2p«: https://github.com/libp2p/go-libp2p
[22] Siehe auch: https://github.com/multiformats/multistream
[23] Diese Grafik ist eine Aufbereitung von: https://github.com/libp2p/go-libp2p/tree/master/p2p/net

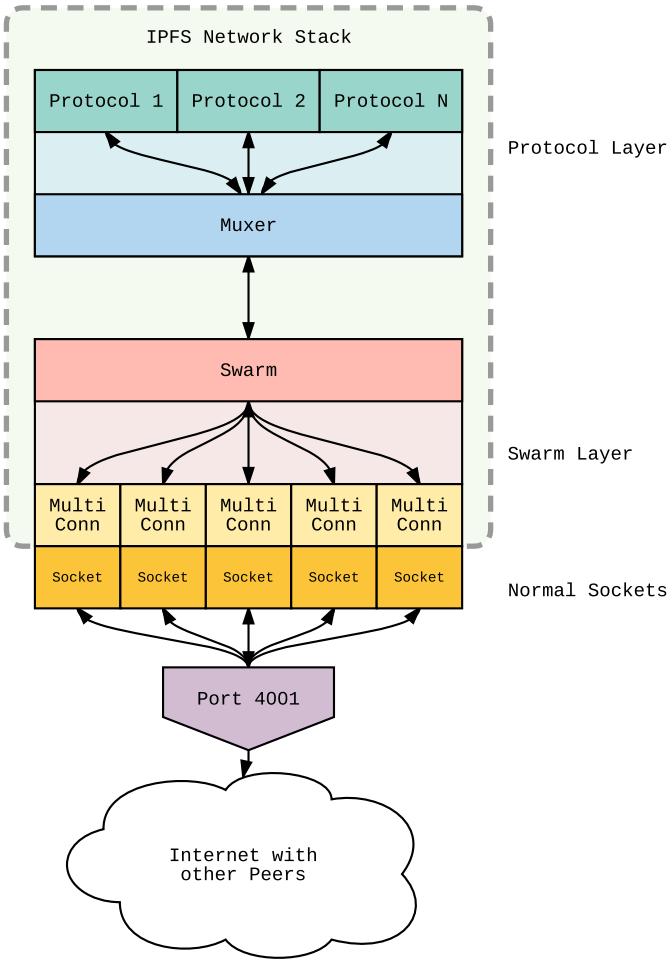


Abbildung 5.14.: Der Netzwerkstack von `ipfs` im Detail[23].

werden:

- `UPDATE`: Eine Nachricht die aktiv an alle Teilnehmer in der Remote–Liste geschickt wird. Sie enthält einen einzelnen Checkpoint. Die darin beschriebene atomare Änderung sollte dann auf Empfängerseite direkt in den Staging–Bereich eingegliedert werden.
- `DIFF <COMMIT_HASH>`: Wie `FETCH`, gibt aber nur die Änderungen seit dem angegebenen `COMMIT_HASH` zurück.

5.4.3. Benutzermanagement

In den Anforderungen in Kapitel 3 wird eine menschenlesbare Identität gefordert, mit der Kommunikationspartner einfach erkennbar sind. Der von `ipfs` verwendete Identitätsbezeichner ist allerdings eine für Menschen schwer zu merkende Prüfsumme (die »Peer–ID«).

Es wurden in dieser Arbeit bereits einige Identifikationsbezeichner beispielhaft verwendet. Diese entsprechen einer abgeschwächten Form der Jabber–ID[24] (*JID*, vgl. auch [24], S. 14). Diese hat, ähnlich wie eine E–Mail Adresse, die Form `user@domain/resource`. Beim Jabber/XMPP Protokoll ist der Teil hinter dem »/« optional, der Rest ist zwingend erforderlich. Als Abschwächung ist bei `brig` auch der Teil hinter dem »@« optional. Darüber hinaus sollen folgende Regeln gelten:

- Es sind keine Leerzeichen erlaubt.
- Ein leerer String ist nicht valide.
- Groß- und Kleinschreibung wird nicht unterschieden. Es wird empfohlen den Namen klein zu schreiben, so wie es bei Mailadressen und URLs der Fall ist.
- Der String muss valides UTF8[25] sein.
- Der String muss der »UTF-8 Shortest Form[26]« entsprechen
- Der String darf durch die »UTF-8 NKFC Normalisierung[27]« nicht verändert werden.
- Alle Charaktere müssen druckbar und auf dem Bildschirm darstellbar sein.

Insbesondere die letzten vier Punkte dienen der Sicherheit, da ein Angreifer versuchen könnte eine Unicode–Sequenz zu generieren, welche visuell genauso aussieht wie die eines anderen Nutzers, aber einer anderen Byte–Reihenfolge und somit einer anderen Identität entspricht.

Valide Identitätsbezeichner wären also beispielsweise:

- `alice`
- `alice@company`
- `alice@company.de`
- `alice@company.de/laptop`
- `böb@subdomain.company.de/desktop`

Die Wahl der JID als Basis hat einige Vorteile:

- Eine E–Mail Adresse oder eine JID ist gleichzeitig ein valider Identitätsbezeichner.

[24] Mehr Details unter: https://de.wikipedia.org/wiki/Jabber_Identifier
[25] Siehe auch: https://de.wikipedia.org/wiki/UTF-8
[26] Siehe auch: http://unicode.org/versions/corrigendum1.html
[27] Siehe auch: http://www.unicode.org/reports/tr15//#Norm_Forms

- Der Nutzer kann eine fast beliebige Unicode Sequenz als Name verwenden, was beispielsweise für Nutzer des kyrillischen Alphabetes nützlich ist.
- Unternehmen können die Identifikationsbezeichner hierarchisch gliedern. So kann *Alice* der Bezeichner `alice@security.google.com` zugewiesen werden, wenn sie im Sicherheitsteam arbeitet.
- Der *Ressourcen*–Teil hinter dem »/« ermöglicht die Nutzung desselben Nutzernamens auf verschiedenen Geräten, wie beispielsweise `desktop` oder `laptop`.

Eine Nutzung des `domain` und `resource`–Teils ist kein Zwang, wird aber als Konvention empfohlen, da es eine Unterteilung in Gruppen und Geräte ermöglicht.

Um den Identifikationsbezeichner im Netzwerk auffindbar zu machen, wendet `brig` einen »Trick« an. Jeder `brig`–Knoten veröffentlicht einen einzelnen `blob` in das `ipfs`–Netzwerk mit dem Inhalt `brig#user:<username>`. Dieses Verfahren wird *Publishing* genannt. Ein Nutzer, der nun einen solchen menschenlesbaren Namen zu einer Netzwerkadresse auflösen möchte, kann den Inhalt des obigen Datensatzes generieren und daraus eine Prüfsumme bilden. Mit der entstandenen Prüfsumme kann wie in Listing 5.4.3 mittels dem folgenden Verfahren[28] herausgefunden werden, welche Knoten diesen Datensatz anbieten:

```
$ USER_HASH=$(printf 'brig#user:%s' alice  | multihash -)
$ echo $USER_HASH
QmdNdLHqc1ryoCU5LPEMMCrxkLSafgKuHzpVZ5DFdzZ61M
# Schlage Hash in der Distributed Hash Table nach:
$ ipfs dht findprovs $USER_HASH
<PEER_ID_OF_POSSIBLE_ALICE_1>
<PEER_ID_OF_POSSIBLE_ALICE_2>
...
```

Da prinzipiell jeder Knoten sich als *Alice* ausgeben kann, wird aus den möglichen Peers, derjenige ausgewählt, dessen `ipfs`–Identitätsbezeichner (bei `brig` wird dieser als *Fingerprint* bezeichnet) als vertrauenswürdig eingestuft wurde. Vertrauenswürdig ist er, wenn der Fingerprint in der Remote–Liste in der Kombination von Nutzernamen und Fingerprint auftaucht. In diesem Fall muss der Nutzer explizit authentifiziert worden sein. Abb. 5.15 zeigt dieses Verfahren noch einmal graphisch.

Analog kann das Konzept auch übertragen werden, um bestimmte Gruppen von Nutzern zu finden. Angenommen, Alice, Bob und Charlie arbeiten im gleichen Unternehmen. Das Unternehmen spiegelt sich auch in ihren Identitätsbezeichnern wieder:

- `alice@corp.de/server`
- `bob@corp.de/laptop`
- `charlie@corp.de/desktop`

Neben den gesamten Nutzernamen, können diese drei Nutzer auch ihren Unternehmensnamen (`corp.de`) *publishen*, beziehungsweise auch ihren Nutzernamen ohne den `resource`–Zusatz. So ist es beispielsweise wie in Listing 5.4.3 möglich die öffentlichen Identitäten alle Unternehmensmitglieder aufzulösen:

[28] Benötigt das `multihash` Werkzeug: https://github.com/multiformats/go-multihash/tree/master/multihash

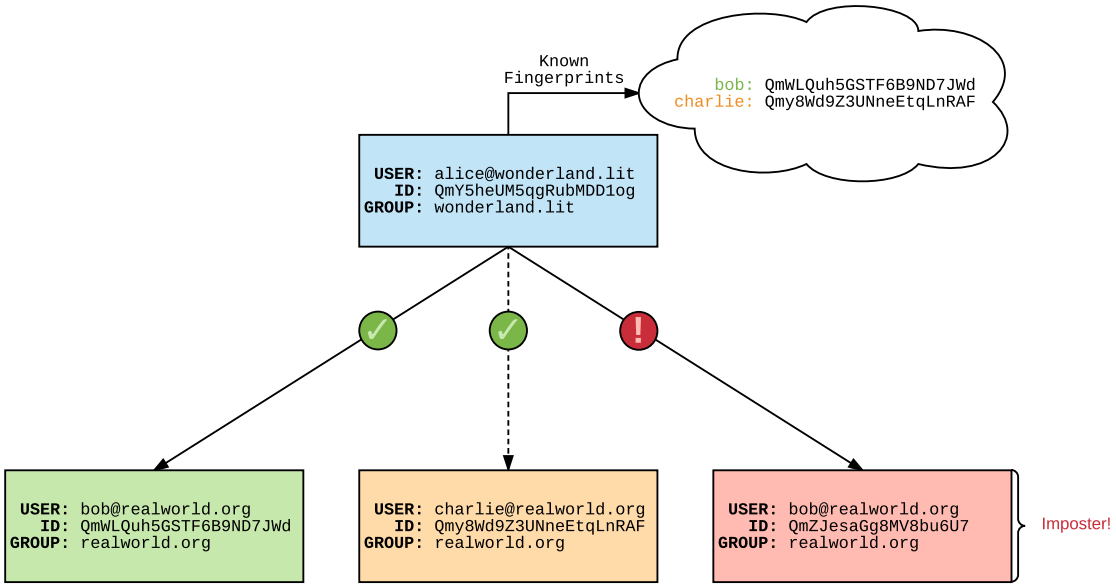


Abbildung 5.15.: Überprüfung eines Benutzernamens mittels Peer–ID

```
$ CORP_HASH=$(printf 'brig#domain:%s' corp.de | multihash -)
$ ipfs dht findprovs $CORP_HASH
<PEER_ID_OF_POSSIBLE_CORP_MEMBER_1>
<PEER_ID_OF_POSSIBLE_CORP_MEMBER_2>
...
```

Die einzelnen IDs können dann, sofern bekannt, zu den »Klarnamen« aufgelöst werden, die in der Remote–Liste jedes Teilnehmers stehen. Insgesamt können folgende sinnvolle Kombinationen (falls möglich, da optional) von `brig` *published* werden, die jeweils eine spezielle Semantik hätten:

- `user`: Finden des Nutzernamens alleine.
- domain: Finden des Gruppennamen.
- `user@domain`: Alle Geräte eines Nutzers.
- `user@domain/resource`: Spezifisches Gerät eines Nutzers.

Das Besondere an dieser Vorgehensweise ist, dass kein Nutzer sich an einer zentralen Stelle registriert. Trotzdem können sich die Nutzer gegenseitig im Netzwerk mit einem aussagekräftigen Namen finden und trauen nicht einer zentralen Instanz, sondern entscheiden selbst welchen Knoten sie trauen. Diese Eigenschaften entsprechen den drei Ecken von *Zooko's Dreieck*[25], von denen gesagt wird, dass immer nur zwei Ecken gleichzeitig erfüllbar sind (siehe Abb. 5.16). Allerdings ist die oben gezeigte Technik als Alternative für Techniken wie *DNS* kaum einsetzbar und ist daher keine allgemeine Lösung für *Zooko's Dilemma*.

Aus Sicht der Usability ist dabei die initiale Authentifizierung ein Problem. Diese kann nicht von `brig` automatisiert erledigt werden, da `brig` nicht wissen kann welche Prüfsumme die »richtige« ist. Es wird im aktuellen Entwurf vom Nutzer erwartet, dass er über einen sicheren Seitenkanal

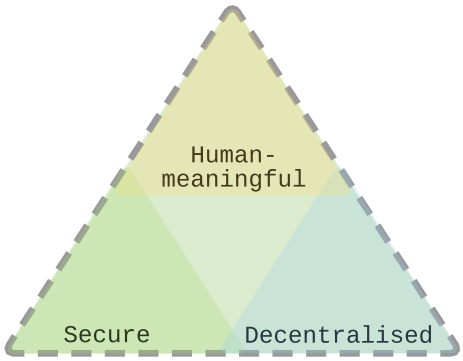


Abbildung 5.16.: Bildliche Darstellung von Zooko's Dreieck.

(beispielsweise durch ein persönliches Treffen) die angepriesene Prüfsumme überprüft.

Die oben vorgestellte Idee kann aber auch in Richtung eines *Web of Trust*[29] erweitert werden. Als Anwendungsfall könnte man eine geschlossene Gruppe von Nutzern betrachten, die sich nur teilweise bekannt sind. Vergrößert sich die Gruppe mit einem neuen Teilnehmer, so muss dieser alle anderen Teilnehmer authentifizieren und gegenseitig auch von diesen authentifiziert werden. Ab einer bestimmten Gruppengröße wird dies eine sehr aufwendige Aufgabe. Eine logische Lösung wäre das Anlegen eines *Blessed Repository*, dem alle Gruppenteilnehmer trauen und das von einem respektierten Teilnehmer der Gruppe betrieben wird. Möchte man diesen zentralen Ansatz nicht, so kann man wie beim *Web of Trust*, ein System einführen, das einem neuen Nutzer automatisch traut, wenn eine ausreichende Anzahl anderer Gruppenteilnehmer dem Neuling vertraut hat.

Daneben sind noch weitere Strategien denkbar, wie das automatische Akzeptieren neuer Teilnehmer (anwendbar, wenn beispielsweise ein Dozent Vorlesungsmaterial verteilen will), oder ein Frage–Antwort–Verfahren wie bei *Off-The-Record-Messaging (OTR)*. Dabei stellen sich beide Teilnehmer eine Frage, die sie jeweils korrekt beantworten müssen. Weitere Konzepte zur Authentifizierung werden in [1] beschrieben.

[29] https://de.wikipedia.org/wiki/Web_of_Trust

6 Implementierung

Dieses Kapitel dokumentiert die Implementierung. Der praktische Status der Implementierung kann in Anhang A betrachtet werden. Dort werden nur Funktionen gezeigt, die auch tatsächlich schon existieren. An dieser Stelle werden eher Implementierungsdetails gezeigt, die einen Einstieg in die technische Umsetzung von `brig` geben sollen.

6.1. Wahl der Sprache

Als Sprache zur Implementierung wurde die relativ junge Programmiersprache *Go* ausgewählt. *Go* ist eine an *C* angelehnte Sprache, die von Ken Thompson, Rob Pike und Robert Griesemer initiiert wurde und mittlerweile von Google getragen und weiterentwickelt wird (siehe auch [26], S. XI ff.). Für dieses spezielle Projekt bietet die Sprache aus Sicht des Autors folgende Vorteile:

Garbage–Collector: Erleichtert die Entwicklung lang laufender Dienste und erleichtert den Programmierer die Arbeit durch den Wegfall der manuellen Speicherallokation und Bereinigung.

Hohe Grundperformanz: Zwar erreicht diese nicht die Performanz von C, liegt aber zumindest in der selben Größenordnung (vgl. [27], S. 37).

Weitläufige Standardbibliothek: Es sind wenig externe Bibliotheken nötig. Insbesondere für die Entwicklung von Netzwerk- und Systemdiensten gibt es eine breite Auswahl von gut durchdachten Bibliotheken. Besonders Erwähnenswert ist das umfangreiche Angebot an gut dokumentierten kryptografischen Primitiven, die eine unsichere Benutzung möglichst ausschließen sollen[1].

Schneller Kompiliervorgang: Selbst große Anwendungen werden in wenigen Sekunden in eine statisch gelinkte Binärdatei ohne Abhängigkeiten übersetzt. Kleinere bis mittlere Anwendungen können ähnlich wie bei einer Skriptsprache direkt mittels des `go run` Befehls ausgeführt werden.

Cross–Kompilierung: Anwendungen können für viele verschiedene Systeme von einem Entwicklungsrechner aus gebaut werden. Da die entstehende Binärdatei statisch gelinkt ist, werden zudem keine weiteren Abhängigkeiten benötigt. Dadurch ist es möglich für verschiedene Systeme bereits gebaute Binärdateien anzubieten.

Eingebauter Scheduler: Parallele und nebenläufige Anwendungen wie Netzwerkserver sind sehr einfach zu entwickeln ohne für jede Aufgabe einen dedizierten Thread starten zu müssen. Stattdessen wechseln sich viele Koroutinen[28] (*Go-Routinen* genannt) auf einer typischerweise geringeren Anzahl von Threads ab. Dadurch entfällt die Implementierung eines expliziten Mainloops und das Starten von Threads per Hand.

Hohe Portabilität: Die meisten Programme lassen sich ohne Anpassung auf den gängigsten Desktop–Betriebssystemen kompilieren. Die Möglichkeit native Anwendungen für Android und iOS zu entwi-

[1] So wurde beispielsweise der unsichere ECB–Betriebsmodus für Blockchiffren absichtlich weggelassen: https://github.com/golang/go/issues/5597

ckeln ist ebenfalls in der Entwicklung[2].

Große Anzahl mitgelieferter Werkzeuge: Im Gegensatz zu anderen Sprachen umfasst das *Go*–Paket nicht nur die Sprache, sondern auch ein Buildsystem, ein Race–Condition–Checker, ein Testrunner, ein Dokumentationsgenerator, ein Static–Code–Checker, eine Formatierungshilfe und eine Art Paketmanager.

Einfache Installation und rapides Prototyping: Durch das `go get`–Werkzeug, ist es möglich direkt Bibliotheken und Anwendungen von Plattformen wie *GitHub* zu installieren. Gleichzeitig ist es einfach eigene Bibliotheken und Anwendungen einzustellen.

Einheitliche Formatierung: Durch das »`go fmt`« Werkzeug und strikte Stilrichtlinien[3] sieht jeder *Go*–Quelltext ähnlich und damit vertraut aus. Dies erleichtert externen Entwicklern den Einstieg.

Geringe Sprachkomplexität: Die Sprache verzichtet bewusst auf Konstrukte, die die Implementierung des Compilers verlangsamen würden oder das Verständnis des damit produzierten Quelltextes erschweren würde. Daher ist *Go* eine Sprache, die verglichen mit *Python* zwar relativ wiederholend und gesprächig ist, aber dadurch gleichzeitig auch sehr einfach zu lesen ist.

Auch *Go* ist keine perfekte Sprache. Daher werden nachfolgend einige kleinere Nachteile und deren Lösungen im Kontext von `brig` aufgezählt:

Schwergewichtige Binärdateien: Da bei *Go* alles statisch gelinkt wird, ist die entstehende Binärdatei relativ groß. Im Falle des `brig`–Prototypen sind das momentan etwa 35 Megabyte. Werkzeuge wie upx[4] können dies auf rund 8 Megabyte reduzieren, ohne dass der Anwender die Binärdatei selbst entpacken muss.

Vendor: Der »Paketmanager« von go beherrscht nicht die Installation einer bestimmten Paketversion. Stattdessen wird einfach immer die momentan aktuelle Version installiert. Viele Projekte, `brig` eingeschlossen, brauchen und bevorzugen aber einen definierten Versionsstand, der von den Entwicklern getestet werden konnte. Dienste wie *gopkg.in*[5] versuchen eine zusätzliche Versionierung anzubieten, der aktuelle »Standard« ist die Nutzung des `vendor` Verzeichnisses. Diese Lösung läuft darauf hinaus, alle benötigten Abhängigkeiten in der gewünschten Version in das eigene Quelltext–Repository zu kopieren. Diese unelegante, aber gut funktionierende Lösung wird von `brig` verwendet[6].

6.2. Status der Implementierung

Die momentane Implementierung setzt die vorher besprochene Architektur größtenteils um. Der Code der zuständig für die Synchronisierung ist funktioniert zwar, ist jedoch noch nicht so detailliert wie in der Architektur ausgearbeitet. Insbesondere beherrscht er noch nicht die Synchronisation leerer Verzeichnisse und kann kompatibel Änderungen nur sehr bedingt auflösen. Ansonsten unterscheidet sich die tatsächliche Implementierung und die theoretische Architektur nur in Details.

[2] Siehe dazu: https://golang.org/wiki/Mobile
[3] Siehe auch: https://blog.golang.org/go-fmt-your-code
[4] Ein Packprogramm für Binärdateien. Mehr Informationen unter http://upx.sourceforge.net
[5] http://labix.org/gopkg.in
[6] Eigenes Repository für verwendete Bibliotheken: https://github.com/disorganizer/brig-vendor

6.2.1. Umfang

Die Statistik in Tabelle 6.1 wurde mit dem freien Werkzeug `cloc`[7] erstellt. Autogenerierte Dateien wurden dabei nicht mit eingerechnet, Testdateien hingegen schon. Auch fehlen in der Statistik die Module aus Abschnitt 6.5, die zwar geschrieben worden sind, aber aufgrund sich ändernder Designanforderungen wieder gelöscht worden sind. Es wurde versucht, die Quelltextbasis möglichst klein zu halten.

Tabelle 6.1.: Quelltextumfang, gestaffelt nach Sprache.

Sprache	Dateianzahl	Leerzeilen	Kommentare	Codezeilen
Go	86	2944	1700	11427
Go Tests	28	667	335	2890
Protocol Buffers	4	95	60	316
Bourne Shell	5	11	8	134
make	4	6	1	34
$\sum$	*127*	*3743*	*2104*	*14801*

6.2.2. Dokumentation

Die Implementierung ist nicht auf ein Paradigma festgelegt. Zwar wird wie bei vielen Projekten hauptsächlich auf *objektorientierte Programmierung (OOP)* gesetzt, doch erlaubt *Go* auch die Anwendung prozeduraler und funktionaler Programmiertechniken. Aus diesem Grund macht eine Beschreibung der Implementierung als UML für `brig` wenig Sinn, da einige Konzepte von dieser Beschreibungssprache nicht ausreichend gut abgebildet werden können. Zudem würde eine Beschreibung aller implementierten Typen schlicht den Rahmen dieser Arbeit sprengen.

Einen guten Überblick über die Implementierung und aller benutzten Typen erlaubt die API–Dokumentation, die unter *»godoc.org«*[8] einsehbar ist. Die Software ist möglichst nahe an der Beschreibung von *Effective Go* gehalten[9], was den Einstieg für andere *Go*–Programmierer erleichtern sollte. Eines der meist genutzten Idiome bildet dabei die strikte Fehlerbehandlung, bei der jede Funktion, die einen Fehler zurückgeben kann, einen zweiten `error`–Wert zurückgibt. Dieser wird innerhalb der Funktion möglichst früh zurückgeben. So entstehen zwei »vertikale Linien« im optischen Aussehen des Quelltextes. Die eine Linie kümmert sich um die Fehlerbehandlung, die andere um den Erfolgsfall. Listing 6.2.2 zeigt ein Beispiel für diese Regel:

```
func someAction(msg string) (int, error) {
    //  | Fehlerbehandlung ist eingerückt.
    if len(msg) < 10 {
        return -1, ErrTooShort
    }// |
     // |
```

[7] https://github.com/AlDanial/cloc
[8] https://godoc.org/github.com/disorganizer/brig
[9] https://golang.org/doc/effective_go.html

```
    if len(msg) > 20 {
        return -1, ErrTooLong
    }// |
     // |
    // Erfolgsfall ist nicht eingerückt.
    return len(msg) * len(msg), nil
}
```

6.2.3. Paketübersicht

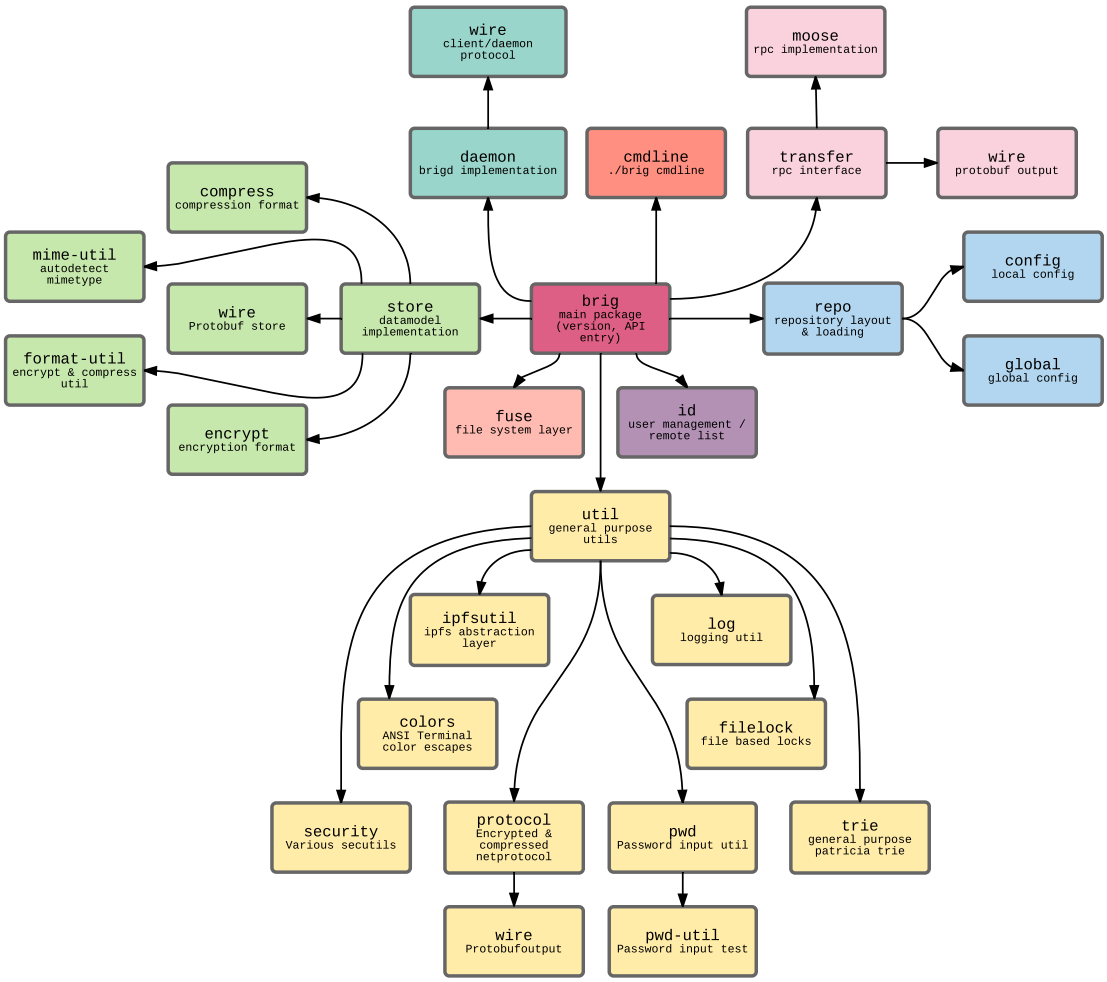


Abbildung 6.1.: Übersicht über alle Pakete in `brig`.

Abb. 6.1 zeigt die Aufteilung des Quelltextes in die einzelnen Go–Pakete. Die Software ist dabei in fünf Hauptpakete und drei »umliegende« Pakete aufgeteilt. Die Hauptpakete sind dabei:

- `repo`: Implementiert das Anlegen, Laden und Schreiben einer Repository–Struktur samt Konfiguration. Dient zudem als oberster Eintrittspunkt in die API von `brig`, da hier alle wichtigen Instanzen (`ipfs`–Layer, *Stores* etc.) vereint sind.
- `store`: Implementiert das eigentliche Datenmodell und alle Operationen darauf.

- `daemon`: Implementiert die Netzwerkschnittstelle zwischen `brigctl` und `brigd`.
- `id`: Implementiert das Benutzermanagement, Identitätsvalidierung und die Remote–Liste.
- `transfer`: Implementiert den RPC–Mechanismus zwischen zwei `brig`–Knoten und alle darin aufrufbaren Methoden.

Die umliegenden Pakete bestehen aus:

- `cmdline`: Implementiert die Kommandozeile.
- `fuse`: Implementiert die Dateisystemschicht.
- `util`: Implementiert allgemein nützliche Funktionen für alle Pakete.

6.3. Ausgewählte Themen

Aufgrund des großen Umfangs der Implementierung würde eine detaillierte Beschreibung derselben den Rahmen dieser Arbeit sprengen. Stattdessen werden hier einige ausgewählte Stellen der Implementierung näher beleuchtet. Besonderer Wert wird dabei auf Details gelegt, die in der Besprechung der Architektur noch nicht vorkamen.

6.3.1. Aufbau des *Store*

Der *Store* kapselt alle Knoten des MDAG und implementiert die Operationen auf diesen Knoten. Die besondere Schwierigkeit bei der Implementierung ist dabei, dass jede Modifikation des Stores serialisiert werden muss, damit sie nach einem Neustart oder Absturz von `brigd` noch weiterhin vorhanden ist. Für diesen Zweck setzt `brig` eine eingebettete[10] Key–Value–Datenbank namens *BoltDB*[11] ein.

BoltDB verfolgt ein sehr minimales Konzept, indem keine besonderen Datentypen unterstützt werden. Alle Schlüssel und Werte sind ausschließlich Binärdaten, um deren Serialisierung sich der Programmierer zu kümmern hat. Im Falle von `brig` wird jeder Knoten als Protobuf–Nachricht in der Datenbank gespeichert und wieder ausgelesen. Die Verschachtlung von Daten wird durch sogenannte *Buckets* (dt. Eimer) unterstützt. Jeder Schlüssel kann einen Bucket enthalten, der wiederum weitere Buckets und normale Schlüsselwertpaare enthalten kann. Dadurch ist die Bildung einer Hierarchie möglich.

Für jeden Knotentypen (File, Directory, Commit) und Metadatentypen (Checkpoint, Ref) wurde eine eigene, gleichnamige Go–Struktur eingeführt, welche die Daten in der Datenbank kapselt und alle Operationen darauf implementiert. Jede dieser Knotenstrukturen implementiert dabei ein gemeinsames Interface namens `Node`. Jedes `Node` weiß wie ein Knoten serialisiert und deserialisiert wird. Zudem fasst das Interface Metadaten zusammen, die für alle Knotentypen gleich sind (also Prüfsumme, Elternpfad, eigener Name, Größe, Änderungszeitpunkt und UID). Ein Verzeichnis speichert dabei allerdings nicht seine Kindknoten (siehe Listing 6.3.1) als weitere Verzeichnisstrukturen, sondern verweist auf diese indirekt über deren Prüfsumme:

```
type Directory {
    // Keine direkten Links: children []*Directory
    // Stattdessen Referenzierung über Prüfsumme:
```

[10] Also eine Datenbank, die ohne eigenen Datenbankserver funktioniert.
[11] MIT–Lizensiert; Webseite: https://github.com/boltdb/bolt

```
    children []*Hash

    // Nutze 'fs' als Auflöser für diese Prüfsummen.
    fs *FS
}
```

Würde man die Kindknoten direkt laden, so müsste beim Laden des HEAD–Commit der *gesamte* Graph geladen werden, da HEAD wiederum auf ein Wurzelverzeichnis und sein Vorgänger–Commit verweist. Es muss nun allerdings eine zentrale Instanz geben, die einen Knoten basierend auf dessen Prüfsumme auflösen kann. Diese zentrale Instanz heißt bei `brig` *FS* (kurz für *Filesystem*, dt. Dateisystem), da ihre Funktionalität dem Kern eines Dateisystems ähnelt. Genau wie ein Dateisystem organisiert *FS* den Inhalt der *BoltDB* und macht ihn über Pfade und Prüfsummen höheren Programmebenen zugreifbar. Um diese Aufgabe zu lösen, forciert *FS* eine hierarchische Ablagestruktur (gezeigt in Abb. 6.2) innerhalb der BoltDB, die an `git` angelehnt ist.

```
.
├── objects
│   └── <NODE_HASH>              ⇒ Metadaten eines Knoten
├── uid
│   └── <NODE_UID>               ⇒ Prüfsumme eines Knoten
├── tree
│   └── <FULL_NODE_PATH>         ⇒ Prüfsumme eines Knoten
├── checkpoints
│   └── <HEX_NODE_UID>
│       └── <IDX>                ⇒ Einzelner Checkpoint
├── stage
│   ├── objects
│   │   └── <NODE_HASH>          ⇒ Metadaten eines Staging-Knotens
│   ├── tree
│   │   └── <FULL_NODE_PATH>     ⇒ Prüfsumme eines Knoten
│   └── STATUS                   ⇒ Metadaten des Staging-Commits
├── stats
│   └── node-count               ⇒ Anzahl aller bisher erstellten Knoten
├── refs
│   └── <REFNAME>                ⇒ Prüfsumme eines Commits
└── metadata
    ├── id                       ⇒ Eigentümer (Nutzername) des Stores
    ├── hash                     ⇒ Eigentümer (Prüfsumme) des Stores
    └── version                  ⇒ Aktuelle Version
```

Abbildung 6.2.: Hierarchische Aufteilung in der BoltDB mittels Buckets.

Basierend auf dieser Struktur kann *FS* die folgenden Funktionen effizient implementieren:

func `NodeByHash(hash *Hash) (Node, error)`: Lädt die Metadaten eines Knoten anhand seiner Prüfsumme. Dabei wird zuerst in »`stage/objects/<NODE_HASH>`« nachgesehen und dann in »`objects/<NODE_HASH>`« falls der erste Schlüssel nicht existiert.

func `ResolveNode(nodePath string) (Node, error)`: Löst einen Pfad zu einem Node auf. Es wird probiert die Prüfsumme des Knotens »`stage/tree/<PFAD>`« beziehungsweise »`tree/<PFAD>`« nachzuschlagen. Falls das Nachschlagen erfolgreich war, wird `NodeByHash()` mit der so ermittelten Prüfsumme aufgerufen. Im Falle von Verzeichnissen wird dem Pfad vorher ein ».« angehängt. Das ist nötig, da sonst der Bucket zu dem Verzeichnisinhalten (»`/photos/`«) gefunden werden würde und nicht das Verzeichnis an sich (»`/photos/.`«). Letztere Idee stammt dabei aus dem normalen Unix–

Dateisystem, wo ein einzelner Punkt ebenfalls auf das aktuelle Verzeichnis zeigt. Es gibt allerdings noch kein Äquivalent zu »..«, welches auf das Elternverzeichnis zeigen würde.

func `StageNode(node Node) error`: Fügt dem Staging–Bereich einem Eintrag hinzu. Der Pfad des Knoten und seine Prüfsumme werden unter »`stage/...`« abgespeichert. Dabei sorgt die Funktion auch dafür, dass alle Elternverzeichnisse des Knotens im Staging–Bereich abgespeichert werden, da auch dessen Prüfsummen sich nach einer Modifikation von node verändert haben.

func `StageCheckpoint(ckp *Checkpoint) error`: Fügt einen Checkpoint dem Staging–Commit unter `stage/STATUS` hinzu und speichert ihn in »`checkpoints/<UID>/<LAST_IDX> + 1`« ab.

func `MakeCommit(author id.Peer, message string) error`: Kopiert das Wurzelverzeichnis des Staging–Commits und all seine Kinder in das Archiv (`objects/` und `tree/`). Der bisherige Staging–Commit wird der neue `HEAD` und ein neuer, leerer Staging–Commit wird angelegt, auf den `CURR` zeigt. Der Rest des Staging–Bereichs wird geleert.

func `ResolveRef(refname string) (Node, error)`: Schlägt die Prüfsumme unter »`refs/<refname>`« nach und übergibt diese `NodeByHash()`.

func `SaveRef(refname string, nd Node) error`: Setzt den Wert unter »`refs/<refname>`« auf die Prüfsumme des übergebenen Knoten.

func `History(uid uint64) (History, error)`: Lädt alle Checkpoints für ein bestimmtes Dokument anhand der UID des Dokuments. Für jeden neu angelegten Knoten wird eine neue UID generiert, indem die in »`stats/node-count`« abgespeicherte Ganzzahl um eins inkrementiert wird.

func `LookupNode(repoPath string) (Node, error)`: Löst das Wurzelverzeichnis des Staging–Commits auf und versucht mittels des übergebenen Pfades von dort auf das angeforderte Kind zu kommen, indem die Kinder eines Verzeichnisses mit `NodeByHash()` nachgeladen werden. Diese Funktion unterscheidet sich von `ResolveNode()` dadurch, dass gelöschte Pfade berücksichtigt werden. `ResolveNode()` gibt hingegen den letzten Stand eines Knoten zurück, der an dieser Stelle gespeichert war.

func `MetadataPut(key string, value []byte) error`: Erlaubt das Setzen bestimmter Schlüssel–Wert–Paare unterhalb des `metadata`–Bucket. Aufrufender Code kann dies nutzen, um spezielle Werte persistent zu hinterlegen.

func `MetadataGet(key string) ([]byte, error)`: Holt den Wert unter »`metadata/<key>`« aus der BoltDB.

Bei jeder Operation werden also die Daten direkt aus *BoltDB* geladen, deserialisiert und zu einer `Node`–Struktur umgewandelt. Als Effizienzsteigerung werden bereits aufgelöste Prüfsummen in ein assoziatives Array und bereits aufgelöste Pfade in einem Patricia–Trie[12] gespeichert. Sobald eine Reihe von Änderungen an einem im Speicher befindlichen `Node` gemacht wurde (beispielsweise eine Änderung der Prüfsumme nach einer Modifikation), wird der `Node`, und all seine Eltern (da dessen Prüfsummen sich ja auch geändert haben), in den Staging–Bereich eingefügt (durch Aufruf

[12] Auch Radix–Baum genannt. Speichert gemeinsame Präfixe nur einmal und eignet sich daher gut, um bei vielen Pfaden Speicher zu sparen. Siehe auch: https://de.wikipedia.org/wiki/Patricia-Trie

von `StageNode()`). Der »`stage/...`« Bereich fungiert also als persistentes Sammelbecken für alle Änderungen, während die Änderungen im Speicher den jeweils aktuellsten Stand wiederspiegeln.

Jede weitere Operation auf den Stores läuft auf eine Sequenz von Aufrufen der oben gezeigten Operationen hinaus. Beim Anzeigen aller Commits (`Log()`) wird beispielsweise die Referenz `HEAD` aufgelöst (mittels `ResolveRef()`). Dessen Eltern–Commit wird dann rekursiv aufgelöst (mittels `NodeByHash()`), bis kein weiterer Eltern–Commit gefunden werden konnte. Die so gefundenen Commits werden dann von `Log()` in einem Array zurückgegeben.

Beim Abspeichern in der Datenbank wird der sich im Speicher befindliche `Node` wieder in eine Protobuf–Nachricht übertragen (siehe Listing 6.3.1). Diese fasst für alle Knotentypen gemeinsame Attribute zusammen:

```
message Node {
  required NodeType type = 1;     // Knotentyp (Enumeration)
  required uint64 ID = 2;         // UID des Knoten.
  required uint64 node_size = 3;  // Größe in Byte.
  required bytes mod_time = 4;    // Letzte änderung als RFC 3339 Timestamp.
  required bytes hash = 5;        // Prüfsumme.
  required string basename = 6;   // Basename der Datei.
  required string dirname = 7;    // Verzeichnis in dem die Datei liegt.

  // Unternachrichten für die eigentlichen Knoten:
  optional File file = 8;
  optional Directory directory = 9;
  optional Commit commit = 10;
}
```

6.3.2. FUSE–Dateisystem

Filesystem in Userspace (kurz *FUSE*[13]) ist eine Technik, die es ermöglicht einen Ordner anzuzeigen, in dem von einem Programm (dem *Userspace–Treiber*) generierte Dateien angezeigt werden. Technisch basiert es darauf, dass ein Programm die spezielle Blockdatei `/dev/fuse` öffnet und mithilfe dieser mit dem Kernel kommuniziert. Die API, die dabei implementiert werden muss ist relativ umfangreich weswegen sich ein Wrapper anbietet, der eine »saubere« API für die jeweilige Sprache anbietet. Für *Go* gibt es mit `bazil/fuse`[14] eine sehr gute, entsprechende Bibliothek.

Die API basiert dabei auf Callbacks. Werden bestimmte Aktionen vom Nutzer getriggert (Beispiel: Er öffnet eine Datei), so wird im FUSE–Layer von `brig` eine entsprechende Funktion namens `Open()` aufgerufen. Dieser wird immer ein *Request-* sowie ein *Response–Objekt* mitgegeben. Die Aufgabe der aufgerufenen Funktion ist das Auslesen der Details aus dem *Request–Objekt* (Beispiel: In welchem Modus soll die Datei geöffnet werden?), das Ausführen einer Aktion (Beispiel: Öffne Datenstrom von `ipfs`) und das Befüllen des *Response–Objekts* (Beispiel: Kein Fehler, neuer Dateideskriptor wird zurückgegeben). Bei Fehlern kann verfrüht abgebrochen werden und ein spezieller Fehlercode wird

[13] Siehe auch: https://de.wikipedia.org/wiki/Filesystem_in_Userspace
[14] https://github.com/bazil/fuse

zurückgegeben (Beispiel: `fuse.EIO` für einen Input/Output–Fehler). Auf diese Weise können die meisten Systemaufrufe[15] (die normal vom Kernel vorgegeben sind) durch eigenen Code implementiert werden. Beispielsweise wird auch ein Callback aufgerufen, wenn das Kind eines Verzeichnisses nachgeschlagen werden muss.

Die meisten Operationen wie `mkdir()`, `create()`, `rename()` und `remove()` haben in der API des Store eine natürliche Entsprechung und sind entsprechend einfach zu implementieren. Schwieriger ist das Auslesen und vor allem die Modifikation von Dateiströmen. Die Sprache *Go* bietet mit dem Konzept von `io.Reader` und `io.Writer` ein sehr leicht wiederverwendbares Pattern, mit dem sich komplexe Dateistromverarbeitungen für den Nutzer transparent erledigen lassen, indem mehrere `io.Reader` ineinander verschachtelt werden. Abb. 6.3 zeigt auf der linken Seite alle nötigen `io.Reader`, um dem Nutzer des FUSE–Dateisystems den Inhalt einer Datei zu liefern.

FUSE fordert dabei Daten blockweise an. Das bedeutet, dass ein Funktionsaufruf einen Block nach einem bestimmten Offset in der Datei zurückliefert. Da der Nutzer des Dateisystems frei darin ist, beliebig oft `Seek(<offset>)` auf einen Dateideskriptor aufzurufen, kann die Reihenfolge dieser Leseoperationen beliebig sein. Hier erklärt sich auch der Grund warum das Verschlüsselungs- und Kompressionformat- auf effizienten wahlfreien Zugriff Wert legt. Wird ein Block angefragt, so muss er also erst von `ipfs` beschafft werden, dann entschlüsselt, dann dekomprimiert und dann mit eventuellen Modifikationen vereint werden.

Wird eine Datei schreibbar geöffnet, müssen zusätzlich die gemachten Änderungen zurück geschrieben werden. Problematisch ist dabei, dass im Moment nur die gesamte Datei neu zu `ipfs` hinzugefügt werden kann. Würde das bei jedem Aufruf der `Write()`–Funktion im FUSE–Layer geschehen, wäre das spätestens bei großen Dateien sehr ineffizient. Als Kompromisslösung wird jeder geschriebene Block samt seinen Offset im Hauptspeicher zwischengelagert. Erst beim Aufruf von `Close()` (Schließen des Dateideskriptors) oder `Flush()` (explizites Herausschreiben aller zwischengelagerten Daten) werden die gespeicherten Blöcke mit dem darunterliegenden Datenstrom wie in Abb. 6.4 kombiniert. Die kombinierte Version wird dann wieder komprimiert, verschlüsselt und `ipfs` übergeben. Die Implementierung ist etwas trickreich, da durch eine Modifikation auch der darunterliegende Datenstrom verlängert (durch mehrere `Write()`–Aufrufe am Ende) oder verkürzt werden kann (durch ein `Truncate()`).

Nachteilig an dieser Vorgehensweise ist vor allem, dass der Hauptspeicher sehr schnell überlaufen kann, wenn eine große Datei komplett neu geschrieben wird. Zukünftige Implementierungen sollten hier einzelne Blöcke auf die Festplatte auslagern können oder `ipfs` so erweitern, dass individuelle Blöcke direkt zurückgeschrieben werden können. Letzteres war im Rahmen dieser Arbeit zu zeitintensiv, um akkurat implementiert zu werden.

6.3.3. Repository Struktur

Abb. 6.5 zeigt den Aufbau eines Repositories auf der Festplatte, kurz nach dem Anlegen wenn `brigd` noch nicht läuft. Alle Daten sind nicht direkt im Repository hinterlegt, sondern liegen in einem Unterordner namens »`.brig`«. Ursprünglich sollte das FUSE–Dateisystem über das (größtenteils) leere Verzeichnis gelegt werden, um es wie einen normalen Ordner aussehen zu lassen. Das ist

[15] Siehe Wikipedia für eine Liste: https://de.wikipedia.org/wiki/Liste_der_Linux-Systemaufrufe

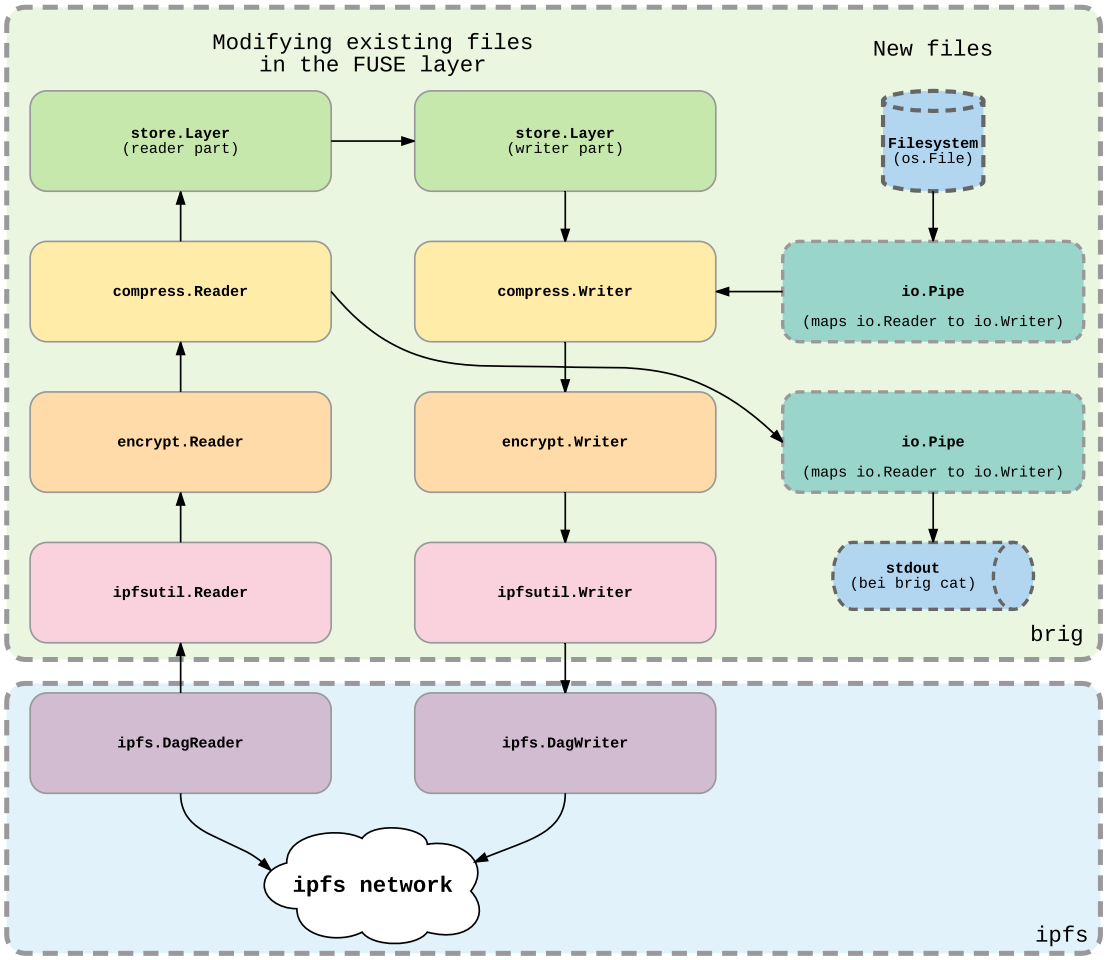


Abbildung 6.3.: Alle io.Reader und io.Writer auf einen Blick.

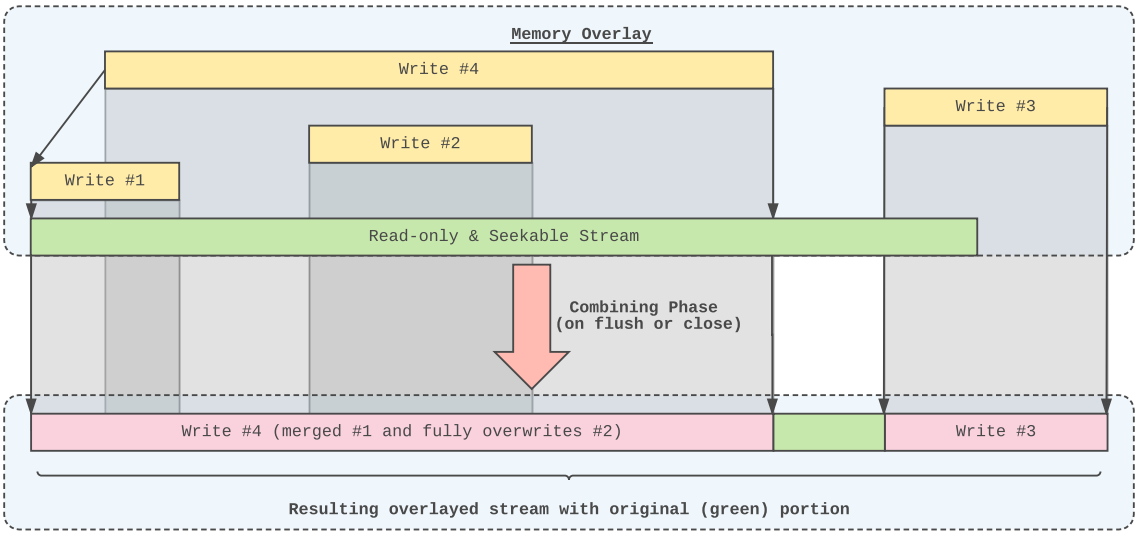


Abbildung 6.4.: Funktionsweise eines beschreibbaren Overlays.

```
/path/to/repo
└── .brig
    ├── index
    │   └── alice@wonderland.lit\desktop.locked
    ├── config
    ├── ipfs
    │   ├── blocks
    │   ├── config
    │   ├── datastore
    │   ├── version
    │   └── ...
    ├── master.key.locked
    ├── remotes.yml.locked
    └── shadow

5 directories, 8 files
```

Abbildung 6.5.: `tree`–Ausgabe auf das Verzeichnis des Repositories.

technisch möglich, wenn vor dem Erstellen des FUSE–Dateisystems ein offener Dateideskriptor auf das `.brig`–Verzeichnis vorhanden ist. Leider unterstützt `ipfs` dies nicht und stürzt beim versuchten Zugriff auf seine Datenbank ab.

Nach dem Anlegen eines Repositories sind einige Dateien noch verschlüsselt (Endung mit `.locked`). Erst durch Eingabe des Passworts beim Starten von `brigd` werden diese Dateien entschlüsselt. Der Schlüssel wird dabei vom Passwort mit der Schlüsselableitungsfunktion `scrypt`[29] generiert. Das Verschlüsselungsformat entspricht dabei dem in Abschnitt 5.4.1.1 beschriebenen Verfahren.

Ansonsten haben die Dateien folgenden Inhalt:

- `config`: Enthält die Konfiguration des `brig`–Repositories.
- `remotes.yml.locked`: Enthält für jedes bekannte Remote seine Prüfsumme und einen Zeitstempel, wann dieser zuletzt online war.
- `index/`: Enthält für jeden Benutzer eine BoltDB mit seinen Metadaten.
- `ipfs/`: Ein `ipfs`–Repository. Hier werden die eigentlichen Daten gespeichert. Die Struktur des Verzeichnisses selbst wird von `ipfs` bestimmt.
- `shadow`: Enthält die Prüfsumme der vom Nutzer angegebenen Passphrase. Wird zum Abgleich der Passphrase beim Öffnen des Repositories genutzt.
- `master.key`: Noch keine Verwendung. Wird zufällig beim Anlegen eines Repositories generiert. Soll als Basis einer Schlüsselhierarchie dienen (vgl. [1]).

6.3.4. Nennenswerte Bibliotheken

Einige Bibliotheken haben bei der Entwicklung von `brig` sehr geholfen. Bei der Auswahl wurde auf drei Kriterien geachtet:

- Lizenz der Bibliothek muss mit der APGLv3–Lizenz von `brig` kompatibel sein.
- Die Bibliothek sollte plattformübergreifend funktionieren.
- Die Bibliothek sollte möglichst rein in *Go* geschrieben sein. Dies vereinfacht die Installation, da neben *Go* keine weiteren Abhängigkeiten installiert werden müssen.

Nennenswert sind dabei folgende Bibliotheken, die sich alle auf *GitHub* befinden. In Klammern wird jeweils die Lizenz der Bibliothek mit angegeben:

- `urfave/cli`: Umfangreiche Bibliothek um Kommandozeilen zu parsen. (*MIT*)
- `golang/snappy`: Go–Implementierung des Snappy–Kompressionsalgorithmus. (*BSD–3–Clause*)
- `bkaradzic/go-lz4`: Go–Implementierung des LZ4–Kompressionsalgorithmus. (*BSD–3–Clause*)
- `dustin/go-humanize`: Enthält nützliche Konvertierungsfunktionen, um beispielsweise Bytes in eine passende, menschenlesbare Form zu formatieren. (*MIT*)
- `jbenet/go-multihash`: Implementiert die Enkodierung und Dekodierung des *Multihash*–Format. (*MIT*)
- `VividCortex/godaemon`: Wird benutzt, um den Pfad zur eigenen ausführbaren Datei plattformübergreifend zu finden. (*MIT*)
- `gogo/protobuf/proto`: Optimierte Version des Original–Protobuf–Compilers. (*BSD–3–Clause*)
- `codahale/chacha20poly1305`: Nutzt die Streaming–Cipher *ChaCha20* und die MAC *Poly1305*, um authentifizierte Verschlüsselung umzusetzen. (Siehe auch: [19], *MIT*).
- `chzyer/readline`: Komfortable Eingabe von Text auf dem Terminal. (*MIT*)
- `nbutton23/zxcvbn-go`: Prüft eine Passphrase auf ihre Entropie. (*MIT*)

6.3.5. Sonstiges

Logging: `brigd` nutzt eine farbige Log–Ausgabe und Unicode–Glyphen, um dem Entwickler das Erkennen von verschiedenen Log–Leveln zu erleichtern (siehe Abb. 6.6). Farbig ist die Ausgabe nur, wenn `brigd` im Vordergrund läuft und auf `stdout` ausgibt. Läuft der Daemon im Hintergrund, werden die Log–Ausgaben in eine Datei geschrieben und die Farbinformationen weggelassen.

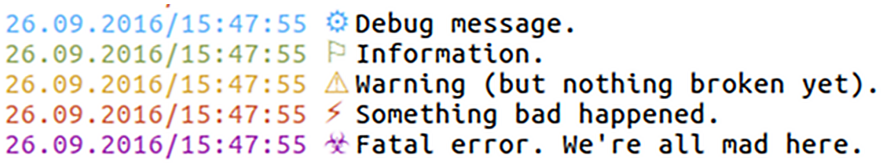


Abbildung 6.6.: Beispielhafte Ausgabe mit allen verfügbaren Log–Leveln

Konfiguration: Einige Parameter von `brig` sind konfigurierbar. Diese werden in einer menschenlesbaren YAML–Datei[16] gespeichert. Der Zugriff auf einen Wert erfolgt dabei durch einen mit ».« getrennten Pfad. So liefert der Schlüssel »`daemon.port`« dem Schlüssel »`port`« in dem assoziativen Array »daemon« (siehe Beispiel Listing 6.3.5).

```
daemon:
  port: 6666                        # Port von brigd.
ipfs:
  path: /tmp/alice/.brig/ipfs       # Pfad zum Repository.
  swarmport: 4001                   # Port von ipfs.
repository:
  id: alice@wonderland.lit/laptop   # Nutzername.
```

[16] https://de.wikipedia.org/wiki/YAML

Global–Config: Es ist möglich, mehrere `brig`–Repositories auf einem Rechner parallel laufen zu lassen. Dabei ist allerdings darauf zu achten, dass `brigd` zwei Ports pro laufender Instanz benötigt (4001 für `ipfs` und 6666 für `brigd` selbst). Deshalb hinterlegt jedes angelegte Repository in der sogenannten *Global Config* einen Eintrag, welche Ports es nutzt. Neu angelegte Repositories konsultieren die *Global Config*, um automatisch eine vernünftige Portkonfiguration zu erhalten. Die Konfiguration ist wie die lokale Konfiguration eine YAML–Datei und befindet sich im Home–Verzeichnis des Nutzers unter `.brig-config/`. Listing 6.3.5 zeigt ein Beispiel mit zwei unterschiedlichen Repositories. Die *Global Config* ist zudem dazu gedacht, globale Standardwerte für neue Repositories zu definieren (ähnlich wie die globale `git config`).

```
# ~/.brig-config/repos
repos:
  - alice@wonderland.lit/desktop:
      repopath: /home/alice/.brig
      daemonport: 6666
      ipfsport: 4001
  - rabbithole@wonderland.lit/desktop:
      repopath: /var/rabbithole/.brig
      daemonport: 6667
      ipfsport: 4002
```

Umgebungsvariablen: Das Verhalten von `brig` wird teilweise auch über Umgebungsvariablen gesteuert, sofern diese nicht von der Kommandozeile überschrieben werden. Momentan gibt es drei Variablen, die gesetzt werden können.

- `BRIG_PATH`: Falls gesetzt, operiert `brig` auf diesem Verzeichnis anstatt dem aktuellen Arbeitsverzeichnis. Kann dazu genutzt werden, um außerhalb des Repositories zu arbeiten.
- `BRIG_LOG`: Schreibt die Logdatei an den Pfad in der Umgebungsvariable.
- `BRIG_PORT`: Überschreibt den konfigurierten Port mit der Ganzzahl in der Umgebungsvariable.

6.4. Entwicklungsumgebung

Die gesamte Implementierung wurde auf einem herkömmlichen Linux–System mit aktuellen Softwarestand geschrieben und getestet. Als Editor kam »neovim«[17] mit dem »vim-go«–Pluginset zum Einsatz.

Neben der von *Go* mitgelieferten Toolbox werden für die Verwaltung von `brig` noch `glide`[18] und `gometalinter`[19] verwendet. Ersteres verwaltet alle Abhängigkeiten von `brig` und wird genutzt, um sie aktuell zu halten. Letzteres ist ein Programm zur statischen Code–Analyse. Es lässt viele verschiedene in der Go–Welt gebräuchlichen Programme auf den Quelltext laufen und sammelt deren Ergebnisse in einer konsistenten Ausgabe. Die Prüfungen dabei sind vergleichsweise strikt und umfangreich. Beispielsweise werden nicht nur undokumentierte Funktionen gefunden, sondern auch duplizierter Code. Es wurde versucht, ein Großteil der so gefundenen Probleme zu reparieren.

[17] https://neovim.io
[18] https://github.com/Masterminds/glide
[19] https://github.com/alecthomas/gometalinter

Der gesamte Quelltext wird mit `git` verwaltet und zu mindestens drei verschiedenen Rechnern synchronisiert. Dazu gehört der bereits genannte GitHub–Account (https://github.com/disorganizer/brig), sowie ein von Herrn Prof. Schöler dankenswerterweise bereitgestelltes GitLab–Repository[20]. Zusätzlich wird der Quelltext noch auf einem privaten Rechner synchronisiert und ist auf den Entwicklerrechnern vorhanden.

Dabei wird sämtlicher Code auf dem `master`–Branch entwickelt. Nach einem öffentlichen Release sollte der `master`–Branch stets den aktuellsten, stabilen Stand von `brig` widerspiegeln, während der `develop`–Branch alle möglicherweise instabilen Änderungen sammelt. Vom `develop`–Branch sollten `feature/<name>` oder `bugfix/<name>`–Branches abgezweigt werden, in denen unabhängig ein eigenes Feature entwickelt wird. Später werden diese »Feature«–Branches dann wieder mit dem `develop`–Branch zusammengeführt, welcher ebenfalls vor dem Release einer neuen Version mit dem `master`–Branch zusammengeführt wird. Jedes Release soll zudem mit einem *Tag* versehen werden, dessen Name sich nach den Regeln des *Semantic Versioning*[21] richtet.

Bei jedem veröffentlichten Commit auf GitHub werden zudem von der Continous–Integration–Plattform *Travis* alle Tests automatisch ausgeführt. Bei Fehlern wird man durch eine E–Mail benachrichtigt. Diese Plattform ist für freie Softwareprojekte dankenswerterweise kostenfrei.

6.5. Entwicklungshistorie

Der Beginn der Entwicklung reicht bis in den November des Jahres 2015 zurück. Zu diesem Zeitpunkt war `brig` konzeptuell noch anders gelagert und es wurde beispielsweise die Verwendung von `ssh` und `rsync` als Backend diskutiert. Erst nach der Beschäftigung mit `ipfs` und seinen Möglichkeiten entstand der Grundgedanke, der hinter dem heutigen `brig` steht.

6.5.1. Sackgassen bei der Entwicklung

Leider wurden auch einige Techniken sehr zeitaufwendig ausprobiert und wieder verworfen. Dazu gehört auch der geplante Einsatz von *XMPP*[22] als sicherer Steuerkanal und als Möglichkeit, ein Benutzermanagement zu implementieren. Nach kurzer Recherche stellte sich heraus, dass zum damaligen Zeitpunkt für *Go* noch keine verwertbaren Bibliotheken existierten. Daher wurde ein eigener XMPP–Client entwickelt, der in Kombination mit *Off-the-Record-Messaging (OTR)*[23] für eine sichere Verbindung zwischen zwei `brig`–Knoten sorgen sollte. Nach einigem Implementierungsaufwand stellte sich dieser als zu langsam und ineffizient heraus (teilweise Verbindungsaufbau > 30 Sekunden)[24]. Zudem handelt es sich bei *XMPP* um kein gänzlich dezentralisiertes Protokoll, da die meisten Nachrichten über zentrale Server geleitet werden. Von *XMPP* ist im heutigen Konzept nur das Format des Benutzernamens geblieben, welcher an die JID angelehnt ist.

Als Ersatz für XMPP wurde *MQTT*[25] erwogen. Dabei handelt es sich um ein offenes Machine–to–Machine Nachrichtenprotokoll. Clients registrieren sich bei einem (normalerweise zentralen) Broker

[20] https://r-n-d.informatik.hs-augsburg.de:8080/brig/brig
[21] Mehr Informationen hier: http://semver.org
[22] https://de.wikipedia.org/wiki/Extensible_Messaging_and_Presence_Protocol
[23] https://de.wikipedia.org/wiki/Off-the-Record_Messaging
[24] Alte Implementierung: https://github.com/disorganizer/brig/tree/253208a0651b8649d54b159024b2756319458b94/im
[25] https://de.wikipedia.org/wiki/MQ_Telemetry_Transport

auf benannte Kanäle (*Topics* genannt) und werden benachrichtigt, wenn ein anderer Client eine Nachricht auf einem registrieren Topic veröffentlicht. Die Idee war, jeden `brig`–Knoten zu einem MQTT–Broker zu machen. Dabei ist jeder Knoten auch ein *Client*, der auf den *Topics* des eigenen Brokers und aller anderen, benachbarten Knoten hört. Aus einem zentral aufgebauten Protokoll wurde so ein dezentrales Protokoll gemacht. Und obwohl die Lösung dem ursprünglichen Konzept von *MQTT* widersprach, hat ein Prototyp zufriedenstellend funktioniert. Nachteilig war aber vor allem die schwierige Absicherung des Datenverkehrs und dass ein zusätzlicher Port für MQTT gebraucht wurde. Als beinahe unlösbar hat sich auch die Notwendigkeit herausgestellt, MQTT über NAT–Grenzen hinweg zu betreiben. Zusammen haben diese »Sackgassen« zwei bis drei Monate Entwicklungszeit verschlungen.

Bevor das in Kapitel 5 beschriebene Datenmodell eingeführt wurde, wurde sehr lange Zeit ein simpleres Datenmodell genutzt. Dieses existierte alleine im Speicher und wurde zu bestimmten Anlässen in seiner Gesamtheit in die *BoltDB* geschrieben. Die Basis hat ein Patricia–Trie gebildet, der alle Pfade als Schlüssel und die Metadaten als deren zugeordnete Werte gespeichert hat. Aufgrund von Problemen bei der Erweiterbarkeit in Richtung Versionsverwaltung und Wartung wurde dieses Datenmodell mit dem heutigen ersetzt. In der Anfangszeit der Entwicklung wurde `ipfs` nicht als Bibliothek genutzt, sondern es wurde direkt das `ipfs`–Programm als Subprozess für jedes auszuführende Kommando (Beispiel: `ipfs add`) gestartet. Aufgrund von schlechter Effizienz wurde dieser Ansatz nicht weiter verfolgt.

Als Lehre wurden drei Abstraktionsschichten in `brig` eingebaut, die die Austauschbarkeit einiger Komponenten erleichtern soll. Diese sind wie folgt:

- Abstraktionsschicht zwischen `brig` und `ipfs`. Jede benötigte `ipfs`–Funktion erhält eine eigene Funktion im Paket `ipfsutil`. Sollte sich die Semantik bestimmter Funktionen ändern, so kann dies an zentraler Stelle angepasst werden, auch wenn `ipfs` selbst bisher nicht sinnvoll zu ersetzen ist.
- Abstraktionsschicht zwischen `brig` und BoltDB. Es wird nicht direkt auf die API von BoltDB zugegriffen. Stattdessen kommt auch hier ein »Wrapper« zum Einsatz, hinter dem auch eine andere Key–Value–Datenbank, der Hauptspeicher oder sogar ein normales Dateisystem stehen können.
- Abstraktionsschicht zwischen `brig` und Transfer–Schicht. Diese wurde zur selben Zeit eingeführt wie das oben genannte MQTT–Experiment. Deshalb konnte MQTT später relativ schnell durch ein eigenes Kommunikationsprotokoll basierend auf `ipfs` ersetzt werden.

6.5.2. Beiträge zu anderen Projekten

Im Laufe der Entwicklung wurden einige kleinere Beiträge zu anderen Projekten gemacht. Teilweise auch zu Projekten, die zu diesem Zeitpunkt gar nicht mehr von `brig` genutzt werden. Diese werden hier der Vollständigkeit halber in umgekehrter chronologischer Reihenfolge aufgelistet:

- https://github.com/bazil/fuse/pull/152: Option für »*AllowNonEmpty*« hinzugefügt.
- https://github.com/ipfs/go-ipfs/issues/2567: Fehler in einer `Seek()` Funktion von `ipfs`.
- https://github.com/ipfs/go-ipfs-util/pull/1: Konstante für den *DefaultHash* hinzugefügt.
- https://github.com/tang0th/go-ecdh/pull/1: Änderung kaputter Import–Pfade.
- https://github.com/cathalgarvey/go-minilock/issues/8: Crashreport für *minilock*.
- https://github.com/tucnak/climax/pull/3: Gruppierung mehrerer Subkommandos.
- https://github.com/chzyer/readline/pull/18: Beispiel für ein zuvor angeregten Passwortprompt hinzugefügt.
- https://github.com/ipfs/go-ipfs/pull/1981: Erwähnung von `IPFS_PATH` in der Hilfe.
- https://github.com/tsuibin/goxmpp2/pull/1: Flexibleres Nachschlagen des SRV–Eintrags.

7 Usability

In diesem Kapitel werden Anforderungen beleuchtet, die `brig` zu einer für den »Otto–Normal–Nutzer« benutzbaren Software machen soll. Zudem sollen die in Zukunft notwendigen Schritte beschrieben werden, um diese Anforderungen umzusetzen. Dazu gehört unter anderem die Konzeption einer grafischen Oberfläche.

7.1. Einführung

Ob eine Bedienoberfläche verständlich ist oder ästhetisch auf den Benutzer wirkt, ist leider sehr subjektiver Natur. Es können nur empirisch Daten gesammelt werden, ob ein gewisser Prozentanteil der Nutzer die Software verständlich und ästhetisch findet. Aus diesem Grund ist der unten gezeigte Vorschlag für eine Bedienoberfläche lediglich ein Konzept unter vielen Möglichkeiten. Der Begriff »Usability« wird dabei gleichbedeutend mit dem deutschen Wort »Gebrauchstauglichkeit« benutzt. Da es aber keine einheitliche Übersetzung des Begriffs gibt, wird der englische Originalbegriff verwendet.

7.2. Anforderungen an die Usability

Eine besondere Schwierigkeit bei `brig` ist, dass Sicherheit, Funktionalität und Usability gegeneinander abgewogen werden müssen. Zu viel und zu schnell präsentierte Funktionalität erschwert dem Nutzer den Einstieg in die Software. Zu viele sichtbare Sicherheitsmechanismen schrecken den normalen Nutzer ohne technischen Hintergrund ab. Hingegen werden Nutzer mit technischen Hintergrund tendenziell eher mehr Funktionalität und striktere Sicherheitsmechanismen erwarten.

Es ist daher schwierig, die Anforderung beider Nutzergruppen von einer gemeinsamen Oberfläche erfüllen zu lassen. Deshalb erscheint es sinnvoller mehr als eine Oberfläche anzubieten. Momentan wurde dabei nur zwischen der *Kommandozeile* (für technisch versierte Nutzer) und einer grafischen Oberfläche, die im Folgenden `brig-ui` genannt wird, unterschieden.

Für beide Varianten lassen sich trotzdem gemeinsame Anforderungen finden:

1) Die Oberfläche muss möglichst immer im Hintergrund bleiben. Nur wenn sie benötigt wird soll der Benutzer sich mir ihr beschäftigen müssen.
2) Es sollte nur das Minimum an nötigen Informationen angezeigt werden, um den »Cognitive Load« (siehe auch [30]) des Nutzers zu minimieren.
3) Die Oberfläche soll einfach installierbar sein. Da Nutzer meist einfach »nur ein Problem« lösen wollen, greifen sie oft zur schnellst möglich installierbaren und damit nutzbaren Variante. Zudem mögen viele Nutzer nicht die Oberfläche, vom eigentlichen, dahinter liegenden Programm unterscheiden können.
4) Die Oberfläche muss dem Nutzer vertraute Konzepte (Listen, Auswahlmenüs…) und Metaphern (Dateien, Verzeichnisse…) bieten.

5) Die Oberfläche muss konsistent in ihrer Benutzung sein. Sieht etwas in der Applikation gleich oder ähnlich aus, so muss es auch gleich oder ähnlich funktionieren. Im Umkehrschluss sollen auch konsistent Begriffe wie »Passphrase« statt »Passwort« genutzt werden, um anzuzeigen, dass die Eingabe länger sein soll als ein herkömmliches »Passwort«.
6) Die Oberfläche muss sich möglichst gut in das System des Benutzers integrieren. Im Falle einer grafischen Oberfläche bedeutet dies die Nutzung einer nativen Desktopapplikation (anstatt einer Webapplikation), die beispielsweise nativ *Drag&Drop* unterstützt und ein *Trayicon* anzeigen kann.
7) Die Oberfläche soll ein möglichst zeitgemäßes Design aufweisen, welches neue Nutzer nicht abschreckt.
8) Alle verwendeten Texte sollten in die lokale Sprache des Benutzers übersetzt werden.
9) Die Funktionsweise der Oberfläche soll sich durch Konfiguration an die Bedürfnisse des Benutzers anpassen lassen, aber vernünftige Standardwerte mitbringen.
10) Die Oberfläche sollte dem Nutzer direkt Feedback geben, ob eine Aktion erfolgreich war. Unwichtige Fehler sollten nach Möglichkeit ignoriert werden, wichtige Fehler sollten Handlungsinstruktionen beinhalten.

Diese Anforderungen wurden teilweise von *www.usabilitynet.org*[1] abgeleitet und ergaben sich teilweise nach Betrachtung der existierenden Synchronisationswerkzeuge. Die Liste ist subjektiv und keineswegs komplett. Obwohl beispielsweise `syncthing` sich als *»Easy to use«*[2] bezeichnet, verletzt es unter anderem die Anforderung *2)* und präsentiert dem Nutzer in der Hauptansicht die Systemauslastung (siehe Abb. 2.6) und andere Informationen, die in diesem Kontext nicht wichtig sind.

7.3. Die Kommandozeile

Momentan ist die Kommandozeile `brigctl` die einzige, implementierte Möglichkeit die gesamte Funktionalität von `brig` zu nutzen. Die genaue Funktionsweise der Kommandozeile wird in Anhang A beleuchtet. Beim Design der Optionen und Unterkommandos wurde darauf geachtet, dass `git`–Nutzern die Benutzung schnell *vertraut* vorkommt, wo die Konzepte sich ähneln (`brig remote`/`remote`). Wo sie sich unterscheiden, wurden bewusst andere Namen gewählt (`brig stage` statt `git add` und `brig sync` statt `git pull`/`push`). Das Projekt `gitless`[3] zeigt zudem einige Usability–Verbesserungen an der `git`–Kommandozeile auf. Einige dieser Ideen könnten für die weitere Entwicklung genutzt werden.

Eine eingebaute Hilfe kann für ein bestimmtes Kommando mit dem Befehl `brig help <topic/command>` angezeigt werden. Das initiale Anlegen eines Repositories erfordert die Eingabe einer Passphrase mit einer bestimmten Mindestentropie. Wie man in Abb. 7.1 erahnen kann, wird die Entropie »live« bei der Eingabe des Passworts angezeigt, um den Nutzer direkt Feedback zu geben. Momentan wurde die Kommandozeile noch nicht in weitere Sprachen übersetzt, da sie sich genau wie der Rest der Implementierung noch ändern kann. In späteren Versionen könnte zudem ein lokalisiertes Tutorial angeboten werden, welches die wichtigsten Konzepte nach Eingabe von »`brig tour`« vermittelt.

[1] http://www.usabilitynet.org/trump/methods/recommended/requirements.htm
[2] https://syncthing.net
[3] Siehe auch: http://gitless.com

```
$ brig init alice@wonderland.lit
✔  74    New passphrase:
✔ Well done! Please re-type your password now:
⚡  30 Retype passphrase:
⚠ This did not seem to match. Please try again.
✔  74 Retype passphrase:
22.09.2016/19:38:07 ⚠ ==========================================
22.09.2016/19:38:07 ⚠ Please make sure to remember the passphrase!
22.09.2016/19:38:07 ⚠ Your data will not be accessible without it.
22.09.2016/19:38:07 ⚠ ==========================================
22.09.2016/19:38:07 ⚐ generating 2048-bit RSA keypair...
```

Abbildung 7.1.: Angabe der Passphrase beim Anlegen eines neuen Repositories.

Eine weitere Verbesserung wäre die Unterstützung von »Shorthashes«. Der Benutzer muss derzeit immer eine volle Prüfsumme angeben (`QmSiM3qaUMxCrLiWwVvEeGZTrKUXLD7bULo22WYoGfHwZD`), auch wenn meist ein kleiner Präfix (`QmSiM3q`) davon bereits eindeutig identifizierbar ist. In der Ausgabe von `brigctl` sollte dann auch möglichst die Präfixvariante bevorzugt werden, um die Ausgabe klein und verständlich zu halten.

7.4. Grafische Oberfläche

Für normale Benutzer ist eine grafische Oberfläche unabdingbar. Für die Akzeptanz der Oberfläche ist es wichtig, dass sie dem Benutzer vertraute Konzepte bietet. Daher wird ein großer Teil der Benutzung durch einen normalen Dateisystemordner abgewickelt, der sich kaum von anderen Ordnern unterscheidet. Daher hat die grafische Oberfläche eher die Aufgabe einer Konfigurationsanwendung und eines Einrichtungsassistenten, der nur bei Bedarf aufgerufen wird. Konkret sind die nötigen Aufgabenbereiche wie folgt:

- Einrichtung und Konfiguration eines neues Repositories.
- Oberfläche zur Versionsverwaltung.
- Schalter, um zwischen Online- und Offline–Modus zu wechseln.
- Hinzufügen und Verwalten von Remotes.
- Integrierter Dateibrowser, um Dateien zu verwalten und zu pinnen.
- Einstellungen zur Funktionsweise und Sicherheit.

Bestehende grafische Oberflächen sind aus Portabilitätsgründen meist web–basiert und fügen sich daher meist nicht optimal in eine Desktopumgebung ein. Daher wurde das nachfolgende Konzept als native Desktopanwendung für den GNOME–Desktop[4] entworfen. Dabei wurde die Oberflächenbibliothek *GTK+*[5] benutzt. Neben den obigen Anforderungen wurde versucht, möglichst alle Regeln der *»Gnome Human Interface Guidelines«* (GNOME HIG[6]) umzusetzen. Es handelt sich dabei um eine Anleitung des GNOME–Projekts, um den Oberflächenentwurf zu vereinfachen und einheitlich zu gestalten. Offizielle GNOME–Anwendungen müssen diesen Guidelines folgen.

GTK+ wurde benutzt, weil der Autor sich mit dieser Bibliothek auskennt und bereits eine im Aussehen »ähnliche« Anwendung geschrieben hat, die als Basis für unten stehende Mockups benutzt wurde. Leider bietet *GTK+* für die Programmiersprache *Go* noch keine native Unterstützung. Alternativ wäre

[4] Eine freie Desktopumgebung für Linux (https://www.gnome.org)
[5] Eine freie GUI–Bibliothek (http://www.gtk.org)
[6] Siehe auch: https://developer.gnome.org/hig/stable

daher eine Umsetzung, einer Bibliothek wie *Gallium*[7] zu evaluieren. Diese zeigt, vereinfacht gesagt, eine Weboberfläche als Desktopanwendung und integriert auch native Elemente wie ein Trayicon.

7.5. Mockups von brig-ui

In Abb. 7.2 findet sich eine Übersicht über alle Bildschirme der Oberfläche, wobei jeder Bildschirm für eine andere Aufgabe zuständig ist. Jedem Bildschirm gemein ist die sogenannte »*Headerbar*«, eine etwas breitere Fensterleiste, in der eigene Knöpfe platziert werden können. Auf der linken Seite derselben findet sich ein Schalter, der anzeigt ob man mit dem Netzwerk verbunden ist. Ein Klick auf diesen trennt die Verbindung. Auf der rechten Seite findet sich ein Knopf mit einer Lupe, der die Anwendung in den Suchmodus schaltet. Die Suche ist kontextspezifisch, findet also je nach Bildschirm etwas anderes. Neben dem Suchknopf findet sich ein Knopf mit einem Zahnrad. Bei einem Klick darauf öffnet sich das in Abb. 7.3 gezeigte Menü, von dem aus jeder weitere Bildschirm erreichbar ist. Meist kommt man aber durch das Ausführen bestimmter Aktionen automatisch auf einen anderen Bildschirm, ohne dass man das Menü bemühen muss. Im Folgenden werden die Aufgaben der einzelnen Bildschirme besprochen.

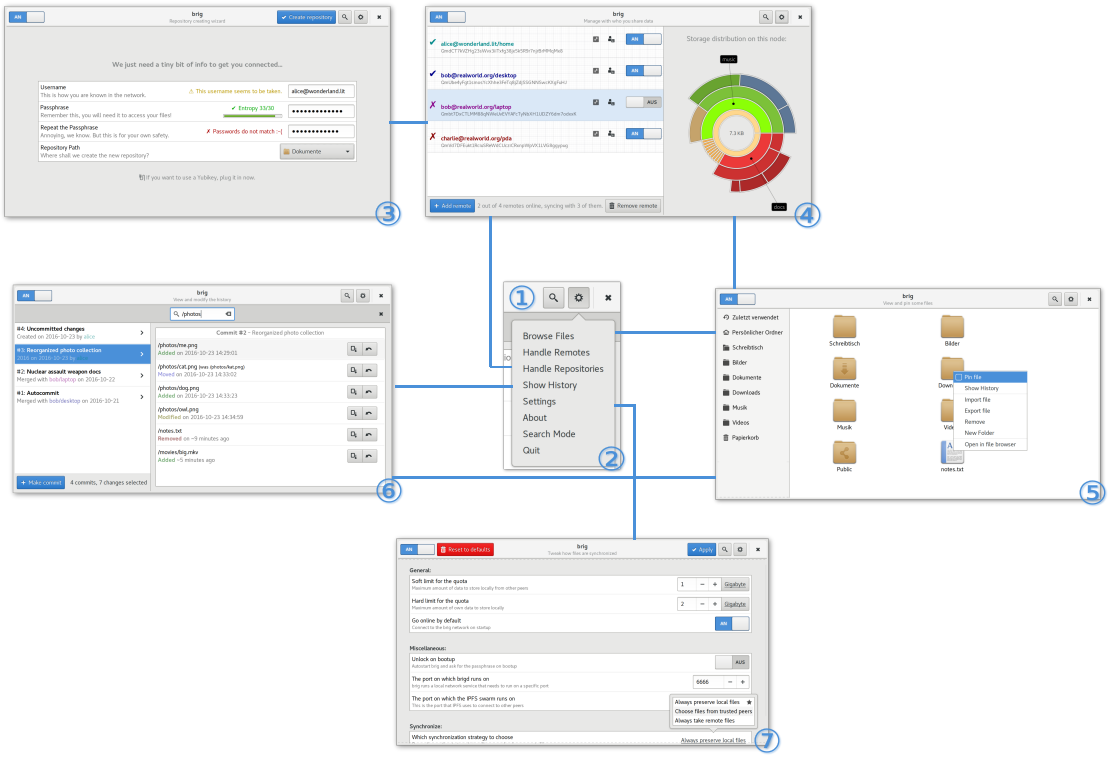


Abbildung 7.2.: Übersicht über das GUI–Konzept, bestehend aus fünf Bildschirmen.

7.5.1. Anlegen eines neuen Repositories

Dieser Bildschirm taucht beim erstmaligen Starten der grafischen Oberfläche auf, sofern kein vorhandenes `brig` Repository gefunden werden konnte. Der Bildschirm fragt alle Daten ab, die auch der Befehl »`brig init`« benötigt. Ein Vorteil der Oberfläche ist dabei, dass dem Nutzer direkt Feedback

[7] Siehe auch: https://github.com/alexflint/gallium

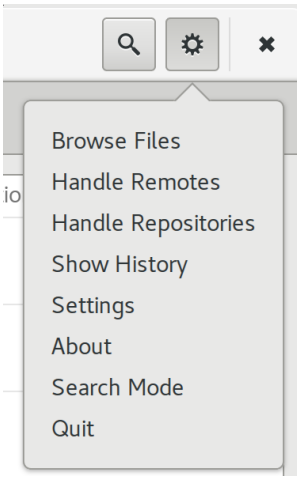


Abbildung 7.3.: Das Menü hinter dem »Zahnrad«–Knopf.

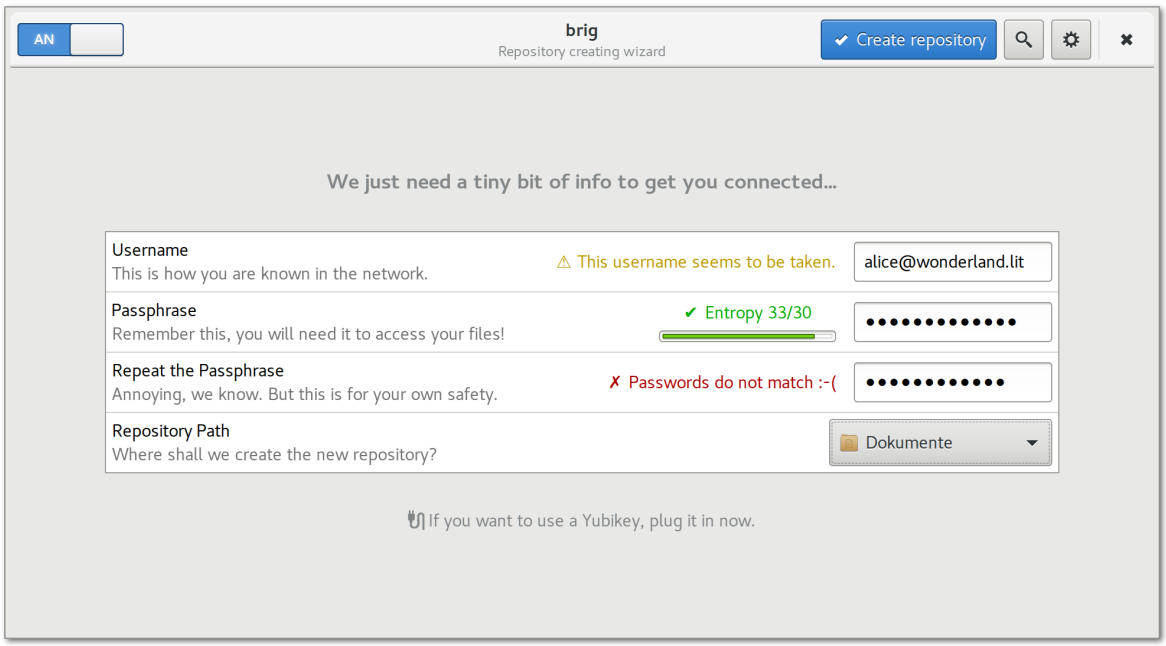


Abbildung 7.4.: Mockup: Bildschirm zum Anlegen eines Repositories.

bei der Eingabe gegeben werden kann. Konkret wird dabei der Nutzername auf formale Korrektheit überprüft und ob bereits ein solcher Name vergeben wurde. Es wird zudem geprüft, ob das Passwort einer bestimmten Mindestentropie entspricht und ob das wiederholte Passwort mit dem ersten übereinstimmt. Dadurch, dass der Nutzer mithilfe der Oberfläche den Anlegeort für das neue Repository auswählt, wird ein fehlerhafter Pfad ausgeschlossen.

In der späteren Entwicklung soll `brig` auch mit Geräten zur Zweifaktorauthentifizierung (siehe [1]) wie dem YubiKey[8] zusammenarbeiten. Daher wird am unteren Bildschirmrand eine entsprechende Nachricht angezeigt. Entsprechende Schaltflächen zur Konfiguration werden erst angezeigt, wenn ein angeschlossener YubiKey erkannt wurde.[9]

Ein auch im Folgenden häufig verwendetes Designelement ist das Hervorheben einer Aktion als »Empfohlen«. In Abb. 7.4 wird die »Create Repository«–Aktion blau hervorgehoben, um dem Nutzer anzuzeigen, dass dies die naheliegendste Aktion ist, die er vermutlich nehmen wird. Drückt man diese »empfohlene Aktion«, so wird das Repository angelegt und der Hintergrunddienst `brigd` gestartet. Im Erfolgsfall wird eine Wischanimation zum Remote–Bildschirm hin angezeigt. (siehe Abschnitt 7.5.2).

7.5.2. Verwalten und Hinzufügen von Remotes

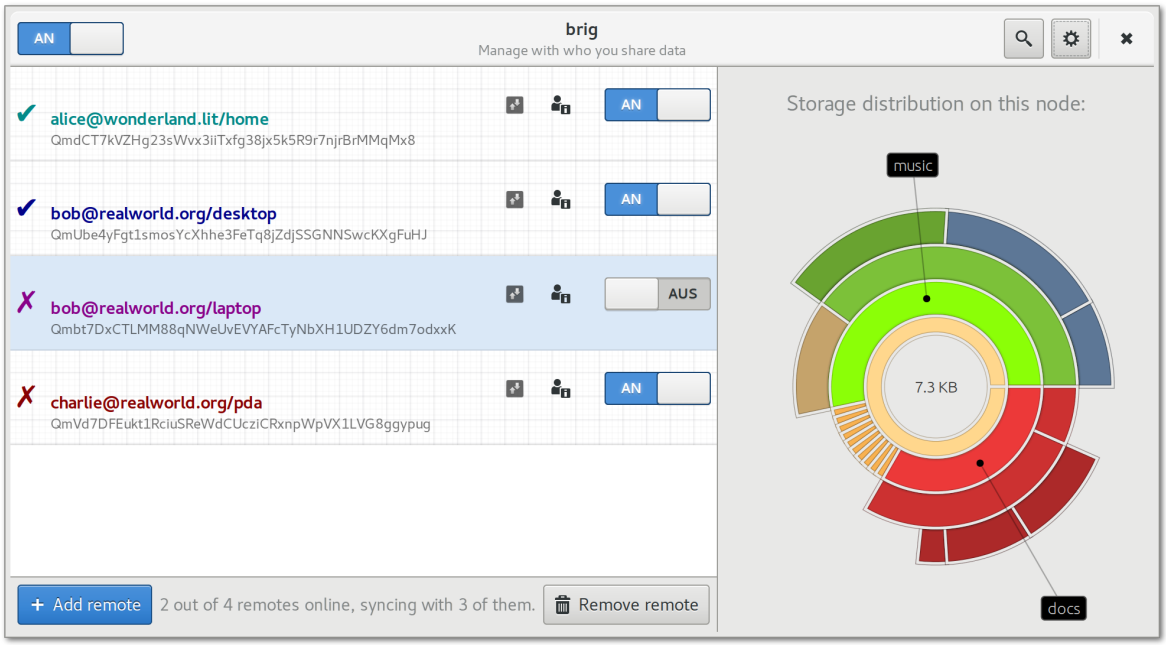


Abbildung 7.5.: Mockup: Verwalten und Hinzufügen von Remotes.

In dieser Ansicht kann der Nutzer existierende Remotes verwalten und Neue hinzufügen. Eine Idee, die vom Instant–Messanger *Signal* übernommen wurde, ist die Einfärbung eines Remotes mit einer bestimmten Farbe. Dies soll dem Nutzer helfen, den Kontakt mit dieser Farbe zu assoziieren und stellt gleichzeitig ein Sicherheitsmechanismus dar, da die Farbe basierend auf der Identitäts–Prüfsumme des Gegenübers gewählt wird. Ändert sich diese, so wird auch eine andere Farbe angezeigt.

Auf der linken Seite des Bildschirms findet sich eine Liste aller bekannten Remotes. Ist die Liste

[8] Siehe auch: https://en.wikipedia.org/wiki/YubiKey
[9] Dies entspricht Anforderung 2).

leer, wird dort ein Hinweis angezeigt, dass noch keine Remotes vorhanden sind und man durch die »empfohlene Aktion« unten links ein neues Remote anlegen kann.

Jedes Remote wird durch einen Eintrag in der Liste dargestellt. Das »✓« oder »✗« am Anfang indiziert dabei, ob das betreffende Remote online ist. Daneben wird in jedem Eintrag der Name des Remotes und seine Prüfsumme angezeigt. Eventuell wäre hier die alleinige Anzeige des Nutzernamens (`Alice` statt `alice@wonderland.lit/home`) benutzerfreundlicher und weniger verwirrend, sofern der Name `Alice` eindeutig unter den Remotes ist. Auch die Anzeige eines Zeitstempels der letzten Synchronisation wäre denkbar. Zur Rechten jeder Zeile finden sich drei Schaltflächen, die (in dieser Reihenfolge) folgendes bewirken: sofortiges Synchronisieren (Pfeilknopf), Öffnung eines Detailfensters zum entsprechenden Remote und das An- und Ausschalten der automatischen Synchronisation mit diesem Remote. Remotes mit denen automatisch synchronisiert wird, werden mit einem Hintergrund hinterlegt, der kariertem Papier ähnelt. Eine Synchronisation ist nur möglich, wenn `brig` im Online–Modus ist. Im Suchmodus können die Namen der Remotes zusätzlich durch Angabe eines Stichwortes gefiltert werden.

In der unteren Statusleiste wird zudem in Zahlen zusammengefasst, wie viele Remotes online sind und mit wie vielen Remotes synchronisiert wird. Der Knopf zum Löschen eines Remotes wird nur dann angezeigt, wenn mindestens ein Remote aus der Liste ausgewählt wurde (hellblau hinterlegt).

Die rechte Seite des Bildschirms besteht aus einem segmentierten Kreisdiagramm. Es wird nur angezeigt wenn genau ein Remote ausgewählt ist. Allerdings kann auch die Trennlinie in der Mitte des Bildschirms benutzt werden, um das Diagramm auszublenden, indem es auf die rechte Seite geschoben wird. Das Diagramm selbst zeigt an, welche Teile der synchronisierten Daten das Gegenüber physikalisch bei sich speichert. Die Gesamtmenge wird in der Mitte als Dateigröße gezeichnet. Im Beispiel speichert `bob@realworld.org/laptop` größtenteils die Dateien aus dem `music`- und aus dem `docs`–Ordner. Die Anzeige kann auch durch den Klick auf ein beliebiges Segment verfeinert werden. Dann werden nur noch dieses Segment und seine Untersegmente angezeigt. Ein Klick in die Mitte des Diagramms führt dabei wieder auf die oberste Segmentebene zurück.

Ein gewisses Usability–Problem stellt noch die initiale Authentifizierung dar. Der Dialog, der nach einem Klick auf »`Add Remote`« erscheint (nicht gezeigt), fragt nur nach den Nutzernamen und der Prüfsumme des Gegenübers. Dies setzt voraus, dass der Nutzer die Prüfsumme kennt, indem beide Teilnehmer sich vorher über einen Seitenkanal ausgetauscht haben. Leider ist der Austausch der Prüfsumme schwierig, da es sich um eine schwer merkbare Zeichenkette handelt. Eine mögliche Lösung für dieses Dilemma wäre die Einführung von QR–Codes (siehe Abb. 7.6), welche die Prüfsumme des Gegenübers visuell enkodieren. Treffen sich beispielsweise zwei Teilnehmer persönlich, so könnten sie ihre Mobiltelefone benutzen, um den QR–Code einzuscannen und zu verifizieren. Auch würde sich der QR–Code eignen, um auf Visitenkarten abgedruckt zu werden.

7.5.3. Dateibrowser

Der Dateibrowser zeigt alle Dateien an, die der jeweilige Synchronisationsteilnehmer verwaltet. Dies entspricht einer grafischen Sicht auf den FUSE–Dateisystemordner. Den Hauptunterschied bilden die zusätzlichen Optionen im Kontextmenü:

[10] Quelle: https://commons.wikimedia.org/wiki/File:QRCodeWikipedia.svg

Abbildung 7.6.: Ein typischer QR–Code[10].

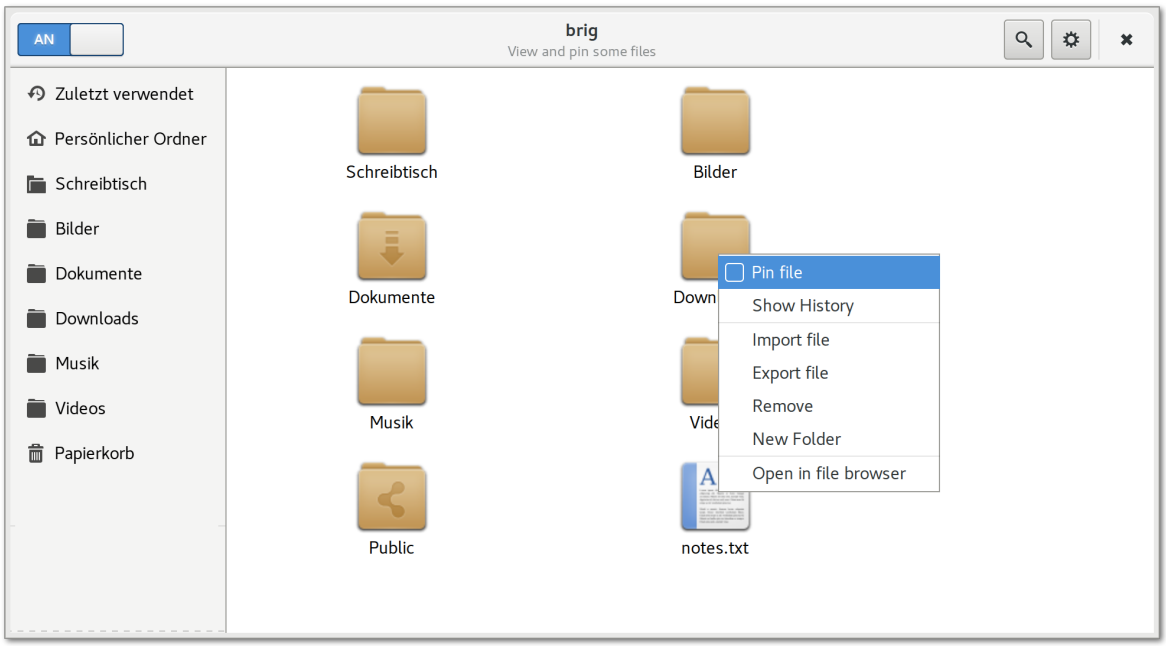


Abbildung 7.7.: Mockup: Bildschirm des Dateibrowsers.

- *Pin file:* Setzt oder entfernt einen Pin für die Datei oder rekursiv für das Verzeichnis.
- *Show History:* Wechselt zum Versionsverwaltungsbildschirm und zeigt die Historie der Datei (siehe Abschnitt 7.5.4).
- *Import/Export file:* Speichere Datei auf der Festplatte ab. Entspricht `brig stage` und `brig cat`.
- *Remove:* Entfernt die Datei im *Staging-Bereich*.
- *New Folder:* Entspricht `brig mkdir`.
- *Open in file browser:* Öffnet den Dateibrowser des Systems. Dies ist nützlich wenn komplexere Features benötigt werden.

Die Ansicht ist zudem durchsuchbar. Wird ein Teil eines Pfades eingegeben, so werden alle Dateien angezeigt, die dieses Pfadfragment beinhalten. Die Oberfläche ähnelt dabei sehr dem Dateibrowser *Nautilus*[11], da die selben Widgets zur Anzeige der Dateien benutzt werden.

7.5.4. Versionsverwaltung

Dieser Bildschirm bietet dem Nutzer Zugriff auf die eingebaute Versionsverwaltung. Die Ansicht ist zweigeteilt. Auf der linken Seite findet sich eine Liste mit allen bekannten Commits (entspricht etwa `brig log`). Jede Commitzeile enthält dabei den Index des Commits, der Commit-Message, dem Erstellungsdatum und dem farbig hervorgehobenen Autor. Auf der rechten Seite jeder Zeile findet sich der *Checkout*-Button, mit dem der aktuelle Stand auf den Stand im ausgewählten Commit zurücksetzbar ist. Unter dem Namen *Uncommited Changes* findet sich an oberster Stelle zudem immer der *Staging Commit*. Im linken, unteren Bereich wird zudem eine Statusleiste eingeblendet, in der als empfohlene Aktion das Anlegen eines neuen Commits möglich ist. Diese Aktion wird nur eingeblendet wenn sich `HEAD` und `CURR` unterscheiden.

Auf der rechten Seite werden für den aktuell ausgewählten Commit alle darin gemachten Änderungen aufgelistet. Dabei wird für jeden Checkpoint des Commits eine Zeile angezeigt. Diese beinhaltet den Pfadnamen, den Änderungstyp (farbig kodiert) und den Änderungszeitpunkt. Liegt der Änderungszeitpunkt noch nicht lange zurück, so wird er relativ zum aktuellen Zeitpunkt angegeben (»*5 minutes ago*«). Auf der rechten Seite jeder Zeile finden sich zwei Knöpfe. Der Linke erlaubt den Export der Datei zu einem beliebigen Pfad im Dateisystem im entsprechenden Zustand. Der rechte Knopf setzt im *Staging-Commit* die entsprechende Datei auf diesen Stand zurück.

Diese Ansicht ist durchsuchbar. Wird ein Pfad eingegeben (im Beipsiel `/photos`), so werden alle Commits angezeigt, in denen der `/photos`-Ordner verändert wurde.

7.5.5. Einstellungen

Über den Einstellungsbildschirm sind alle verfügbaren Einstellungen erreichbar. Die Einstellungen sind in mehrere Kategorien aufgeteilt (hier *General*, *Miscellaneous*, und *Synchronize*). Jedes Einstellungsmerkmal entspricht dabei einer Zeile, die links jeweils eine kurze und eine etwas längere Beschreibung der Einstellung beinhaltet. Rechts findet sich je eine Schaltfläche, die den aktuellen Wert anzeigt und eine Modifikation erlaubt. Dabei erhalten Größenangaben (*Soft limit*) entsprechend eine Schaltfläche, um eine physikalische Größe einzustellen, während boolesche Werte (*Go online by default*) einen simplen Aus/An-Schalter erhalten. Enumerationswerte, bei denen es eine genau

[11] Siehe auch: https://de.wikipedia.org/wiki/Nautilus_(Dateimanager)

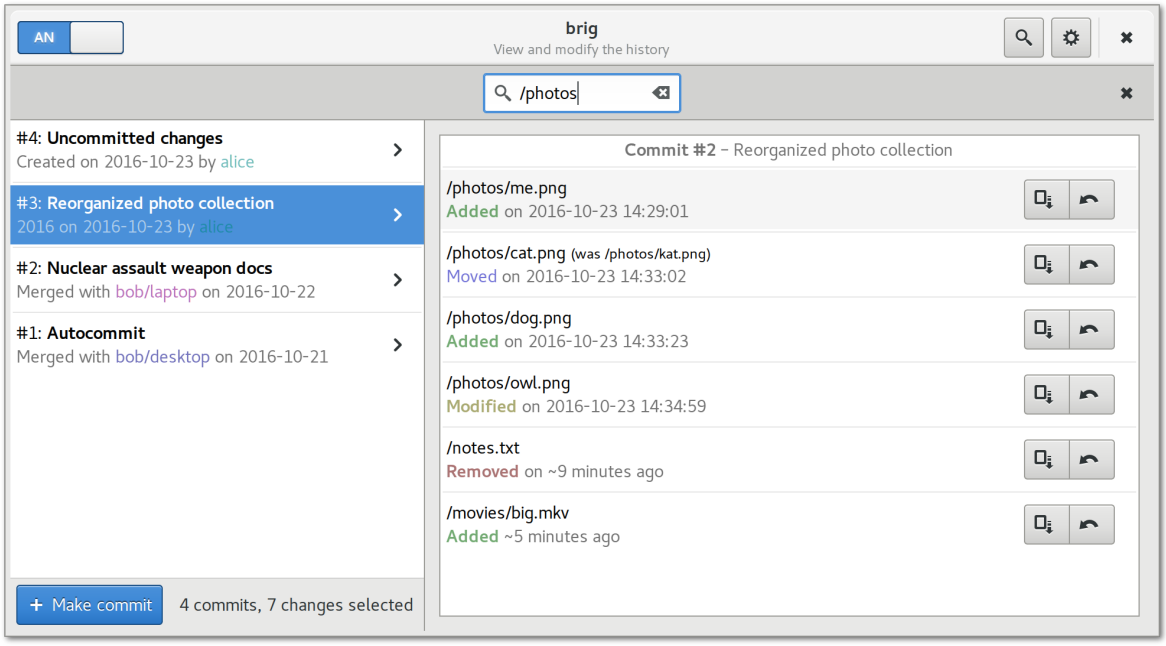


Abbildung 7.8.: Mockup: Bildschirm zur Versionsverwaltung.

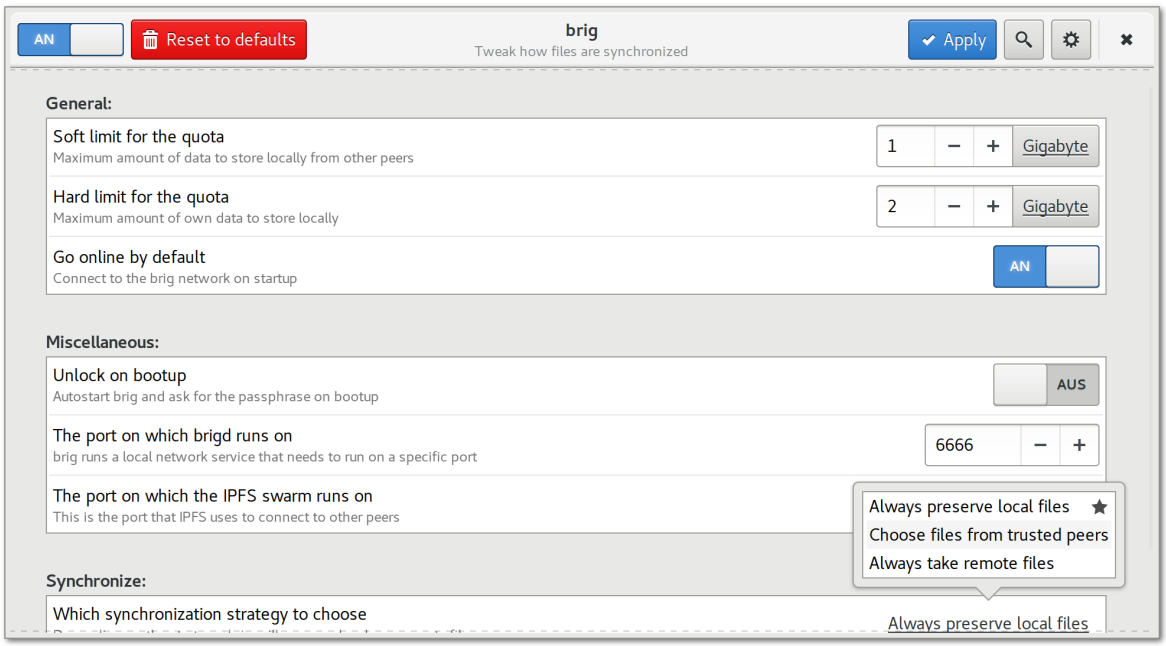


Abbildung 7.9.: Mockup: Bildschirm des Einstellungseditors.

festgesetzte Menge an Auswahlmöglichkeiten gibt (*Which synchronization strategy to choose*), zeigen nach einem Klick auf die Schaltfläche ein Auswahlmenü an. In diesem werden alle Möglichkeiten angezeigt, wobei die Standardmöglichkeit mit einem Stern gekennzeichnet wird und die ausgewählte Möglichkeit grau hinterlegt wird.

Die Änderung der Einstellungen muss explizit mit dem (empfohlenen, da blau hervorgehobenen) »Apply«–Knopf bestätigt werden. Wechselt man die Ansicht ohne zu bestätigen, so werden die Änderungen nicht übernommen. Möchte man alle Werte auf »Werkseinstellungen« zurücksetzen, so kann der Nutzer den *»Reset to defaults«*–Knopf betätigen. Dieser ist rot hervorgehoben, um anzuzeigen, dass es sich hierbei um eine destruktive Operation handelt.

Auch diese Ansicht ist durchsuchbar. Wird ein Stichwort eingegeben, so werden nur diejenigen Zeilen angezeigt, deren kurze oder lange Beschreibung dieses Stichwort enthalten.

8 Evaluation

In diesem Kapitel wird die Implementierung und die dahinter stehende Architektur auf Schwächen untersucht. Es wird gezeigt, was die Software nicht zu leisten vermag und welche eingangs definierten Anforderungen sie (noch) nicht erfüllen kann. Abgeschlossen wird das Kapitel mit verschiedenen Geschwindigkeitsmessungen, sowie einigen Konzepten zur weiteren Entwicklung.

8.1. Was `brig` *nicht* ist

`brig` kann nicht die beste Alternative in allen Bereichen sein. Keine Software kann die sprichwörtliche »eierlegende Wollmilchsau«[1] sein und sollte auch nicht als solche benutzt werden. Insgesamt ist es für folgende Bereiche weniger geeignet:

High Performance: Besonders im Bereich Effizienz kann es nicht mit hochperformanten Cluster–Dateisystemen wie Ceph[2] oder GlusterFS[3] mithalten. Das liegt besonders an der sicheren Ausrichtung von `brig`, welche oft Rechenleistung zugunsten von Sicherheit eintauscht (siehe Abschnitt 8.4).

Echtzeitanwendungen: Schreibt ein Nutzer etwas in eine Datei, so ist diese Änderung nicht augenblicklich anderen Nutzern zugänglich. Selbst wenn Live–Updates (siehe Abschnitt 5.4.2) verfügbar sind, kann `brig` selbst entscheiden, wann die Änderungen synchronisiert werden.[4] Insbesondere macht es beispielsweise kaum Sinn, SQL–Datenbanken von `brig` synchronisieren zu lassen. Hierfür gibt es weitaus bessere Alternativen wie *CockroachDB*[5].

Volle POSIX-Kompatibilität notwendig: Der *POSIX*–Standard definiert (unter anderem) eine gemeinsame, standardisierte API, die von vielen (zumeist unixoiden) Betriebssystemen implementiert wird (siehe auch [31]). Nicht alle Teile dieses Interfaces können von `brig` umgesetzt werden. So gibt es kaum eine verträgliche Definition von harten und weichen Verlinkungen (*Hardlinks* und *Symbolic Links*) für dezentrale Netzwerke. Auch spezielle Dateien wie *FIFOs* können in diesem Kontext nicht ohne Race–Conditions umgesetzt werden. Entsprechende Operationen werden von *FUSE–Layer* mit dem *POSIX*–Fehlercode `ENOSYS` (»nicht implementiert«) quittiert.

Glaubhafte Abstreitbarkeit: Auch wenn ein `brig`–Repository in der geschlossenen Form als sicherer »Datensafe« einsetzbar ist, so bietet `brig` nicht die Eigenschaft der »glaubhaften Abstreitbarkeit«[6], die Werkzeuge wie Veracrypt bieten.

Zeilenbasierte Differenzen: Im Gegensatz zu Versionsverwaltungssystemen wie `git`, kann `brig` keine zeilenbasierten Differenzen zwischen zwei Dateien anzeigen, da es nur auf den Metadaten von Dateien arbeitet.

[1] Siehe auch: https://de.wikipedia.org/wiki/Eierlegende_Wollmilchsau
[2] Webpräsenz: http://ceph.com
[3] Webpräsenz: https://www.gluster.org
[4] In der momentanen Implementierung bei jedem `fsync()` und beim Schließen einer Datei.
[5] https://www.cockroachlabs.com
[6] Siehe auch: https://de.wikipedia.org/wiki/VeraCrypt

Reiner Speicherdienst auf der Gegenseite: Auf der Gegenseite muss ein `brig`–Daemon–Prozess laufen, um mit der Gegenseite zu kommunzieren. Daher können reine Speicherdienste wie *Amazon S3*[7] nicht ohne weiteres als Datenlager benutzt werden. Das kann allerdings leicht umgangen werden, indem der entfernte Speicher lokal gemounted[8] wird, und der `brig`–Prozess lokal gestartet wird. Werkzeuge wie `rsync` oder `git-annex` benötigen lediglich einen `ssh`–Zugang zum Datenlager und funktionieren daher auch ohne Gegenüber.

Keine starke Ausfallsicherheit: `brig` speichert nur ganze Dateien auf 1 bis n Knoten. Es wird kein *Erasure-Enconding*[9] angewendet, wie beispielsweise `Tahoe-LAFS` das tut. Damit eine Datei im Falle des Ausfalls eines Knotens wiederherstellbar ist, muss mindestens ein anderer Knoten, die Datei vollständig gespeichert haben, während andere Werkzeuge kleine Blöcke der Dateien redundant auf mehreren Rechnern ablegen. Werden diese beschädigt, können diese sich selbst reparieren oder von anderen Knoten neu übertragen werden. Für die meisten Anwendungszwecke ist aus Sicht des Autors Redundanz auf dem Dateilevel ausreichend.

Embedded Devices: `brig` benötigt ein vollständiges Betriebssystem mit Netzwerkanschluss, Hauptspeicher und einer ausreichend starken CPU. Die »unterste Grenze« für einen vernünftigen Betrieb wäre vermutlich ein aktueller Raspberry Pi in Version 3.

8.2. Erfüllung der Anforderungen

Im Folgenden wird die Umsetzung der in Kapitel 3 aufgelisteten Anforderungen betrachtet. Auf jede Anforderung wird dabei kurz zusammenfassend eingegangen und die Erfüllung wird mit »✓« (*Erfüllt*), »✎« (*Teilweise erfüllt*) und »✗« (*Überwiegend nicht erfüllt*) bewertet.

8.2.1. Anforderungen an die Integrität

Entkopplung von Metadaten und Daten (✓): Daten und Metadaten sind vollkommen entkoppelt und werden sowohl getrennt gespeichert (`ipfs` und *BoltDB*) als auch getrennt behandelt. Die Daten können irgendwo im `ipfs`–Netzwerk liegen, die Metadaten werden von allen Teilnehmern vorgehalten.

Pinning (✎): Es ist möglich, einen Pin zu Dateien und Verzeichnissen hinzuzufügen (`brig pin`) und wieder zu entfernen (`brig pin -u`). Allerdings wird dieses Konzept von `brig` selbst noch sehr simpel behandelt. Neu hinzugefügte Dateien bekommen automatisch einen Pin, die Pins eines Synchronisationspartners werden nicht übernommen. Der Pin von gelöschten Dateien wird entfernt. Es wird allerdings im momentanen Zustand weder eine Speicherquote eingehalten, noch wird der Pin automatisch ab einer bestimmten Versionierungstiefe entfernt.

Langlebigkeit (✎): Redundante Speicherung von Dateien ist manuell möglich, aber noch ist keine Anzahl minimaler Kopien einstellbar, die von `brig` überwacht wird. Eine Veränderung der Datei kann durch Neuberechnung der Prüfsumme überprüft werden.

[7] Mehr Informationen unter: https://de.wikipedia.org/wiki/Amazon_Web_Services/#Speicher
[8] Möglich mittels Werkzeugen wie `sshfs` (https://de.wikipedia.org/wiki/SSHFS) und `s3fs` (https://github.com/s3fs-fuse/s3fs-fuse)
[9] Eine Enkodierung, welche die Wiederherstellung der Inhalte bis zu einem gewissen, konfigurierbaren *Beschädigungsgrad* erlaubt. Siehe auch [32].

Verfügbarkeit (✎): Lokale Daten sind stets verfügbar. Daten von Synchronisationspartnern sind verfügbar wenn diese online sind. Ein Knoten der automatisch alle Metadaten von mehreren Partnern sammelt scheint technisch machbar (entsprechend dem Nutzer »`rabbithole@wonderland`« in Abschnitt 5.2.1), ist aber noch nicht implementiert. Problematisch für den Nutzer ist der Umgang mit momentan nicht verfügbaren Dateien. `brig` selbst hat keine Informationen darüber ob die Datei tatsächlich verfügbar ist, da diese Aufgabe von `ipfs` übernommen wird. Daher wird dies für den Benutzer erst ersichtlich wenn er versucht die Datei auszulesen. Sollte die Datei nicht verfügbar sein, so wird das Öffnen der Datei eine lange Zeit benötigen und schließlich mit einem Zeitüberschreitungsfehler enden. Hier müsste `brig` mehr Aufwand betreiben, um den Nutzer dabei zu helfen, nicht zugreifbare Dateien frühzeitig zu erkennen.

Integrität (✓): Jede Datei ist in Blöcke aufgeteilt, von denen jeder eine MAC speichert. Mithilfe dieser können absichtliche und unabsichtliche Modifikationen erkannt werden. Eine Integritätsprüfung für Metadaten (beispielsweise eine MAC, die den Store–Inhalt vor der Übertragung absichert) ist allerdings noch nicht implementiert.

8.2.2. Anforderungen an die Sicherheit

Verschlüsselte Speicherung: (✓) Jede Datei wird in einem verschlüsselten Container (siehe Abschnitt 5.4.1.1) abgelegt. Der Schlüssel wird momentan zufällig generiert und wird mit den anderen Metadaten zum Synchronisationspartner übertragen.

Verschlüsselte Übertragung: (✓) Nicht nur `ipfs`–Verbindungen an sich werden verschlüsselt, auch `brig's` Transferprotokoll (welches darauf aufsetzt) verschlüsselt die Daten zusätzlich, um sich gegen eventuelle Lücken in `ipfs` abzusichern.

Authentifizierung: (✓) Bevor eine Synchronisation stattfinden kann, müssen die Teilnehmer auf beiden Seiten ihr Gegenüber initial authentifizieren. Das geschieht indem sie den Nutzernamen mit der dazugehörigen Identitäts–Prüfsumme über einen sicheren Seitenkanal vergleichen. Da die Prüfsumme fälschungssicher ist, muss es sich um den gewünschten Partner handeln. Nach erfolgreicher initialer Authentifizierung wird der Partner in die Remote–Liste unter seinem Nutzernamen aufgenommen. Bei jedem erneuten Verbindungsaufbau zum Kommunikationspartner wird dieser basierend auf den Informationen in der Remote–Liste authentifiziert.

Identität: (✓) Als menschenlesbarer Identifikationsbezeichner wird eine modifizierte Form der Jabber–ID eingesetzt. Dieses Format ist gut lesbar und schränkt den Nutzer bei der Namenswahl nicht signifikant ein. Wie in Abschnitt 5.4.3 beschrieben, wird keine zentrale Instanz zur Registrierung benötigt.

Transparenz: (✓) Die Implementierung steht unter der freien `APGLv3`–Lizenz. Diese stellt rückwirkend die Freiheit des Quelltextes sicher. Zukünftige Versionen könnten prinzipiell proprietär werden, falls die Entwickler sich dazu entscheiden sollten. Auch wenn das nicht die aktuelle Absicht der Entwickler ist, könnte `brig` in diesem Fall von der Open–Source–Community weiterentwickelnd werden. Eine Einsicht in den Quelltext oder Beteiligung am Projekt ist durch die Code–Hosting–Plattform *GitHub* leicht möglich.

8.2.3. Anforderungen an die Usability

Automatische Versionierung: (✓) Eine Versionierung von Dateien ist gegeben, die vergleichbar mit `git` ist und umfangreicher als die, der meisten existierenden Werkzeuge. Momentan wird (hardkodiert) alle 15 Minuten ein automatisierter Commit gemacht (falls Änderungen vorlagen). Wie bereits oben beschrieben, wurde allerdings noch keine Quota implementiert, weshalb viele Änderungen an großen Dateien schnell sehr viel Speicherplatz benötigen werden.

Portabilität: (✎) Bei der Entwicklung wurde bei der Auswahl der Bibliotheken auf leichte Portierbarkeit zu anderen Systemen geachtet. Getestet und eingesetzt wurde die Software bisher nur auf einem Linux–System. Prinzipiell sollte sie auf anderen unixoiden Systemen lauffähig sein. Die volle Portierung auf Windows ist problematischer, da dort FUSE nicht lauffähig ist. Dabei gäbe es entweder die Möglichkeit eine Implementierung für das ähnliche, rein Windows–komaptible *Dokany*[10] zu liefern oder einen WebDAV[11] Server zu implementieren. Bei letzterer Option würde `brigd` als WebDAV–Server fungieren, der von Windows und anderen Betriebssystemen als Dateisystem eingehängt werden kann. Noch schwieriger wird der Einsatz von `brig` auf mobilen Plattformen. Dort ist *Go* momentan nur bedingt einsetzbar[12]. Sollte es einsetzbar werden, müsste eine eigene grafische Oberfläche implementiert werden, um `brig` beispielsweise auf einem Android–Smartphone nutzen zu können.

Einfache Installation: (✓) Auf unixoiden Betriebssystemen ist die Installation sehr einfach (siehe auch Anhang A.1). Die einzige Abhängigkeit von `brig` ist *Go*. Im späteren Projektverlauf können für die meistgenutzten Plattformen und Architekturen auch fertige Binärdateien angeboten werden.

Keine künstlichen Limitierungen: (✓) Durch den FUSE–Layer wird ein ganz normaler Systemordner bereitgestellt. Bis auf den lokalen Festplattenspeicher hat dieser keine zusätzlichen Limitierungen. Der einzige Unterschied für den Benutzer ist, dass die darin gespeicherten Daten entweder gar keinen Speicherplatz brauchen oder (durch das Verschlüsselungsformat) geringfügig größer sind als die eigentliche Datei. Durch die Versionierung benötigen zudem alte Kopien zusätzlichen Speicherplatz.

Generalität: (✎) `brig` ist mit denen im Punkt »Portabilität« genannten Einschränkungen auf allen Rechnern lauffähig und macht keine Annahmen zum Dateisystem oder zur Hardware auf der es läuft. Momentan sind allerdings alle Nutzer, mit denen synchronisiert werden soll, gezwungen `brig` zu nutzen. Dies betrifft auch Nutzer, mit denen nur eine einzelne Datei geteilt werden soll. Ein »HTTPS–Gateway«, mit dem einzelne Dateien veröffentlicht werden können, wurde noch nicht implementiert.

Stabilität: (✗) Die momentane Implementierung ist vergleichsweise instabil und bräuchte mehr Testfälle, um ein gewisses Vertrauen in die Stabilität der Software herzustellen. Welchen Umfang die Testsuite momentan hat, kann in Abschnitt 8.3 nachgelesen werden.

Effizienz: (✎) `brig` ist schnell genug, um auf einem typischen Arbeitsrechner eine lokale Full–HD Filmdatei vom FUSE–Dateisystem aus abzuspielen. Details zu der Geschwindigkeit finden sich

[10] https://github.com/dokan-dev/dokany, eine Go–Bibliothek ist bereits verfügbar: https://godoc.org/github.com/keybase/kbfs/dokan
[11] https://de.wikipedia.org/wiki/WebDAV
[12] Siehe dazu: https://golang.org/wiki/Mobile

in Abschnitt 8.4. Besonders im FUSE–Dateisystem sind noch einige Optimierungsmöglichkeiten vorhanden, welche die Gesamteffizienz steigern können.

8.3. Stand der Testsuite

Obwohl die Testsuite im momentanen Zustand zu klein ist, existieren für die meisten Pakete bereits Unittests. Insgesamt gibt es derzeit 26 Dateien, die Tests beinhalten, in denen sich 56 einzelne Unittests befinden. Diese versuchen immer möglichst kleine Teile der Codebasis anzusprechen, um die Fehlersuche zu erleichtern. So wird für viele Tests ein *Mock*–Store in einem temporären Verzeichnis angelegt oder es wird ein temporäres `ipfs`–Repository angelegt anstatt diese Arbeit den Quelltext hinter »`brig init`« erledigen zu lassen. Diese Methode hilft zusätzlich, um den Quelltext allgemein und austauschbar zu halten.

Noch existieren keine Zahlen, was die Testabdeckung angeht. Diese machen aus Sicht des Autors zum jetzigen Zeitpunkt auch noch keinen Sinn, da sich die Implementierung noch stark verändern wird. In Zukunft sollte die »Coverage« aber ein wichtiges Instrument werden, um nicht getesteten Quelltext aufzuspüren.

8.4. Benchmarks

Die Umsetzung von sauberen Benchmarks ist schwierig, da `brig` genau wie andere Synchronisationswerkzeuge ein sehr komplexes System ist, dessen Effizienz von einer Vielzahl von Faktoren abhängt. Grundsätzlich ist es schwierig, die Gesamteffizienz eines Systems sinnvoll zu messen, da folgende Komponenten sich von System zu System drastisch unterscheiden können:

- Hauptspeicher und Prozessor.
- Speichermedium und Netzwerklatenz.
- Betriebssystem und verwendeter Scheduler.
- Spezielle Befehlserweiterungssätze[13].

8.4.1. Aufbau

Aus diesem Grund wird im Folgenden rein die lokale Effizienz beim Hinzufügen einer Datei in Megabyte pro Sekunde untersucht. Alle Benchmarks werden direkt im Hauptspeicher ausgeführt, es erfolgen keine Festplattenzugriffe. Möglich wird das durch den Einsatz eines `ramfs`[14], welches ein temporäres Dateisystem bereitstellt, in dem alle Dateien direkt im Hauptspeicher geschrieben und gelesen werden. Dadurch ist der Prozessor[15] der ausschlaggebende Faktor bei der Effizienz in diesem Benchmark, da ausreichend Hauptspeicher vorhanden war.

Als Eingabedateien wurden zwei unterschiedliche Datensätze genommen:

- Eine schlecht komprimierbare Filmdatei[16].

[13] Beispielsweise SSE4.x: https://de.wikipedia.org/wiki/Streaming_SIMD_Extensions_4
[14] https://de.wikipedia.org/wiki/Ramfs
[15] In diesem Fall ein *AMD Phenom(tm) II X4 955*.
[16] Die 1080p Version von *Big Buck Bunny* von http://bbb3d.renderfarming.net/download.html

- Ein gut komprimierbarer Textkorpus in deutscher Sprache[17].

Aus beiden Dateien werden jeweils zehn kleinere Dateien durch Abschneiden hergestellt. Diese sind jeweils 1, 2, 4, 8, 16, 32, 64, 128, 256 und 512 MB groß und werden in das `ramfs` gelegt. Es entstehen also insgesamt 20 kleinere Dateien. Im `ramfs` wird ebenfalls ein `brig`–Repository und ein `ipfs`–Repository angelegt. Zudem wird im `ramfs` noch ein FUSE–Dateisystem, basierend auf dem `brig`–Repository, angelegt.

Basierend auf diesen Eingabedateien werden für beide Datensätze folgende Zeitmessungen erhoben:

- Lesen mit/ohne Dekompression und mit/ohne Entschlüsselung.
- Schreiben mit/ohne Kompression und mit/ohne Verschlüsselung.
- Lesen direkt von `ipfs` mit/ohne Entschlüsselung plus Dekompression.
- Schreiben direkt zu `ipfs` mit/ohne Verschlüsselung plus Kompression.
- Hinzufügen der Datei über `brig stage`.
- Ausgabe der Datei mit `brig cat` und über das FUSE–Dateisystem mit `cat`.

Als Grunddurchsatz (»Baseline«) wird zusätzlich gemessen wie lange ein direktes Kopieren der Datei im `ramfs` dauert. Zur Kompression wurde immer der *Snappy*–Algorithmus verwendet. *LZ4* zeigt ähnliche Eigenschaften, ist aber stets etwas langsamer. Zur Verschlüsselung wird *ChaCha20* eingesetzt. Das ebenfalls unterstützte *AES256* im GCM–Modus war ebenfalls immer etwas langsamer.

Die untenstehenden Ergebnisse wurden halbautomisch mit einem Shellskript erhoben und mit einem Python–Skript, mithilfe der `pygal`[18]–Bibliothek, in Plots gerendert. Beide Skripte finden sich in Anhang C.

8.4.2. Ergebnisse

Die Plots nutzen kubische Interpolation zwischen den einzelnen Messpunkten. Die Zeitachse ist zudem logarithmisch aufgetragen, um den linearen Zusammenhang zwischen den Achsen zu verdeutlichen. Insgesamt wurden vier Plots erstellt. Zwei für jede Eingabedatenmenge (Filmdatei und Textkorpus) und dafür jeweils ein Plot für das Schreiben und Lesen dieser Eingabedaten.

Insgesamt können folgende Konklusionen aus den Ergebnissen gezogen werden:

- Der Lese/Schreib–Durchsatz bleibt unabhängig von der Dateigröße größtenteils konstant.
- »`brig stage`« hat gegenüber »`ipfs add`« mit Verschlüsselung und Kompression einen geringen Overhead durch die interne Programmlogik. Dieser steigt aber bei einer größeren Datenmenge nicht weiter an (siehe Abb. 8.1 und Abb. 8.3).
- Ähnliches lässt sich zu »`brig cat`« und »`ipfs add`« feststellen (siehe Abb. 8.2 und Abb. 8.4).
- Kompression und Dekompression ist stets weniger ressourcenaufwändig als Verschlüsselung und Entschlüsselung. Ob an der Implementierung etwas verbessert werden kann, muss separat untersucht werden.
- Der Zugriff über das FUSE–Dateisystem (12.8 MB/s) ist verglichen mit `brig cat` (85 MB/s) deutlich langsamer. Die Gründe hierfür liegen vermutlich weniger an FUSE an sich, als an der aktuellen, ineffizienten Implementierung.

[17] http://corpora2.informatik.uni-leipzig.de/downloads/deu_news_2015_3M.tar.gz
[18] http://pygal.org/en/stable

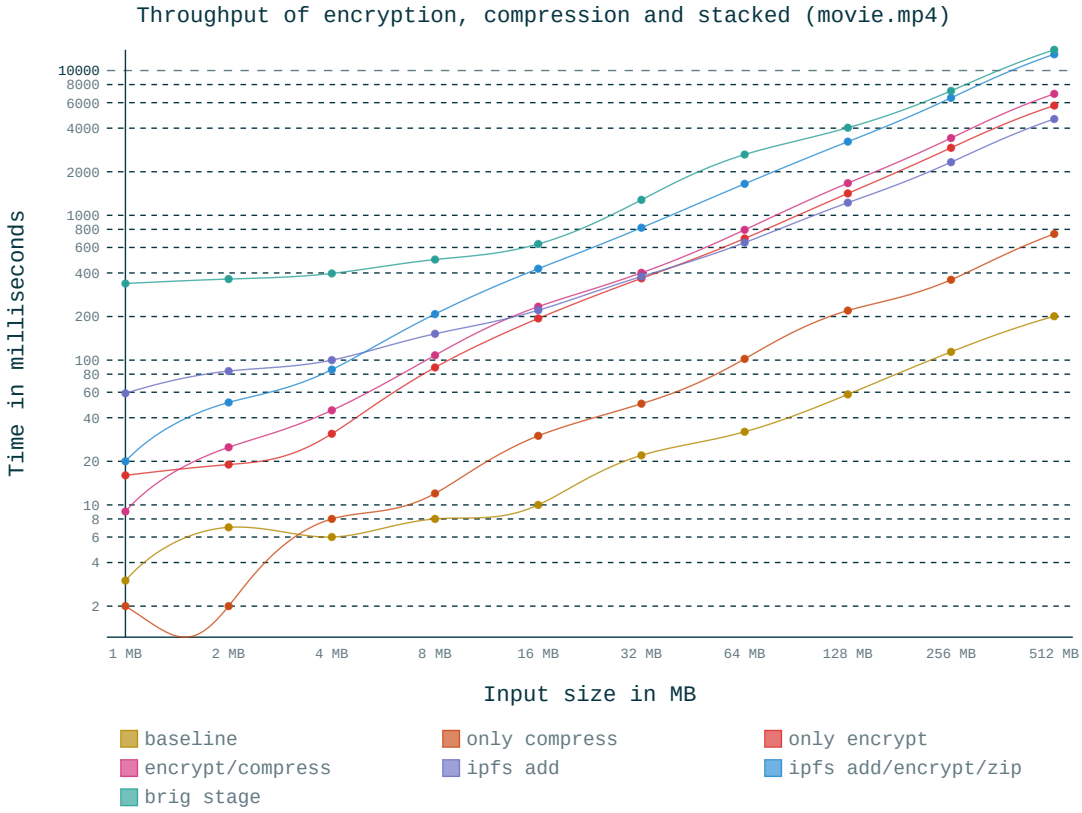


Abbildung 8.1.: Schreiboperationen auf der Datei `movie.mp4`

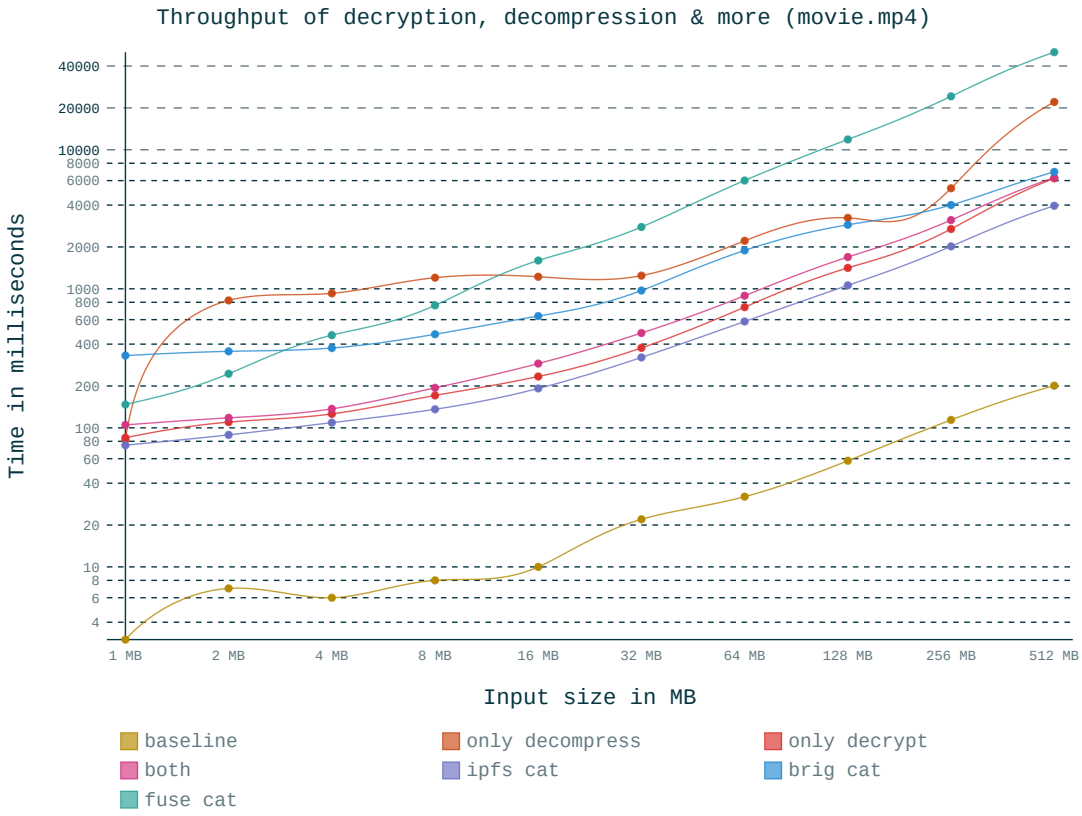


Abbildung 8.2.: Leseoperationen auf der Datei `movie.mp4`

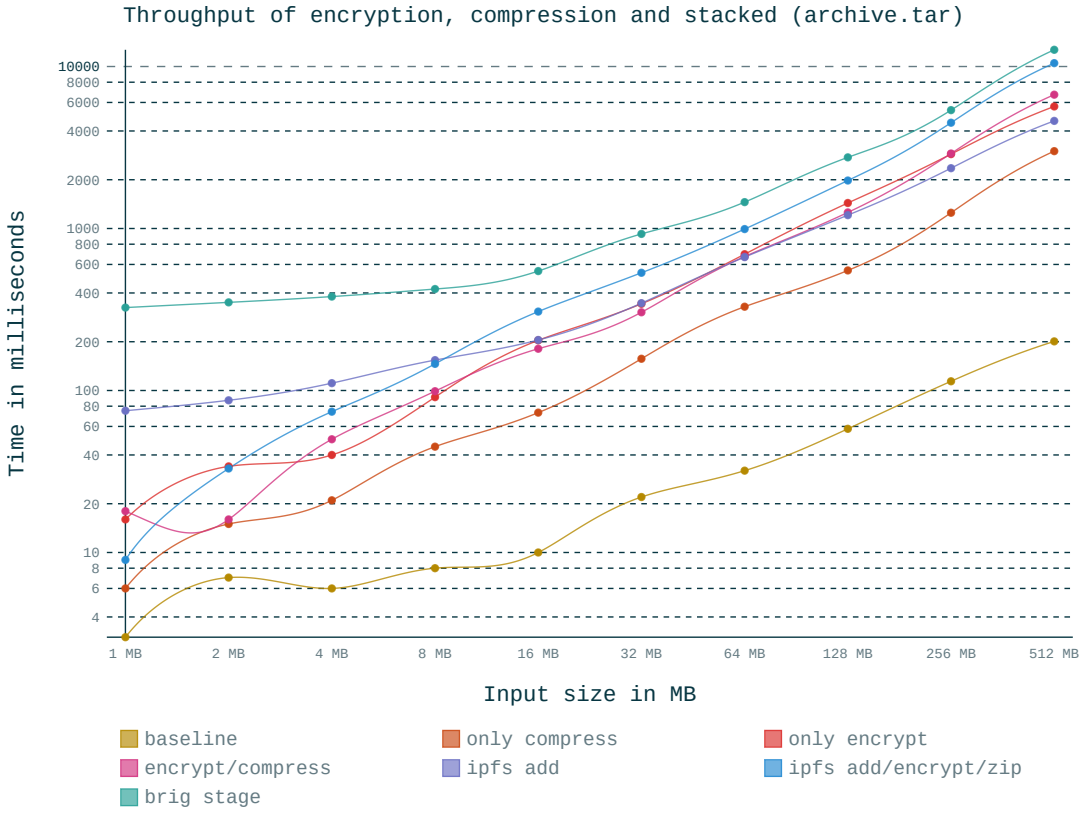


Abbildung 8.3.: Schreiboperationen auf der Datei `archive.tar`

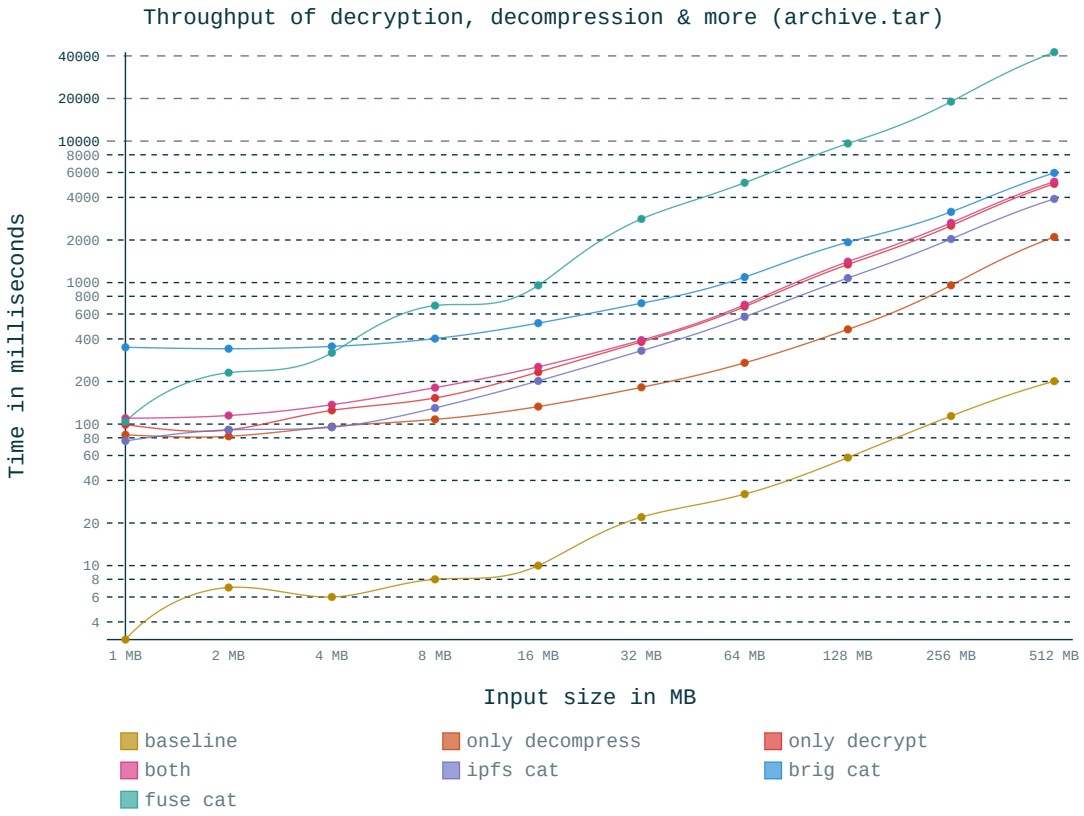


Abbildung 8.4.: Leseoperationen auf der Datei `archive.tar`

- Die Messergebnisse bis 8MB sind noch relativ stark von Messungenauigkeiten beeinträchtigt. Erst bei höheren Datenmengen werden die Ergebnisse repräsentativ.

Als Fazit lässt sich sagen, dass viel Optimierungspotenzial vorhanden ist, auch wenn die momentanen Durchsatzraten für viele Anwendungsfälle ausreichend sind. In vielen Szenarios werden zudem nicht die lokalen Dateioperationen der Flaschenhals sein, sondern die Übertragung über das Netzwerk.

8.5. Zukünftige Erweiterungen

Abschließend sollen noch einige mögliche Erweiterungsmöglichkeiten der momentanen Implementierung besprochen werden. Diese werden erst angegangen, sobald der momentane Prototyp stabilisiert, dokumentiert und veröffentlicht wurde. Die folgenden Ideen sind also noch in der Konzeptionsphase. Unterteilt wird die Auflistung in Verbesserungen an der existierenden Implementierung (welche vergleichsweise einfach umsetzbar sind), sowie konzeptuelle Erweiterungen (welche typischerweise weitgehendere Änderungen erfordern).

8.5.1. Verbesserungen an der Implementierung

Kompression basierend auf MIME–Type: Kompression lohnt sich nicht bei allen Dateiformaten. Gut komprimieren lassen sich Dateien mit Textinhalten (wie XML–Dateien) oder allgemein Daten mit sich wiederholenden Mustern darin. Schlecht komprimieren lassen sich hingegen bereits komprimierte Bilder, Videos und Archivdateien. Es wäre sinnvoll, basierend auf dem MIME–Type[33] einen geeigneten Kompressionsalgorithmus auszuwählen. Textdateien könnten so beispielsweise mit dem speicherplatzeffizienteren *LZ4* komprimiert werden, größere Dateien mit dem schnelleren *Snappy*. Multimediadateien könnten von der Kompression ausgenommen werden. Der MIME–Type kann dabei in vielen Fällen automatisiert durch das Lesen der ersten Bytes einer Datei erkannt werden. Entsprechender Code existiert bereits[19], wird aber noch nicht eingesetzt.

Integritätsprüfung: Wie in Kapitel 3 beschrieben, können Daten auf der Festplatte sich ohne Zutun des Nutzers verändern. Dieser Datenverlust ist nicht nur aus Sicht der fehlenden Information kritisch, sondern führt auch dazu, dass die Datei sich möglicherweise nicht mehr synchronisieren lässt, da bei einer Übertragung festgestellt werden würde, dass die Datei sich unerwartet verändert hat. An dieser Stelle könnte ein »`brig fsck`«–Kommando ansetzen, welches jede gespeichert Datei neu von `ipfs` liest und die Prüfsumme neu berechnet. Treten Diskrepanzen auf, so kann versucht werden, den fehlerhaften Block aus alten Versionsständen oder von einem Synchronisationspartner zu beziehen. Diese Funktionalität könnte auch direkt in `ipfs` eingebaut werden. Auch könnte eine solche Reparaturfunktion die BoltDB von `brig` auf Konsistenz prüfen und nötigenfalls und falls möglich reparieren. Im Gegensatz zu `git` sollten bei `brig` keine unerreichbaren Referenzen mehr im MDAG entstehen. Trotzdem könnte ein Programm wie »`brig gc`« einen *Garbage–Collector* laufen lassen, der solche Referenzen aufspüren und bereinigen kann. Diese würden auf Programmfehler hindeuten. Weiterhin könnte dieses Kommando genutzt werden, um `ipfs gc` zu starten.

Nutzung eines existierenden OpenPGP–Schlüssels: Momentan wird beim Anlegen eines Repositories ein neues RSA–Schlüsselpaar generiert. Viele Nutzer haben aber bereits ein Schlüsselpaar in

[19] https://github.com/disorganizer/brig/blob/master/store/mime-util/main.go

Form eines OpenPGP–Schlüsselpaars oder eines SSH–Schlüsselpaars. Diese könnten beim Anlegen des Repositories importiert werden. Sollte das Repository neu angelegt werden müssen, so kann der existierende Schlüssel in einem gängigen Format exportiert werden. Es muss allerdings darauf geachtet werden, dass keine zwei Repositories dasselbe Schlüsselpaar benutzen, da dies von `ipfs` nicht vorgesehen ist. Auch hier könnte die Funktionalität in `ipfs` direkt eingebaut werden.

Update Mechanismus: Sicherheitskritische Software wie `brig` sollte möglichst aktuell gehalten werden, um Sicherheitslücken schnell schließen zu können. Wie ein solcher Mechanismus im Detail aussehen könnte, zeigt die Arbeit von Herrn Piechula (siehe [1]).

HTTPS–Gateway: Wie in Abb. 3.1 gezeigt könnte `brig` als Webserver fungieren, der eine Schnittstelle zum »normalen« Internet bildet. Dieser könnte alle Dateien in einem speziellen Verzeichnis (beispielsweise mit dem Namen `/Public`) nach außen über einen Link zugreifbar machen. Dies hätte den Vorteil, dass bestimmte Dateien von anonymen Nutzern direkt zugegriffen werden können, ohne dass diese zu einem zentralen Dienst im Internet hochgeladen werden müssen. Voraussetzung dazu ist, dass ein Rechner von »außen« (also vom Internet) aus zugreifbar ist. Möglich wäre auch die Implementierung eines Passwortschutzes, um den Zugriff auf die Dateien zusätzlich abzusichern. Die Verbindung kann dabei durch HTTPS abgesichert werden. Dies benötigt auf Seite des Webservers ein gültiges TLS–Zertifikat. Mittlerweile gibt es dafür automatisierte Dienste wie *LetsEncrypt*[20]. Der in *Go* geschriebene Webserver `caddy`[21] beherrscht bereits das automatische Besorgen eines *LetsEncrypt*–Zertifikats.

Hooking Mechanismus: Um die Erweiterbarkeit von `brig` zu gewährleisten, könnte `brigd` seinen Clients Benachrichtigungen mitgeben, sofern sich diese dafür registrieren. Dieser würde den jeweiligen Client mitteilen wenn sich eine Datei geändert hat, gelöscht wurde oder Ähnliches. Auch Statistiken wie die aktuelle Speicherplatzauslastung könnten über diese Schnittstelle realisiert werden.

Packfiles: Mehrere Dateien könnten zu einem gemeinsamen, komprimierten *Pack* zusammengeschlossen werden, um Speicherplatz zu sparen. Für eine besonders effiziente Kompression können einzelne Versionen einer Datei zusammengepackt werden. Diese unterscheiden sich oft nur in einzelnen Blöcken und es wäre aus Sicht der Speichereffizienz ungünstig, diese redundant zu speichern. Wie in Abb. 8.5 gezeigt, können immer kleine Blöcke (beispielsweise 16 Kilobyte) von beispielsweise vier Dateien genommen werden und zu einem größeren Block (hier 64 Kilobyte) zusammengefasst werden. Diese großen Blöcke haben den Vorteil, dass sie viel redundante Informationen speichern, wenn sich dieser Block in den einzelnen Versionsständen nicht signifikant geändert hat. Kompressionsalgorithmen wie *Snappy* arbeiten auf 64 Kilobyte Blöcken[22], daher kann ein solcher Block relativ platzsparend komprimiert werden. Das Prinzip lässt sich auch auf mehr als die Versionen einer Datei erweitern. Mittels einer Heuristik können Dateien ausgewählt werden, die ähnlich groß sind und auch diese gemeinsam gepackt werden.

Bei kleinen Dateien (< 64 KB) ist bereits das Packen zu einer gemeinsamen Datei vorteilhaft, da diese mit weniger Overhead und effizienterer Kompression gepackt werden können. Die *Packfiles* von `git` nutzen einen anderen Ansatz, indem nur Deltas in den einzelnen Archiven gespeichert werden.

[20] https://letsencrypt.org
[21] https://caddyserver.com
[22] Siehe auch: https://github.com/google/snappy/blob/master/framing_format.txt#L91

Dies wäre bei Dateien möglicherweise auch für `brig` eine valide Herangehensweise und bleibt zu evaluieren.

Es wäre also möglich Speicherplatz im Tausch gegen Rechenzeit zu sparen, indem zwischen `ipfs` und `brig` noch eine Abstraktionsschicht eingebaut wird, die intelligent Dateien in *Packs* verpackt und zugreifbar macht. Die Implementierung dieses Konzeptes hätte zu viel Zeit in Anspruch genommen, weswegen hier weitere Arbeiten ansetzen könnten.

Atomarität und Transaktionen: In der momentanen Implementierung ist bei einem Ausfall von `brigd` (beispielsweise durch einen Absturz oder Stromausfall) nicht sichergestellt, dass eine Aktion (wie `MakeCommit()`) vollständig, atomar abgelaufen ist. Auch wenn die jeweilige Aktion von der API aus durch Locks atomar ist, wird im momentanen Zustand kein Rollback bei Fehlern ausgeführt. BoltDB an sich unterstützt atomare Transaktionen, aber durch die Abstraktion von der konkreten Datenbank werden mehrere kleine Transaktionen nicht zu einer zusammenhängenden, großen Transaktion zusammengefasst. Da aber die Datenbank austauschbar bleiben soll, muss von der Abstraktionsschicht eine Möglichkeit implementiert werden, zurückspulbare Transaktionen zu starten. Dazu dürfen Modifikationen an der Datenbank nicht direkt ausgeführt werden, sondern müssen in einem »Journal«[23] zusammengefasst werden. Dieses kann dann in einer einzigen, atomaren Datenbank–Transaktion zusammengefasst werden.

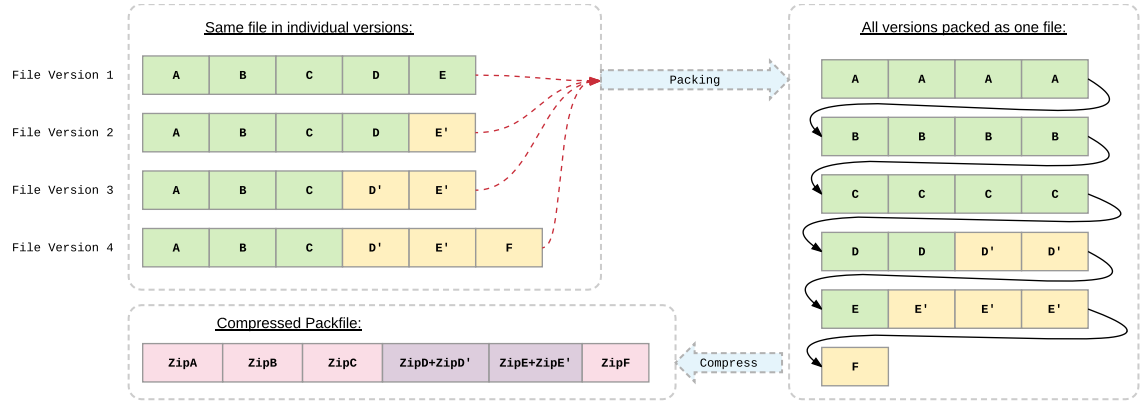


Abbildung 8.5.: Beispielhaftes Packen von vier einzelnen Versionständen.

8.5.2. Konzeptuelle Verbesserungen

Zugriffsrechte: `brig` unterscheidet im jetzigen Konzept nicht zwischen lesbaren, schreibbaren oder ausführbaren Dateien. Auch gibt es keinen Besitzer oder eine Gruppenzugehörigkeit der Datei. Die einzigen Dateiattribute bilden momentan die Größe und der letzte Änderungszeitpunkt. Aus diesem Grund bewirkt der Aufruf von »chmod« auf eine Datei im FUSE–Dateisystem nichts. Es muss nicht das System von Unix übernommen werden, allerdings wären die Informationen über den Eigentümer wichtig, um selektive Synchronisation zu implementieren. Dabei könnte eine Nutzergruppe angelegt werden. Nur die Nutzer, die dieser Gruppe angehören, können dann Dateien und Verzeichnisse einsehen, die auch dieser Gruppe zugeordnet sind.

Automatische Synchronisation: Änderungen müssen explizit synchronisiert werden. Um eine

[23] Siehe auch: https://de.wikipedia.org/wiki/Journaling-Dateisystem

Dropbox–ähnliche Funktionalität zu erreichen, sollte eine neue Option eingeführt werden: »`brig sync --auto bob@wonderland.lit`«. Dabei wird zuerst regulär mit *Bob* synchronisiert. Im Anschluss wird der Knoten von *Bob* angewiesen, *Alice* alle Änderungen auf seiner Seite sofort zu schicken. *Alice* empfängt diese und ändert ihre eigenen Dateien im Staging–Bereich, um die Änderung nachzuahmen.

Schlagwortbasiertes Dateisystem: `brig` arbeitet momentan rein als hierarchisches Dateisystem. Einzelne Knoten des MDAG werden also vom Nutzer mittels Pfad zugegriffen. Eine Erweiterung dazu könnte die Einführung eines schlagwortbasierten Ansatzes (ähnlich zu Tagsistant[24]) sein, welcher es möglich macht, die Menge aller Dateien semantisch durchsuchbar zu machen. Dateien und Verzeichnisse können vom Nutzer mit einem Schlagwort versehen werden (`brig tag <path> [<tag>...]`). Im FUSE–Dateisystem könnte das Konzept durch die Einführung eines speziellen Ordners (beispielsweise `/tags`) eingeführt werden. Dieser würde alle definierten Schlagworte als Ordner beinhalten. Unter jedem Schlagwortordner werden alle entsprechend verschlagworteten Dateien angezeigt.

Auto–Discovery anderer Nutzer: Momentan kann `brig` einen anderen Nutzer nur finden, wenn man seinen Nutzernamen kennt. Eine »unscharfe« Suche nach Benutzernamen wäre praktisch, ist aber aufgrund der dezentralen Natur von `brig` schwer umzusetzen. Machbar erscheint aber die automatische Erkennung von anderen `brig`–Nutzern »in der Nähe« (also im selben, lokalen Netzwerk). Für diesen Anwendungsfall würde sich das *Zeroconf-Protokoll*[25] eignen. Auch diese Funktionalität ließe sich eventuell direkt in `ipfs` integrieren.

In–Memory Laden des Repositores: Beim Starten von `brigd` werden einige sensible Dateien im Repository entschlüsselt. Solange `brigd` läuft bleiben diese auch lesbar und werden erst wieder verschlüsselt, wenn `brigd` sich beendet. Dies ist problematisch, wenn ein Angreifer in Besitz einzelner Dateien des Repositories kommen kann. Auch werden die Dateien nicht verschlüsselt, wenn `brigd` unvermittelt abstürzt und unsauber beendet wird. Schöner wäre eine reine Entschlüsselung der Daten im Hauptspeicher. Änderungen werden direkt in verschlüsselter Form wieder zurückgeschrieben.

Intelligenteres Key–Management: In der momentanen Implementierung werden alle Schlüssel in den Metadaten der Dateien gelagert. Die Metadaten werden als Ganzes zum Synchronisationspartner übertragen. Hier wäre entweder eine Trennung von den Metadaten (und damit gesonderte Übertragung) sinnvoll oder eine Schlüsselhierarchie, bei denen der eigentliche Schlüssel beispielsweise noch mit einem Gruppenschlüssel verschlüsselt wird. Siehe auch [1] für weitere Details.

Anonymisierung: Eine Anonymisierung des Datenverkehrs ist momentan nicht implementiert. Angreifer können zwar den Datenverkehr nicht mitlesen, doch können sie durchaus feststellen, welche Partner miteinander kommunizieren. In manchen Fällen kann dies bereits eine wichtige Information für einen Angreifer sein. Abhilfe könnte die Nutzung des Tor–Netzwerks[26] schaffen. Die einzelnen Knoten würden dabei nicht direkt miteinander kommunizieren, sondern schicken ihren Datenverkehr, verpackt in mehrere verschlüsselte Schichten, über mehre Knoten des Tor–Netzwerks. Diese Funktionalität muss direkt in `ipfs` implementiert werden. Entsprechende Überlegungen scheinen auf `ipfs`–Seite bereits zu existieren[27].

[24] https://en.wikipedia.org/wiki/Tagsistant
[25] https://de.wikipedia.org/wiki/Zeroconf
[26] Siehe auch: https://www.torproject.org
[27] Siehe auch: https://github.com/ipfs/notes/issues/37

9 Fazit

9.1. Zusammenfassung

Es wurde ein neuer, interdisziplinärer Ansatz für ein Dateisynchronisationssystem vorgestellt, der viele bestehende Ideen in einem stimmigen Konzept vereint. Eine funktionierende, quelloffene und für alle zugängliche Implementierung wurde vorgestellt und dokumentiert. Die anfangs gestellten Anforderungen konnte im Großen und Ganzen umgesetzt werden, auch wenn die Implementierung den Konzepten etwas nachsteht. Letztlich ist eine solide Basis für weitere Entwicklungen entstanden, die in absehbarer Zeit einem größeren Publikum präsentiert werden kann. Eine Abgrenzung zu anderen, existierenden Werkzeugen ergibt sich vor allem dadurch, dass die technischen Internas von `brig` vergleichsweise leicht verständlich sind und auch von fortgeschrittenen Nutzern verstanden werden können. Das Ziel, Usability für eine breite Masse zu bieten, konnte mangels grafischer Oberfläche noch nicht erreicht werden. Das Ziel eine sichere Basis zu schaffen konnte hingegen umgesetzt werden. Die Effizienz ist steigerungsfähig, sollte aber für viele Anwendungszwecke ausreichend sein.

9.2. Selbstkritik

Wie in Kapitel 8 diskutiert, ist noch Verbesserungspotenzial vorhanden. In Retrospektive hätte man sich stärker auf die Kernfunktionalität der Software und die zugrunde liegenden Konzepte konzentrieren müssen. Zusatzmodule wie Verschlüsselung sind wichtig, hätten aber auch zu späteren Zeitpunkten nachgerüstet werden können.

Es wurde viel Zeit darauf verwandt, Konzepte in Quelltext zu gießen, die letztlich keine Anwendung fanden oder nicht aufgingen. Das lässt sich bei großen Projekten kaum vermeiden, aber das Testen neuer Konzepte hätte auch mittels »unsauberer« Lösungen funktioniert. Hätte man die Software beispielsweise, nach Unix–Philosophie (wie `git`), als Sammlung kleiner Werkzeuge konzipiert, hätte man diese kurzzeitig mit einer Skriptsprache wie `bash` zusammenschließen können, um Probleme in den eigenen Ideen aufzudecken.

Obwohl der zeitliche Rahmen aufgrund der Suche nach Investoren, dem zeitgleichen Abschließen des Studiums und privaten Problem sehr eng war, ist mit `brig` eine erstaunlich flexible Idee entstanden, von der wir glauben, dass sie wirklich nützlich ist und die Welt etwas verbessern könnte.

9.3. Offene Fragen

Besonders fraglich ist wie gut das System in der Praxis funktioniert und auf größere Nutzermengen skaliert. Da `brig` von technisch versierten Nutzern entwickelt wurde, ist es auch fraglich wie gut verständlich es für neue, unerfahrenere Benutzer ist. Aus Sicht der Usability gibt es noch einige technische und konzeptuelle Probleme:

[1] Bildquelle: xkcd (https://xkcd.com/927)

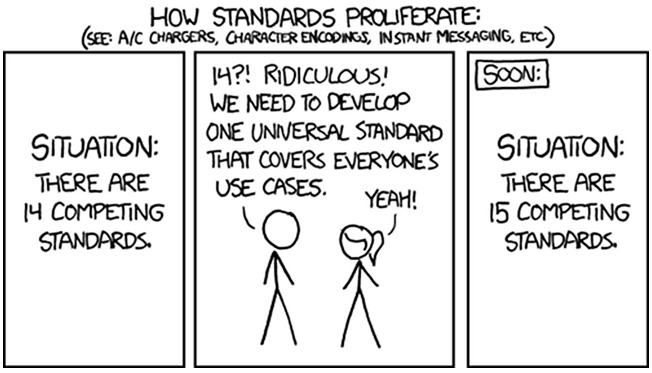


Abbildung 9.1.: Ist »brig« letztlich nur ein weiterer Standard?[1]

- Keine grafische Oberfläche, nur Kommandozeile.
- Initiale Authentifizierung von Hand nötig.
- Partner muss online sein, um mit ihm synchronisieren zu können.
- Noch keine automatische »Echtzeit«–Synchronisation.

Eine Veröffentlichung lohnt sich erst, wenn obige Punkte ansatzweise gelöst worden sind.

9.3.1. Beziehung zum `ipfs`–Projekt

Momentan wird `brig` vollkommen separat von `ipfs` entwickelt. Das hat vor allem den Grund, dass zu Anfang des Projektes die Richtung der Entwicklung noch nicht klar war. Die komplette Separation als eigenes Projekt, macht es deutlich einfacher mit verschiedenen Konzepten zu experimentieren. In Zukunft spricht jedoch nichts dagegen Teile von `brig`, sofern sie allgemein nützlich sind, auch dem `ipfs`–Projekt anzubieten und dort zu integrieren. Eine Zusammenarbeit wäre für beide Seiten vorteilhaft, da mehr Entwickler sich mit dem Quelltext befassen können und die dazugehörigen Konzepte aufeinander abstimmen können. Von `ipfs`–Seite scheint eine Zusammenarbeit gern gesehen zu sein:

> *Yeah we want to get to this too and would love to support your efforts. I'd request that you consider contributing directly to go-ipfs since much of what you want we want too.*
>
> — *Juan Benet*, Kernentwickler von `ipfs`[2]

Konkret wären folgende Module von `brig` für `ipfs` interessant:

- Das Verschlüsselungsformat.
- Das Kompressionsformat.
- Teile des Datenmodells, insbesondere die `Commit`–Struktur und Versionsverwaltung.
- Die Implementierung eines beschreibbaren FUSE–Dateisystems[3].

9.3.2. Zukunft der Autoren

Fraglich ist auch wie die Zukunft von `brig` aussieht, nachdem die vorliegende Arbeit abgeschlossen wurde. Leider konnte für die weitere Förderung des Projektes kein Sponsor verpflichtet werden. Trotz

[2] Quelle: https://github.com/ipfs/ipfs/issues/120
[3] `ipfs` implementiert momentan nur ein rein lesbares FUSE–Dateisystem.

Motivation der Autoren wird `brig` daher in der näheren Zukunft als Hobbyprojekt weitergeführt. Durch die private Situation beider Autoren wird die Entwicklung sich daher leider verlangsamen.

9.3.3. Veröffentlichung der Software

Nichtsdestotrotz ist es unser Ziel bis spätestens Mitte des Jahres 2017 `brig` auf einen Stand zu bringen, den man der Open–Source–Community präsentieren kann. Bevor es so weit ist, ist nicht nur Feinschliff an der bestehenden Software nötig, sondern es muss auch leicht zugängliche Dokumentation geschrieben werden und die Software für verschiedene Betriebssysteme gepackt werden.

Folgende Plattformen erscheinen uns für eine Präsentation der Software geeignet:

- Bei einem »Linux Tag« (Beispielsweise der »Linux Info Tag« in Augsburg[4]). Dort wäre eine detaillierte Präsentation vor Publikum mit direktem Feedback möglich.
- Kleineres Forum mit technisch versierten Nutzern; beispielsweise ein Forum für fortgeschrittene Linux–User. Dort könnte auch eine Paketierung der Software angesprochen werden.
- Größere Newsportale wie *reddit*. Diese werden von einem sehr breiten Publikum besucht.

Alle drei Möglichkeiten könnten auch zusammen in dieser Reihenfolge genutzt werden. Bei der Veröffentlichung sollte explizit angemerkt werden, dass sich Internas noch ändern können falls dazu Anlass bestehen sollte.

Auch wenn die Arbeit an `brig` persönlich sehr kräftezehrend war, haben wir eine Menge dabei gelernt. Es stecken eine Menge guter Ideen in der Software und aus unserer Sicht ist alleine die Zeit der limitierende Faktor, um `brig` zu einem Produkt zu machen, dass mehr als ein »Standard« (im Sinne von Abb. 9.1) unter Vielen ist.

[4] Siehe auch: http://www.luga.de/Aktionen/LIT-2016

A Anhang: Benutzerhandbuch

Die Funktionalität des `brig`–Prototypen ist im momentanen Zustand nur über eine Kommandozeilenanwendung erreichbar. Die Hilfe dieser Anwendung wird unten gezeigt. Im Folgenden werden die einzelnen zur Verfügung stehenden Optionen und Kommandos erklärt. Daneben wird auch eine Anleitung zur Installation gegeben und es werden Ratschläge zur optimalen Nutzung gegeben. Die Software ist zu diesem Zeitpunkt bei Weitem noch nicht stabil genug für den alltäglichen Einsatz. Es muss mit Abstürzen und Fehlern gerechnet werden.

```
NAME:
   brig - Secure and dezentralized file synchronization

USAGE:
   brig [global options] command [command options] [arguments...]

VERSION:
   v0.1.0-alpha+cd50f68 [buildtime: 2016-07-28T12:55:29+0000]

COMMANDS:
   ADVANCED COMMANDS:
     daemon Manually run the daemon process

   MISC COMMANDS:
     config Access, list and modify configuration values
     mount  Mount a brig repository

   REPOSITORY COMMANDS:
     init    Initialize an empty repository
     open    Open an encrypted repository
     close   Close an encrypted repository
     history Show the history of the given brig file
     pin     Pin a file locally to this machine
     net     Query and modify network status
     remote  Remote management.
     sync    Synchronize with other peers.

   VERSION CONTROL COMMANDS:
     status   Print which file are in the staging area
     diff     Show what changed between two commits
     log      Show all commits in a certain range
```

```
     commit   Print which file are in the staging area
     checkout Checkout an old version of a file or a commit

   WORKING COMMANDS:
     tree     List files in a tree
     ls       List files
     mkdir    Create an empty directory
     stage    Transer file into brig`s control
     unstage  Reset file to last known version
     rm       Remove the file and optionally old versions of it
     mv       Move a specific file
     cat      Concatenates a file

GLOBAL OPTIONS:
   --nodaemon, -n       Don`t run the daemon
   --password value,    Supply user password
   --path value         Path of the repository (default: ".") [$BRIG_PATH]
   --help, -h           show help
   --version, -v        print the version
```

A.1. Installation

`Brig` kann momentan nur aus den Quellen installiert werden. Zudem wurde der Prototyp nur auf Linux Systemen[1] getestet, sollte aber prinzipiell auch unter *Mac OS X* funktionieren. Die Installation aus den Quellen ist in beiden Fällen vergleichsweise einfach und besteht aus maximal zwei Schritten:

Installation von Go: Falls noch nicht geschehen, muss der *Go*–Compiler und die mitgelieferte Standardbibliothek installiert werden. Dazu kann in Linux Distribution der mitgelieferte Paketmanager genutzt werden. Unter Arch Linux ist der Befehl etwa »`pacman -S go`« unter Debian/Ubuntu »`apt-get install golang`«. In allen anderen Fällen kann ein Installationspaket von `golang.org`[2] heruntergeladen werden. Ist *Go* installiert, muss noch der Pfad definiert werden, in dem alle *Go*–Quellen landen. Dazu ist das Setzen der Umgebungsvariable `GOPATH` und eventuell auch `GOROOT` nötig:

```
$ mkdir ~/go
$ export GOPATH=~/go
$ export GOBIN=~/go/bin
$ export PATH=$PATH:~/go/bin
```

Die letzten drei `export` Kommandos sollte man in eine Datei wie `.bashrc` einfügen, um zu gewährleisten, dass die Umgebungsvariablen in jeder Sitzung erneut gesetzt werden.

Übersetzen der Quellen: Ist *Go* installiert, kann mittels des `go get`–Werkzeugs `brig` heruntergela-

[1] Im Falle der Autoren ist das: Arch Linux mit Kernel 4.4 und Go in Version 1.5 bis 1.6.
[2] https://golang.org/dl/

den und kompiliert werden:

```
$ go get github.com/disorganizer/brig
```

Nach erfolgreicher Ausführung (kann je nach Rechner zwischen etwa einer bis zehn Minuten dauern) sollte ein »brig«–Kommando auf der Kommandozeile verfügbar sein. Ohne weitere Argumente sollte das Kommando den oben stehenden Hilfetext produzieren.

A.1.1. Cross–Compiling

Sobald eine erste öffentliche Version von brig veröffentlicht wurde, sollen für die populärsten Plattformen vorgebaute Binärdateien angeboten werden. Um von einem einzigen Host–System aus Binärdateien für andere Plattformen zu erstellen, kann der *Go*–Compiler mittels der Umgebungsvariablen GOOS und GOARCH dafür konfiguriert werden. GOOS steuert dabei, die Zielplattform (z.B. *linux* oder *windows*), GOARCH hingegen steuert die Zielarchitektur der CPU (arm, 386, amd64). Folgendes Shellskript kann daher genutzt werden, um für einen Großteil der Plattformen jeweils eine Binärdatei zu erzeugen:

```
#!/bin/sh
PLATFORMS=( linux darwin windows )
ARCHS=( 386 amd64 arm )
OUTDIR=/tmp/brig-binaries

mkdir -p "$OUTDIR"
cd "$GOPATH/src/github.com/disorganizer/brig" || exit 1

for platform in "${PLATFORMS[@]}"; do
    for arch in "${ARCHS[@]}"; do
        printf "## Building %s-%s\n" $platform $arch
        export GOARCH=$arch; export GOOS=$platform

        # This calls `go install` with some extras:
        make || exit 2
        cp $GOBIN/brig "$OUTDIR/brig-$platform-arch"
    done
done
```

A.2. Grundlegende Benutzung

Die Bedienung von brig ist an das Versionsverwaltungssystem git angelehnt. Genau wie dieses, bietet brig für jede Unterfunktionalität ein einzelnes Subkommando an. Damit git–Nutzer die Bedienung leichter fällt, wurden viele Subkommandos ähnlich benannt, wenn sie in etwa dasselbe tun. So löschen sowohl git rm, als auch brig rm Dateien aus dem Repository.

A.2.1. Eingebaute Hilfe

Neben diesem Dokument und der eingebauten Hilfe gibt es im Moment keine weitere Dokumentation zu den vorgestellten Kommandos. Die eingebaute Hilfe kann entweder allgemein über `brig help` aufgerufen werden (produziert dieselbe Ausgabe, wie die eingangs gezeigte Hilfe) oder für ein spezifisches Subkommando mittels `brig help <subcommand>`. Beispiel für `brig help rm`:

```
NAME:
   brig rm - Remove the file and optionally old versions of it.

USAGE:
   brig rm [command options] <file> [--recursive|-r]

CATEGORY:
   WORKING COMMANDS

DESCRIPTION:
   Remove a spcific file or directory

OPTIONS:
   --recursive, -r  Remove directories recursively
```

A.2.2. Anlegen eines Repositories (`brig init`)

Alle von `brig` verwalteten Dateien werden in einem einzigen *Repository* verwaltet. Dies speichert alle Daten und die dazugehörigen Metadaten in einer Ordnerhierarchie. Um `brig` zu nutzen, muss daher zuerst ein Repository angelegt werden:

```
$ export BRIG_PATH=/tmp/alice
$ brig init alice@wonderland.lit/desktop
```

Der Nutzer wird um die Eingabe einer Passphrase gebeten. Die Formulierung *Passphrase* ist dabei bewusst anstatt dem Wort *Passwort* gewählt, da eine gewisse Mindestentropie Voraussetzung zur erfolgreichen Eingabe ist. Die Komplexität wird dabei von der `zxcvbn`–Bibliothek überprüft[3]. Welche Kriterien es dabei anwendet, kann in [1] nachgeschlagen werden.

Nach wiederholter, erfolgreicher Eingabe der Passphrase wird ein Schlüsselpaar generiert, und die in Abb. 6.5 gezeigte Verzeichnisstruktur angelegt.

A.2.3. Dateien hinzufügen, löschen und verschieben (`brig stage/rm/mv`)

Wurde ein Repository angelegt, können einzelne Dateien oder rekursiv ganze Verzeichnisse hinzugefügt werden:

```
$ cd $BRIG_PATH
$ brig stage ~/photos/cat.png
```

[3] Mehr Informationen hier: https://github.com/dropbox/zxcvbn

```
/cat.png
$ brig stage ~/music/knorkator/
/knorkator
```

Das Hinzufügen größerer Verzeichnisse nimmt etwas Zeit in Anspruch, da die Dateien jeweils komprimiert, verschlüsselt und eine Prüfsumme berechnet werden muss.

Anmerkung: Zum Ausführen dieser Kommandos muss man entweder im Ordner des `brig`-Repositories sein oder in einem Unterordner. Andernfalls wird `brig` eine Meldung wie diese ausgeben:

```
10.08.2016/17:33:11 I: Unable to find repo in path or any parents: "/home/sahib"
10.08.2016/17:33:11 W: Could not load config: open .brig/config:
                       No such file or directory
10.08.2016/17:33:11 W: Falling back on config defaults...
```

Oft genug reichen die Standardwerte der Konfiguration aus, damit der Befehl korrekt funktioniert. Alternativ kann auch die Umgebungsvariable `BRIG_PATH` wie oben gezeigt gesetzt werden, um von überall im Dateisystem das Kommando absetzen zu können.

Die hinzugefügten Dateien werden von `brig` einem virtuellen Wurzelknoten »/« hinzugefügt (`/cat.png`), anstatt den vollen Pfad zu erhalten (`~/photos/cat.png` — letzterer hätte nach der Synchronisation auf andere Rechner keine sinnvolle Bedeutung mehr. Dieses Prinzip wird auch ersichtlich bei Benutzung von `brig ls`:

```
$ brig ls
105 MB  4 seconds ago   /
2.1 MB  4 seconds ago   /photos/
2.1 MB  5 seconds ago   /photos/cat.png
103 MB  4 seconds ago   /knorkator/
 99 MB  4 seconds ago   /knorkator/hasenchartbreaker/
7.9 MB  1 minute ago    /knorkator/hasenchartbreaker/01 Ich bin ein ganz besondrer Mann.mp3
...
```

Möchte man den Inhalt einer Datei von `brig` wieder ausgeben lassen, so übergibt man den Pfad an das `cat`-Subkommando[4]:

```
$ brig cat /photos/cat.png > some-cat.png
$ open ./some-cat.png  # Öffnet die Datei in einem Bildbetrachter.
```

Da `brig cat` die Datei als kontinuierlichen Datenstrom ausgibt, ist es möglich größere Dateien wie Filme ohne Zwischendatei direkt anzuzeigen:

[4] Benannt nach dem traditionellen Unix-Kommando `cat` zum Ausgeben und Konkatenieren von Dateien.

```
$ brig cat /movies/big-buck-bunny.mov | mpv -  # Zeige Film mit `mpv`
```

Auch die üblichen Unix–Kommandos zum Anlegen von Verzeichnissen, sowie dem Löschen und Verschieben von Dateien sind verfügbar:

```
# Anmerkung: Der vordere '/' kann auch nach Belieben weggelassen werden.
$ brig mkdir seen-movies
$ brig mv movies/big-buck-bunny.mov seen-movies/
$ brig rm seen-movies/big-buck-bunny.mov
$ brig tree
```

A.2.4. Nutzung des FUSE–Dateisystems (`brig mount/unmount`)

Die bisherige Nutzung von `brig` erinnert an `git` und ist für alltägliche Aufgaben eher aufwendig und nicht kompatibel mit existierenden Dateimanagern. Leichter wäre es für den Benutzer wenn er seine gewohnten Anwendungen einfach weiterverwenden könnte. Das ist mit dem *FUSE*–Dateisystem möglich. Zur Verwendung muss das Dateisystem »gemounted« werden:

```
$ mkdir /tmp/alice-mount
$ brig mount /tmp/alice-mount
```

Dies erstellt in `/tmp/alice-mount` einen speziellen Ordner, mit den bisher hinzugefügten Dateien:

```
$ ls /tmp/alice-mount
photos  movies  knorkator
```

Es können wie gewohnt Dateien editiert werden, gelöscht und neu angelegt werden:

```
$ gimp /tmp/alice-mount/photos/cat.png
$ cp ~/dog.png /tmp/alice-mount/photos
$ rm /tmp/alice/photos/dog.png
```

Das Erstellen mehrere Mounts an verschiedenen Pfaden ist möglich. Eine Modifikation in dem einen Ordner wird stets auch im anderen Ordner angezeigt.

A.2.5. Versionsverwaltung (`brig status/commit/log/checkout`)

Alle genanten Operationen werden von `brig` im Hintergrund aufgezeichnet und versioniert. Dabei muss zwischen *Checkpoints* und *Commits* unterschieden werden. Erstere beschreiben eine atomare Änderung an einer Datei (also ob sie hinzugefügt, gelöscht, modifiziert oder verschoben wurde). Ein *Commit* fasst mehrere *Checkpoints* zu einem gemeinsamen, logischen Paket zusammen. Ähnlich wie bei `git`, gibt es zudem einen *Staging*–Bereich, der aus den *Checkpoints* besteht, die noch in keinem *Commit* verpackt worden sind. Ein wichtiger Unterschied zu `git` ist allerdings, dass `brig` auch automatisiert (in einem konfigurierten Zeitintevall) *Commits* erstellen kann. Diese dienen dann eher als Sicherungspunkte eines Repositories, beziehungsweise *Snapshots* wie in vielen Backup–Programmen und weniger als zusammenhängende Einheit logischer Änderungen.

```
$ brig status
Changes by alice@wonderland.lit/desktop:

  Added:
        photos/kitten.png
  Removed:
        photos/dog.png
  Moved:
        cat.png -> photos/cat.png
```

Die gemachten Änderungen können mit dem `commit`-Unterkommando in einem *Commit* verpackt werden:

```
$ brig commit -m 'Moved my cat photos to the right place.'
3 changes committed
```

Die Nachricht, die man mittels `-m (--message)` angegeben hat beschreibt, was in diesem *Commit* passiert ist und taucht später als hilfreiche Beschreibung im `log` auf. Man kann diese Nachricht auch weglassen, was `brig` dazu veranlasst eine automatische *Commit*-Nachricht zu verfassen:

```
$ brig stage ~/garfield-small.png /photos/garfield.png
$ brig commit
1 change committed
```

Die gemachten *Commits* lassen sich mittels des `log`-Unterkommandos anzeigen:

```
# Zeige alle gemachten Commits an (Prüfsummen wegen Überlänge gekürzt)
$ brig log
QmNLei78zW by alice, Initial commit
QmPtprCMpd by alice, Moved cat photos to the right place.
QmZNJPSbTE by alice, Update on 2016-08-11 15:33:37.651 +0200 CEST
```

Die *Checkpoints* einer einzelnen Datei zeigt der `history`-Befehl:

```
$ brig history photos/cat.png
/photos/cat.png
 +-- Checkpoint #2 (moved by alice@jabber.nullcat.de/laptop)
 |   +- Hash: Qma2Uquo9bMyuRZ7Fw1oQ1v68Vm7hpCYLRsrQXoLFpZVoK
 |   +- What: /cat.png -> /photos/cat.png
 |   \_ Date: 2016-08-11 15:24:39.993907482 +0200 CEST
 \-- Checkpoint #1 (added by alice@jabber.nullcat.de/laptop)
     |- Hash: Qma2Uquo9bMyuRZ7Fw1oQ1v68Vm7hpCYLRsrQXoLFpZVoK
     \_ Date: 2016-08-11 15:24:15.301565687 +0200 CEST
```

Hat man Änderungen an einer Datei gemacht und möchte diese wieder auf den letzten sauberen Stand zurücksetzen, so kann man den Befehl `stage`-Befehl benutzen:

```
# Setze /photos/cat.png auf den letzten Stand im staging commit zurück.
$ brig unstage /photos/cat.png
```

Möchte man tiefer in die Vergangenheit zurück springen, so kann der checkout–Befehl genutzt werden. Dieser stellt entweder einen früheren Dateibaum wieder her oder setzt ein Datei oder Verzeichnis auf eine frühere Version zurück:

```
# Setze den Stand auf den Commit QmNLei78zW zurück:
$ brig checkout QmNLei78zW
# Setze nur eine einzelne Datei auf den Stand in Commit QmNLei78zW zurück:
$ brig checkout QmNLei78zW -- /photos/cat.png
```

A.2.6. Verwalten von Synchronisationspartnern (`brig remote`)

Um seine Dateien mit anderen Teilnehmern zu teilen, müssen diese erst einmal `brig` bekannt gemacht werden und vom Nutzer authentifiziert werden. Für diese Aufgabe bietet `brig` das `remote`–Unterkommando. Jedes Repository hat dabei eine eindeutige »Identität«, welches es im Netzwerk eindeutig identifiziert. Diese besteht aus einer Prüfsumme, und einem menschenlesbaren Nutzernamen. Für das eigene Repository kann er folgendermaßen angezeigt werden:

```
$ brig remote self
QmZyhL3VAAr35a9msSyhW4zfLPnx9Jn4gMSyMQR5VCBFnx online alice@wonderland.lit/desktop
```

Das Hinzufügen eines anderen Nutzers erfordert beide Werte: Sowohl sein Nutzername, als auch die kryptografische Prüfsumme, der ihn eindeutig identifiziert. Kennt man den Namen seines Kommunikationspartners, so kann `brig` alle Teilnehmer im Netzwerk mit diesen Namen abfragen. Im Beispiel möchte `alice` nun auch ein `brig`–Repository auf ihren Laptop einrichten und auf ihren Arbeitsrechner dieses als Partner eintragen:

```
$ brig remote locate alice@wonderland.lit/laptop
QmVszFHVNj6UYuPybU3rVXG5L6Jm6TVcvHi2ucDaAubfss
QmNwr8kJrnQdjwupCDLs2Fv8JknjWD7esrF81QDKT2Q2g6
```

Für gewöhnlich taucht hier nur eine Prüfsumme auf, in diesem Fall muss zwischen zwei verschiedenen Identitäten gewählt werden. Mindestens eine davon könnte theoretisch ein Betrüger sein, der nur den Nutzernamen *alice@wonderland.lit/laptop* verwendet. In diesem Fall ist es nötig über einen Seitenkanal direkt Kontakt mit der Person aufzunehmen, mit der man synchronisieren will und darüber die Identität abzugleichen. Ein möglicher Seitenkanal wäre ein Telefonanruf, E–Mail oder auch ein Instant–Messanger. Hat man festgestellt was die richtige Identität ist, kann man sie seiner Kontaktliste hinzufügen:

```
$ brig remote add alice@wonderland.lit/laptop \
        QmVszFHVNj6UYuPybU3rVXG5L6Jm6TVcvHi2ucDaAubfss
```

Falls man nur den Teil hinter dem »@« kennt (also die *Domain*), so können auch alle Identitäten mit dieser Domain aufgelistet werden:

```
$ brig remote locate -d wonderland.lit
QmZyhL3VAAr35a9msSyhW4zfLPnx9Jn4gMSyMQR5VCBFnx
QmVszFHVNj6UYuPybU3rVXG5L6Jm6TVcvHi2ucDaAubfss
QmNwr8kJrnQdjwupCDLs2Fv8JknjWD7esrF81QDKT2Q2g6
```

Das Unterkommando »`brig remote list`« zeigt alle verfügbaren Kontakte an und ob diese online sind:

```
$ brig remote list
QmZyhL3VAAr35a9msSyhW4zfLPnx9Jn4gMSyMQR5VCBFnx online alice@wonderland.lit/laptop
```

Das Löschen eines Kontakts ist mit `brig remote remove <username>` möglich und wird nicht weiter demonstriert.

A.2.7. Synchronisieren (`brig sync`)

Anmerkung zur `git` Analogie: Es ist bei `brig` nicht nötig eine gemeinsame Synchronisations–Vergangenheit zu haben. Es wird rein auf Dateiebene synchronisiert. Mit anderen Worten: Konflikte entstehen nur dann wenn mehre Teilnehmern unterschiedliche Checkpoints für einen einzelnen Pfad einbringen. Sollten trotzdem Konflikte auftreten, wird für jede Konfliktdatei eine weitere Datei gespeichert, die mit dem Suffix `.<owner>.conflict` versehen wird. Hat also beispielsweise *Alice* und *Bob* eine Datei namens `/photos/cat.png` und beide haben sie modifiziert, so wird eine Synchronisation mit Alice dazu führen, dass Bob seine eigene Version `/photos/cat.png` behält, aber eine weitere Datei namens `/photos/cat.png.bob.conflict` erhält.

```
$ brig sync alice@wonderland.lit/laptop
No conflicts.
$ brig sync bob
Conflict: /photos/cat.png -> /photos/cat.png.bob.conflict
```

A.2.8. Dateien pinnen (`brig pin`)

Ist man beispielsweise mit dem Zug unterwegs, so kann ein Pfad »gepinnt« werden, um sicherzustellen dass er lokal verfügbar ist:

```
$ brig pin /movies/swiss-army-man.mkv
```

Benötigt man später wieder den Speicherplatz, so kann die Datei wieder »unpinned« werden. `brig` wird diese Datei nach einiger Zeit aus dem lokalen Zwischenspeicher entfernen, sofern ein Platzmangel vorherrscht:

```
$ brig unpin /movies/swiss-army-man.mkv
```

A.2.9. Konfiguration (`brig config`)

`brig` bietet momentan wenige Optionen, um das Verhalten der Software nach seinen Wünschen einzustellen. Ein Überblick über die verfügbaren Optionen liefert das Unterkommando `brig config list`:

```
$ brig config list
daemon:
  port: 6666                        # Der Port von brigd.
ipfs:
  path: /tmp/alice/.brig/ipfs       # Pfad zum IPFS-Store
  swarmport: 4001                   # Port des IPFS Swarm
repository:
  id: alice@wonderland.lit/desktop  # Nutzer-ID
```

Das verwendete Format zur Speicherung und Anzeige entspricht dem YAML–Format. Einzelne Werte können auch direkt angezeigt werden:

```
$ brig config get repository.id
alice@wonderland.lit/desktop
```

Möchte man die Werte editieren, so können diese einzeln gesetzt werden:

```
$ brig config set daemon.port 7777
```

A.3. Fortgeschrittene Nutzung

Die obigen Befehle reichen für die alltägliche Benutzung von `brig` aus. Es gibt einige weitere Befehle, die besonders für technisch versierte Nutzer und Entwickler interessant sind.

A.3.1. Repository öffnen und schließen (`brig open/close`)

Um ein Repository als *Datentresor* zu nutzen, kann mit dem `close`–Unterkommando der `brigd`–Daemon heruntergefahren werden. Danach ist das Repository nur mit der erneuten Eingabe eines Passwortes zugreifbar. Das kann nützlich sein, um Fremdzugriff auch bei physikalischer Abwesenheit am Rechner zu verhindern.

```
$ brig close
# ...Nach einiger Zeit ohne Internetzugang:
$ brig open
Password: **********
```

Ein explizites `brig open` ist bei normaler Benutzung nicht nötig. Jedes Kommando, das von `brigd` abhängt, versucht diesen zu starten, wenn der Daemon nicht erreichbar ist. Dazu fragt es wie `brig open` auch nach dem Passphrase. Das `open`–Unterkommando ist allerdings nützlich für Skriptdateien, wenn der Passwort–Prompt an einer erwarteten Stelle auftauchen soll.

A.3.2. Status von `brigd` (`brig daemon`)

Das `daemon`–Unterkommando bietet einige Optionen, um den Status von `brigd` zu überprüfen und zu verändern. Um zu überprüfen ob `brigd` läuft, kann das `ping`–Unterkommando genutzt werden:

```
$ brig daemon ping
#01 127.0.0.1:33024 => 127.0.0.1:6668: OK (517.310422ms)
#02 127.0.0.1:33024 => 127.0.0.1:6668: OK (522.751µs)
...
```

Das `wait`–Unterkommando wartet bis `brigd` verfügbar ist und Kommandos entgegen nehmen kann. Das ist für Skripte nützlich, die darauf warten müssen ohne Passwort–Prompt normale `brig`–Kommandos abzusetzen:

```
$ echo 'Waiting for brig to start...'
$ brig daemon wait
$ echo 'Available! You can execute brig commands now.'
```

Auch das Starten und Beenden von `brigd` ist mit diesem Unterkommando direkt möglich:

```
$ brig daemon quit    # Momentan selbe Funktion wie `brig close`
$ brig daemon launch  # Momentan selbe Funktion wie `brig open`
```

A.3.3. Netzwerkstatus (`brig net`)

Das `net`–Unterkommando bietet die Möglichkeit sich vom Netzwerk zu trennen und wieder zu verbinden:

```
$ brig net status
true
$ brig net offline
$ brig net status
false
$ brig net online
true
```

A.3.4. Debugging (`brig debug`)

Unter dem `debug`–Unterkommando finden sich einige Hilfsmittel, um die internen Abläufe von `brig` nachvollziehen zu können:

- `brig debug export`: Exportiert den aktuellen Metadatenindex auf `stdout`.
- `brig debug import`: Importiert die serialisierte Version eines Metadatenindex.

A.3.5. Software–Version anzeigen (`brig version`)

Die Versionsnummer von `brig` folgt den Prinzipien des *Semantic Versioning*[5] (in der Version 2.0). Das Format entspricht dabei »v<MAJOR>.<MINOR>.<PATCH>[-<TAG>][+<REV>]«, wobei die Platzhalter folgende Bedeutung haben:

- `MAJOR`: Oberste Versionsnummer. Wird nur bei inkompatiblen Änderungen inkrementiert.
- `MINOR`: Wird bei Erweiterungen inkrementiert, welche nicht die Kompatibilität beeinflussen.

[5] Mehr Informationen unter http://semver.org

- PATCH: Wird bei Berichtigung einzelner Fehler jeweils einmal inkrementiert.
- TAG: Optional. Weist spezielle Entwicklungsstände wie `alpha`, `beta`, `final` etc. aus.
- REV: Optional. Falls bei Kompilierzeit verfügbar, der aktuelle `git`–HEAD.

Nach der eigentlichen Versionsnummer wird zusätzlich zur Information der Kompilierzeitpunkt angezeigt:

```
$ brig -v
v0.1.0-alpha+cd50f68 [buildtime: 2016-07-28T12:55:29+0000]
```

B Anhang: Protokolldefinitionen

B.1. Internes Datenmodell von `brig`

```
syntax = "proto3";
package brig.store;
option go_package = "wire";

////////////// VERSION CONTROL STRUCTURES ///////////

message Author {
  string name = 1;
  string hash = 2;
}

// Optional merge information for merge commits
message Merge {
    string with = 1;
    bytes hash = 2;
}

message Checkpoint {
  // Link to the node id:
  uint64 id_link = 1;
  bytes hash = 2;
  uint64 index = 3;
  int32 change = 4;
  string author = 5;
}

// History is the history of a file:
message History {
    repeated Checkpoint hist = 1 [packed=false];
}

message CheckpointLink {
  uint64 id_link = 1;
  uint64 index = 2;
}
```

```
// Commits is an ordered list of commits
message Commits {
    repeated Commit commits = 1;
}

// Ref is a pointer to a single commit
message Ref {
    string name = 1;
    bytes hash = 2;
}

////////////// NODE BASICS ////////////

// Might be extended with more esoteric types in the future.
enum NodeType {
  UNKNOWN = 0;
  FILE = 1;
  DIRECTORY = 2;
  COMMIT = 3;
}

// An Object is a container for a file, a directory or a Ref.
message Node {
  // Type of this node (see above)
  NodeType type = 1;
  // Global identifier of this node, since hash and path
  // might change sometimes.
  uint64 ID = 2;

  // Size of the node in bytes:
  uint64 node_size = 3;

  // Timestamp formated as RFC 3339
  bytes mod_time = 4;

  // Hash of the node as multihash:
  bytes hash = 5;

  // Name of this node (i.e. path element)
  string name = 6;

  // Path must only be filled when exported to a client.
```

```
  // It may not be used internally and is not saved to the kv-store.
  string path = 7;

  // Individual types:
  File file = 8;
  Directory directory = 9;
  Commit commit = 10;
}

// Just a collection of nodes:
message Nodes {
  repeated Node nodes = 1;
}

////////////// CONCRETE NODES ////////////

message File {
  // Path to parent directory
  string parent = 1;

  // Key of this file:
  bytes key = 2;
}

message Directory {
  // Path to parent object:
  string parent = 1;

  // Directory contents (hashtable contents [name => link]):
  repeated bytes links = 2;
  repeated string names = 3;
}

// Commit is a bag of changes, either automatically done or by the user.
message Commit {
  // Hash of the parent commit:
  bytes parent = 1;

  // Commit message:
  string message = 2;

  // Author of this commit:
```

```
  Author author = 3;

  // Hash to the root tree:
  bytes root = 4;

  // List of checkpoints (one per file):
  repeated CheckpointLink changeset = 5;

  // Merge information if this is a merge commit.
  Merge merge = 6;

  // Checkpoints stored in the commit.
  // This is only used when exported to the client,
  // it is not stored in the kv-store.
  repeated Checkpoint checkpoints = 7;
}

////////////// EXPORT/IMPORT DATA //////////////

// Store is the exported form of a store.
message Store {
  // The boltdb format.
  bytes boltdb = 1;
}
```

B.2. Protokoll zwischen zwei Knoten

```
syntax = "proto3";
package brig.transfer;
option go_package = "wire";

import "store.proto";

enum RequestType {
    INVALID = 0;
    FETCH = 1;
    STORE_VERSION = 2;
    UPDATE_FILE = 3;
}

message Request {
    RequestType req_type = 1;
```

```
    int64 ID = 2;
    int64 nonce = 3;
}

message StoreVersionResponse {
    int32 version = 1;
}

message FetchResponse {
    brig.store.Store store = 1;
}

message Response {
    RequestType req_type = 1;
    int64 ID = 2;
    int64 nonce = 3;
    string error = 4;

    StoreVersionResponse store_version_resp = 5;
    FetchResponse fetch_resp = 6;
}
```

C Anhang: Benchmark–Skripte

C.1. benchmark.sh

```
#!/bin/sh

rm data/ipfs -rf
export IPFS_PATH=./data/ipfs
export BRIG_PATH=./data/brig
ipfs init


function time_it() {
    ts=$(date +%s%N)
    $*
    tt=$((($(date +%s%N) - $ts)/1000000))
    >&2 echo "Time taken: $tt"

}

function create_ramfs() {
    mkdir -p data
    sudo mount -t ramfs -o size=2G ramfs data
    sudo chmod 0777 data
    sudo chown -R sahib:users data
}

function copy_sized() {
    echo "Copying $3 MB of $1 to $2"
    dd if="$1" of="$2" bs=1M count="$3" status=none
}

##### write functions:

function compress_to_ipfs() {
    ./main -f -a $2 $3 -s -c $1 | ipfs add -q > /dev/null
}

function compress_single() {
```

```
    ./main -f -D -a $2 $3 -c $1
}

function add_to_ipfs() {
    ipfs add -q $1
}

function baseline_write() {
    time_it cat $1 > /dev/null
}

##### read functions:

function compress_to_ipfs_and_read() {
    HASH=$(./main -f -a $2 $3 -s -c $1 | ipfs add -q)
    ipfs cat $HASH | time_it ./main -f -a $2 $3 -d $1.$2
}

function compress_single_and_read() {
    ./main -f -a $2 $3 -c $1
    time_it ./main -f -a $2 $3 -d $1.$2
}

function add_to_ipfs_and_read() {
    HASH=`ipfs add -q $1`
    time_it ipfs cat $HASH > /dev/null
}

function baseline_read() {
    time_it dd if=$1 of=/dev/null bs=4M status=none
}

# Init:
create_ramfs
size=1
for i in `seq 10`; do
    copy_sized "input/movie.mp4" "data/movie_$size" $size
    copy_sized "input/archive.tar" "data/archive_$size" $size
    size=$(expr $size \* 2)
done

function sample() {
```

```
    echo "=== $*"
    local size=1
    for i in `seq 10`; do
        $1 "data/archive_$size" $2 $3 $4 $5 $6
        size=$(expr $size \* 2)
    done

    rm data/*.$2 -f
}

sample baseline_read
sample add_to_ipfs
sample compress_single_and_read snappy
sample compress_single_and_read none -e
sample compress_single_and_read snappy -e
sample compress_to_ipfs snappy
sample compress_to_ipfs snappy -e

function create_brig_repo() {
    pkill -9 brig
    rm data/brig -rf
    brig -x ThiuJ9wesh --nodaemon init alice@jabber.nullcat.de/laptop
    brig -x ThiuJ9wesh daemon launch 2>/dev/null &
    echo "...waiting for daemon to catch up..."
    sleep 5
    fusermount -u data/mount
    mkdir -p data/mount
    brig mount data/mount
}


create_brig_repo
size=1
for i in `seq 10`; do
    time_it brig stage "data/archive_$size"
    time_it brig cat "archive_$size" /dev/null
    time_it dd if="data/mount/archive_$size" of=/dev/null bs=4M status=none
    size=$(expr $size \* 2)
done
```

C.2. plot_results.sh

```
#!/usr/bin/env python3
#encoding: utf8

BASELINE_TIME = [3, 7, 6, 8, 10, 22, 32, 58, 114, 201]

# movie write

MOVIE_COMPRESS_SINGLE = \
    [2, 2, 8, 12, 30, 50, 102, 220, 359, 744]
MOVIE_ENCRYPT_SINGLE = \
    [16, 19, 31, 89, 194, 367, 692, 1417, 2926, 5727]
MOVIE_ENCRYPT_PLUS_COMPRESS = \
    [9, 25, 45, 108, 234, 401, 794, 1668, 3417, 6900]
MOVIE_TILL_IPFS = \
    [20, 51, 86, 208, 428, 821, 1650, 3227, 6473, 12943]
MOVIE_IPFS_RAW = \
    [59, 84, 100, 152, 221, 378, 648, 1223, 2328, 4625]
MOVIE_BRIG_ADD =  \
    [338, 363, 397, 495, 634, 1279, 2629, 4028, 7239, 13910]

PLOT_MOVIE_WRITE = {
    'short': 'movie_write.svg',
    'title': 'Throughput of encryption, compression and stacked (movie.mp4)',
    'names': [
        ('baseline', BASELINE_TIME),
        ('only compress', MOVIE_COMPRESS_SINGLE),
        ('only encrypt', MOVIE_ENCRYPT_SINGLE),
        ('encrypt/compress', MOVIE_ENCRYPT_PLUS_COMPRESS),
        ('ipfs add', MOVIE_IPFS_RAW),
        ('ipfs add/encrypt/zip', MOVIE_TILL_IPFS),
        ('brig stage', MOVIE_BRIG_ADD),
    ]
}

# movie read

MOVIE_DECOMPRESS = \
    [82, 825, 928, 1202, 1221, 1244, 2213, 3236, 5282, 22059]
MOVIE_DECRYPT =  \
    [85, 110, 126, 171, 234, 376, 738, 1414, 2693, 6229]
```

```
MOVIE_IPFS_CAT_AND_DECRYPT_DECOMPRESS =  \
    [105, 118, 137, 194, 290, 481, 892, 1692, 3121, 6303]
MOVIE_IPFS_CAT =  \
    [75, 89, 109, 136, 192, 321, 582, 1059, 2019, 3956]
MOVIE_BRIG_CAT =  \
    [331, 355, 375, 471, 637, 972, 1888, 2887, 4005, 6932]
MOVIE_FUSE_CAT = \
    [147, 245, 464, 760, 1598, 2783, 6009, 11861, 24226, 50311]

PLOT_MOVIE_READ = {
    'short': 'movie_read.svg',
    'title': 'Throughput of decryption, decompression & more (movie.mp4)',
    'names': [
        ('baseline', BASELINE_TIME),
        ('only decompress', MOVIE_DECOMPRESS),
        ('only decrypt', MOVIE_DECRYPT),
        ('both', MOVIE_IPFS_CAT_AND_DECRYPT_DECOMPRESS),
        ('ipfs cat', MOVIE_IPFS_CAT),
        ('brig cat', MOVIE_BRIG_CAT),
        ('fuse cat', MOVIE_FUSE_CAT),
    ]
}

# ====================

# Archive write
ARCHIVE_COMPRESS_SINGLE =  \
    [6, 15, 21, 45, 73, 157, 329, 551, 1251, 3003]
ARCHIVE_ENCRYPT_SINGLE =  \
    [16, 34, 40, 91, 204, 343, 694, 1437, 2879, 5667]
ARCHIVE_ENCRYPT_PLUS_COMPRESS =  \
    [18, 16, 50, 99, 181, 304, 671, 1257, 2904, 6701]
ARCHIVE_TILL_IPFS =  \
    [9, 33, 74, 146, 307, 533, 992, 1978, 4498, 10492]
ARCHIVE_IPFS_RAW =  \
    [75, 87, 111, 154, 205, 346, 666, 1209, 2357, 4616]
ARCHIVE_BRIG_ADD =  \
    [325, 350, 380, 423, 546, 926, 1455, 2749, 5380, 12683]

PLOT_ARCHIVE_WRITE = {
    'short': 'archive_write.svg',
    'title': 'Throughput of encryption, compression and stacked (archive.tar)',
```

```
    'names': [
        ('baseline', BASELINE_TIME),
        ('only compress', ARCHIVE_COMPRESS_SINGLE),
        ('only encrypt', ARCHIVE_ENCRYPT_SINGLE),
        ('encrypt/compress', ARCHIVE_ENCRYPT_PLUS_COMPRESS),
        ('ipfs add', ARCHIVE_IPFS_RAW),
        ('ipfs add/encrypt/zip', ARCHIVE_TILL_IPFS),
        ('brig stage', ARCHIVE_BRIG_ADD),
    ]
}

ARCHIVE_DECOMPRESS = \
    [84, 82, 96, 108, 133, 182, 271, 467, 957, 2100]
ARCHIVE_DECRYPT = \
    [99, 91, 125, 153, 233, 381, 676, 1342, 2525, 4990]
ARCHIVE_IPFS_CAT_AND_DECRYPT_DECOMPRESS = \
    [110, 115, 137, 181, 254, 392, 698, 1406, 2644, 5163]
ARCHIVE_IPFS_CAT = \
    [76, 91, 95, 130, 202, 330, 573, 1077, 2033, 3904]
ARCHIVE_BRIG_CAT = \
    [349, 341, 354, 402, 517, 715, 1093, 1928, 3160, 5961]
ARCHIVE_FUSE_CAT = \
    [104, 231, 319, 688, 956, 2817, 5078, 9613, 18996, 42485]

PLOT_ARCHIVE_READ = {
    'short': 'archive_read.svg',
    'title': 'Throughput of decryption, decompression & more (archive.tar)',
    'names': [
        ('baseline', BASELINE_TIME),
        ('only decompress', ARCHIVE_DECOMPRESS),
        ('only decrypt', ARCHIVE_DECRYPT),
        ('both', ARCHIVE_IPFS_CAT_AND_DECRYPT_DECOMPRESS),
        ('ipfs cat', ARCHIVE_IPFS_CAT),
        ('brig cat', ARCHIVE_BRIG_CAT),
        ('fuse cat', ARCHIVE_FUSE_CAT),
    ]
}

import pygal
import pygal.style
```

```
def render_plot(data, logarithmic=False):
    line_chart = pygal.Line(
        legend_at_bottom=True,
        logarithmic=logarithmic,
        style=pygal.style.LightSolarizedStyle,
        interpolate='cubic'
    )
    line_chart.title = data['title']
    line_chart.x_labels = ['{} MB'.format(2 ** idx) for idx in range(0, 10)]
    line_chart.x_title = "Input size in MB"
    line_chart.y_title = "Time in milliseconds"
    for name, points in data['names']:
        line_chart.add(name, points)

    line_chart.render_to_file(data['short'])


render_plot(PLOT_MOVIE_WRITE, logarithmic=True)
render_plot(PLOT_MOVIE_READ, logarithmic=True)
render_plot(PLOT_ARCHIVE_WRITE, logarithmic=True)
render_plot(PLOT_ARCHIVE_READ, logarithmic=True)
```

D Anhang: Inhalt des Datenträgers

Der Datenträger zu dieser Arbeit enthält folgende Dateien und Verzeichnisse:

- `./brig/`: Enthält das `git`–Repository von `brig` zum Abgabezeitpunkt (`git-rev`: `fa9bb63`).
- `./brig-vendor/`: Enthält ein `git`–Repository mit allen Abhängigkeiten zu `brig`.
- `./brig-thesis/`: Die Quellen, die zum Erzeugen des vorliegenden Dokuments nötig sind.
- `./thesis.pdf`: Das vorliegende Dokument im PDF–Format.
- `./thesis-twoside.pdf`: Eine Version von `thesis.pdf`, die sich für beidseitigen Druck eignet.
- `./html-thesis/`: HTML–Version des vorliegenden Dokuments.
- `README.txt`: Der Inhalt dieser Seite in Textform.

Diese Arbeit ist zudem online in einer rudimentären HTML–Version und als PDF verfügbar:

- *HTML, einseitig:* https://disorganizer.github.io/brig-thesis/brig/html/index.html
- *PDF:* https://disorganizer.github.io/brig-thesis/brig/thesis.pdf

Alle Diagramme wurden mit dem Online–Diagrammeditor *Lucidchart* gezeichnet, welcher für Studenten kostenlos nutzbar ist. Die eigentliche Arbeit wurde mit dem Editor »`neovim`[1]« in *Pandoc-Markdown* verfasst und mittels »pandoc« zu LaTeX kompiliert. Dies wurde schließlich mit dem `pdfTeX`–Backend zum vorliegenden Dokument gewandelt.

[1] https://neovim.io

Literaturverzeichnis

[1] C. Piechula, “Sicherheitskonzepte und Evaluation dezentraler Dateisynchronisationssysteme am Beispiel »brig«,” 2016.

[2] C. S. Peter Mahlmann, *Peer-to-Peer-Netzwerke*. eXamen.press, 2007.

[3] J. Quintard, “Towards a worldwide storage infrastructure,” PhD thesis, University of Cambridge, 2012.

[4] J. Borg, “SyncThing: Block exchange protocol (2015).” 2015.

[5] J. P. Valentin Haenel, *Git - verteile versionsverwaltung für code und dokumente*. open source Press, 2011.

[6] A. Technologies, “Durchschnittliche verbindungsgeschwindigkeit der internetanschlüsse in deutschland vom 3. quartal 2007 bis zum 1. quartal 2016 (in kbit/s),” 2016. [Online]. Available: https://de.statista.com/statistik/daten/studie/416534/umfrage/durchschnittliche-internetgeschwindigkeit-in-deutschland/

[7] K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota, “Birthday paradox for multi-collisions,” in *International conference on information security and cryptology*, 2006, pp. 29–40.

[8] R. Pike, “The Good, the Bad, and the Ugly: The Unix Legacy,” 2001. [Online]. Available: http://herpolhode.com/rob/ugly.pdf.

[9] StatCounter, “Marktanteile der führenden betriebssystemversionen weltweit von januar 2009 bis juli 2016,” 2016. [Online]. Available: https://de.statista.com/statistik/daten/studie/157902/umfrage/marktanteil-der-genutzten-betriebssysteme-weltweit-seit-2009.

[10] J. Benet, “Ipfs-content addressed, versioned, p2p file system,” *arXiv preprint arXiv:1407.3561*, 2014.

[11] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *International workshop on peer-to-peer systems*, 2002, pp. 53–65.

[12] M. J. Freedman, E. Freudenthal, and D. Mazieres, “Democratizing content publication with coral.” in *NSDI*, 2004, vol. 4, pp. 18–18.

[13] I. Baumgart and S. Mies, “S/kademlia: A practicable approach towards secure key-based routing,” in *Parallel and distributed systems, 2007 international conference on*, 2007, vol. 2, pp. 1–8.

[14] M. Szydlo, “Merkle tree traversal in log space and time,” in *International conference on the theory and applications of cryptographic techniques*, 2004, pp. 541–554.

[15] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[16] R. Cox and W. Josephson, “File synchronization with vector time pairs,” 2005.

[17] J. R. Douceur, A. Adya, W. J. Bolosky, P. Simon, and M. Theimer, “Reclaiming space from duplicate

files in a serverless distributed file system," in *Distributed computing systems, 2002. proceedings. 22nd international conference on*, 2002, pp. 617–624.

[18] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *International conference on the theory and application of cryptology and information security*, 2000, pp. 531–545.

[19] Y. Nir and A. Langley, "ChaCha20 and poly1305 for ietf protocols," 2015.

[20] K. M. Martin, *Everyday cryptography*. Oxford University Press, 2012.

[21] Wikipedia, "Snappy (datenkompressionssoftware) — wikipedia, die freie enzyklopädie," 2015. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Snappy_(Datenkompressionssoftware)&oldid=149622998.

[22] Wikipedia, "LZ4 — wikipedia, die freie enzyklopädie," 2016. [Online]. Available: https://de.wikipedia.org/w/index.php?title=LZ4&oldid=155946163.

[23] Wikipedia, "Brotli — wikipedia, die freie enzyklopädie," 2016. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Brotli&oldid=154905741.

[24] R. T. Peter Saint–Andre Kevin Smith, *XMPP: THe definitive guide*. O'Reilly, 2009.

[25] Wikipedia, "Zookos dreieck — wikipedia, die freie enzyklopädie," 2015. [Online]. Available: https://de.wikipedia.org/w/index.php?title=Zookos_Dreieck&oldid=145190860.

[26] B. W. K. Alan A. A. Donovan, *The go programming language*. Addison-Wesley, 2015.

[27] R. Pike, "The Go Programming Language," *Talk given at Google's Tech Talks*, 2009.

[28] M. E. Conway, "Design of a separable transition-diagram compiler," *Communications of the ACM*, vol. 6, no. 7, pp. 396–408, 1963.

[29] C. Percival and S. Josefsson, "The scrypt password-based key derivation function," 2015.

[30] S. Oviatt, "Human-centered design meets cognitive load theory: Designing interfaces that help people think," in *Proceedings of the 14th acm international conference on multimedia*, 2006, pp. 871–880.

[31] I.-S. S. Board, Ed., *IEEE standard for information technology-portable operating system interface (posix)-part 1: System application program interface (api)- amendment d: Additional real time extensions [c language]*. 1999, pp. 0_1–114.

[32] H.-Y. Lin and W.-G. Tzeng, "A secure decentralized erasure code for distributed networked storage," *IEEE transactions on Parallel and Distributed Systems*, vol. 21, no. 11, pp. 1586–1594, 2010.

[33] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (mime) part two: Media types," 1996.

Eidesstattliche Erklärung

Hiermit versichere ich, die eingereichte Abschlussarbeit selbständig verfasst und keine andere als die von mir angegebenen Quellen und Hilfsmittel benutzt zu haben. Wörtlich oder inhaltlich verwendete Quellen wurden entsprechend den anerkannten Regeln wissenschaftlichen Arbeitens zitiert. Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht anderweitig als Abschlussarbeit eingereicht wurde.

Das Merkblatt zum Täuschungsverbot im Prüfungsverfahren der Hochschule Augsburg habe ich gelesen und zur Kenntnis genommen. Ich versichere, dass die von mir abgegebene Arbeit keinerlei Plagiate, Texte oder Bilder umfasst, die durch von mir beauftragte Dritte erstellt wurden.

Augsburg, den 17. Oktober 2016 *Christopher Pahl*