Institut für Parallele und Verteilte Systeme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3229

Datenspeicherung in der Cloud: Clustering-Schicht für CloudFS

Thorsten Frosch

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h.c. Kurt Rothermel
Betreuer:	Dipl.-Inf. Damian Philipp
begonnen am:	10. Oktober 2011
beendet am:	17. April 2012
CR-Klassifikation:	D.4.3, E.5

Abstract

CloudFS ist ein Dateisystem, das die Speicherung von Daten bei Cloud-Diensten zur Datenspeicherung ermöglicht, allerdings keinen zentralen Serverdienst und direkte Kommunikation der beteiligten Clients voraussetzt. Es bietet anstelle eines zentralen Kontrollprozesses von herkömmlichen Cluster-Dateisystemen einen verteilten Kontrollprozess zum gleichzeitigen Zugriff von mehreren Geräten auf einen Datenspeicherungsdienst, ohne jedoch auf direkte Kommunikation zwischen den Clients zurückzugreifen. CloudFS speichert dabei selbst keine Daten, sondern bietet eine Abstraktionsschicht auf andere, zugrunde liegende Speicherprotokolle. Diese bleiben für den Nutzer transparent und der Anwender greift auf die Daten zu, als befänden sie sich auf einer lokalen Festplatte. In dieser Arbeit wird der Entwurf von CloudFS beschrieben und eine Prototyp-Implementierung für das Betriebssystem Unix erstellt. Außerdem wird die Implementierung auf ihre Leistungsfähigkeit untersucht und es werden verschiedene Parameter evaluiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1. Einleitung

Cloud Computing ist ein Ansatz für das Bereitstellen von Infrastruktur wie Rechenleistung, Speicher und Netze, sowie das Bereitstellen von Software wie Betriebssysteme und Anwendungen über das Internet, der sich Mitte des vergangenen Jahrzehnts entwickelt hat. Es wird ein Teil der Hard- und Software nicht mehr vom Nutzer selbst betrieben, sondern er nutzt die Ressourcen eines Anbieters, der zumeist geografisch entfernt ist. Cloud Computing wird aber auch von Firmen eingesetzt, die die Dienste über ein Intranet zur Verfügung stellen.

Der Zugriff auf Speicherplatz in der Cloud kann über unterschiedliche Protokolle erfolgen. Weit verbreitet sind das File Transfer Protocol (FTP), das Web-based Distributed Authoring and Versioning-Protokoll (WebDAV) oder auch das internet Small Computer System Interface (iSCSI). Dabei verwenden die Protokolle unterschiedliche Abstraktionsebenen: FTP oder auch WebDAV operieren auf Dateien, während zum Beispiel iSCSI blockorientiert ist, also mit Fragmenten von Dateien arbeitet.

Anwender können nicht zuletzt durch die immer weiter voranschreitende Verbreitung des mobilen Internets mit mehreren Endgeräten Anwendungen der Cloud ausführen und auf online gespeicherte Daten zugreifen. So können Zugriffe auf in der Cloud gespeicherte Daten von Anwendungen erfolgen, die auf einem Heimcomputer, einem Laptop, einem Mobiltelefon oder in der Cloud selbst ausgeführt werden. Um einen parallelen Zugriff auf die Daten zu ermöglichen, ist eine Koordination der Geräte notwendig. Es existieren bereits Cluster-Dateisysteme, die einen zentralen Kontrollprozess besitzen, um den Zugriff auf die Daten zu koordinieren. Allerdings besitzt ein Nutzer nicht zwingend die Möglichkeit, einen solchen Kontrollprozess zentral auf einem Server auszuführen. Protokolle wie NFS oder WebDAV erfordern ebenfalls einen zentralen Server, auf dem der Dienst ausgeführt wird. Es existieren auch Cluster-Dateisysteme, die die Zugriffssynchronisation verteilt realisieren. Durch die Verwendung von Network Address Translation (NAT) kann jedoch eine direkte Kommunikation der Clients unmöglich sein.

CloudFS ist ein neuartiger Ansatz, der die Speicherung von Daten bei Cloud-Diensten zur Datenspeicherung ermöglicht, allerdings keinen zentralen Serverdienst und direkte Kommunikation der beteiligten Clients voraussetzt. Es bietet anstelle eines zentralen Kontrollprozesses von herkömmlichen Cluster-Dateisystemen einen verteilten Kontrollprozess zum gleichzeitigen Zugriff von mehreren Geräten auf einen Datenspeicherungsdienst, ohne jedoch auf direkte Kommunikation zwischen den Clients zurückzugreifen. CloudFS speichert dabei selbst keine Daten, sondern bietet eine Abstraktionsschicht auf andere, zugrunde liegende Speicherprotokolle. Diese bleiben für den Nutzer transparent und der Anwender greift auf die Daten zu, als befänden sie sich auf einer lokalen Festplatte.

Die Architektur von CloudFS ist in Schichten eingeteilt. Die Koordination des Zugriffs auf die Daten übernimmt die Cluster-Schicht. Es sind noch weitere Schichten und Module vorhanden, die die Funktionalität von CloudFS erweitern. So ist beispielsweise eine Kryptografie-Schicht zum Verschlüsseln der Daten ebenfalls Bestandteil von CloudFS. Sie soll dem Nutzer ermöglichen, sensible Daten bei Datenspeicherdiensten abzulegen, obwohl dem Anbieter nicht vertraut wird. Der Hauptbestandteil dieser Arbeit wird der Entwurf der Cluster-Schicht sein.

Eine Prototyp-Implementierung von CloudFS wurde für das Betriebssystem Unix erstellt. Dafür wurde das Framework Filesystem in Userspace (FUSE) verwendet. Es ermöglicht die Nutzung von Dateisystemtreibern im Userspace, wodurch der Anwender keine Root-Rechte benötigt, um CloudFS zu verwenden. Außerdem wird durch das Framework die Implementierung vereinfacht, da kein eigenes Kernel-Modul erstellt werden muss.

Am Ende der Arbeit wird die Prototyp-Implementierung auf ihre Leistungsfähigkeit untersucht. Dabei werden die Ausführungsdauern einzelner Dateisystemoperationen unter verschiedenen Bedingungen evaluiert.

Die Arbeit gliedert sich in mehrere Abschnitte: In Kapitel 2 werden die Grundlagen von Dateisystemen erläutert sowie das Framework FUSE vorgestellt, das zur Implementierung des Treiber-Prototyps verwendet wurde. Außerdem werden verschiedene Verfahren zur Synchronisation von Prozessen bei Lese-/Schreibkonflikten beschrieben, wie sie auch bei Datenbankanwendungen verwendet werden und auch in CloudFS zum Einsatz kommen. Im nächsten Kapitel wird das zugrunde liegende Systemmodell beschrieben, für das CloudFS entwickelt wurde. In Kapitel 4 wird die Architektur von CloudFS vorgestellt. Hier wird vorrangig auf den Entwurf und die verwendeten Protokolle und Algorithmen eingegangen. In Kapitel 5 wird die Implementierung des Treiber-Prototyps vorgestellt. In Kapitel 6 wird anschließend die Evaluation des entwickelten Treibers beschrieben. Abschließend erfolgt in Kapitel 7 eine Zusammenfassung der Arbeit und ein Ausblick auf mögliche Weiterentwicklungen.

2. Grundlagen

In diesem Kapitel werden die Grundlagen von Dateisystemen erläutert. Dabei wird im Speziellen auf die verschiedenen Dateisystemarten eingegangen. Außerdem werden Synchronisationsverfahren vorgestellt, die vor allem in Datenbankanwendungen zum Einsatz kommen, wenn mehrere Prozesse auf eine gemeinsame Ressource lesend und schreibend zugreifen. Ebenfalls wird das FUSE-Framework vorgestellt, mit dem eigene Dateisystemtreiber entwickelt werden können.

2.1. Dateisysteme

Ein Dateisystem ist eine Form der Ablageorganisation von Daten auf einem Datenträger. Es stellt die Verbindungsschicht zwischen physikalischen Datenträger und dem Betriebssystem dar. Dabei wird von den auf dem Datenträger physikalisch gespeicherten Daten auf Dateien, einem Verbund von inhaltlich zusammenhängenden Daten, abstrahiert. Das Dateisystem bietet dem Betriebssystem verschiedene Methoden zur Dateimanipulation an, darunter zum Beispiel Methoden zum Öffnen und Schließen, Lesen und Schreiben und zum Erzeugen und Löschen von Dateien. Eine Datei besitzt je nach Dateisystem verschiedene Attribute, die Informationen wie etwa Zugriffsrechte, Erstellungsdatum oder Art der Datei beinhalten. Der Dateizugriff erfolgt über einen eindeutigen Bezeichner, der in den meisten Fällen aus einem Verzeichnispfad und dem Dateinamen besteht.

Eine vom Nutzer ausgeführte Anwendung hat keine Kenntnis über die Beschaffenheit und Struktur der verwendeten Dateisysteme. Um dies zu ermöglichen, kommt das Virtual File System (VFS) zum Einsatz, einer Abstraktionsschicht über den Dateisystemen. Das VFS stellt einen uniformen Zugriff auf die gespeicherten Daten zur Verfügung, unabhängig vom verwendeten Dateisystem. Dazu muss die Anfrage einer Anwendung wie ein Lese- oder Schreibzugriff in einen Aufruf umgesetzt werden, der der entsprechenden Funktion des betreffenden Dateisystems entspricht.

Journaling-Dateisysteme schreiben Änderungen vor dem eigentlichen Schreiben in einen reservierten Teil des Speichers, dem Journal. Dieses Konzept wird genutzt, um Inkonsistenzen bei Auftreten von Fehlern während des Schreibvorgangs zu vermeiden. Es wird zwischen zwei Arten des Journaling unterschieden: Beim Full-Journaling werden Metadaten und Dateiinhalte ins Journal geschrieben, beim Metadaten-Journaling nur die Metadaten. Der Unterschied besteht bei der Wiederherstellung des Dateisystems nach einem Fehler: Das Metadaten-Journaling garantiert lediglich die Konsistenz des Dateisystems, das Full-Journaling zusätzlich auch die Konsistenz der Nutzerdaten.

2.1.1. Dateisysteme für lokale Speicher

Auf physisch verfügbaren Speichern wie direkt verbundenen Festplatten kommen typischerweise blockbasierte Dateisysteme zum Einsatz. Diese teilen den Speicherplatz des Speichergeräts in einzelne Blöcke fester Größe auf, um eine effiziente Verwaltung des Speichers zu ermöglichen. Die Blockgröße beträgt in den meisten Systemen 512 Bytes, wobei in moderneren Systemen mehrere Blöcke zu Clustern zusammengefasst werden. Aus Sicht des Betriebssystems werden immer ganze Cluster geschrieben und gelesen. Eine Datei ist je nach Größe auf mehrere Cluster aufgeteilt. Die Verwaltung der zugehörigen Cluster unterscheidet sich je nach verwendetem Dateisystem.

Ein Beispiel für ein einfaches blockbasiertes Dateisystem ist das File Allocation Table (FAT, [Micb]), das von Microsoft entwickelt wurde und in vielen Windows-Versionen zum Einsatz kam. Hauptbestandteil dieses Dateisystems sind zwei Tabellen, in denen die Dateien verwaltet werden. Die erste Tabelle, die FAT genannt wird und namensgebend für das Dateisystem war, verwaltet die freien und belegten Cluster des Dateisystems. Für jeden Cluster des Datenbereichs existiert ein Eintrag in dieser Tabelle. Dieser gibt an, ob der Cluster frei, belegt oder defekt ist. Im Fall einer Belegung des Clusters gibt der Eintrag die Nummer des nächsten Clusters der Datei an oder enthält die Information, dass dies der letzte Cluster der Datei ist. Das bedeutet, dass die belegten Cluster einer Datei eine einfach verkettete Liste bilden. Wegen der Wichtigkeit der FAT für die Konsistenz des Dateisystems existieren oft mehrere Kopien, um bei Fehlern eine Datenwiederherstellung zu ermöglichen.

Eine zweite Tabelle ist für die Verwaltung des Stammverzeichnisses und dessen Unterverzeichnisse zuständig. Die erste Version von FAT legte pro Datei oder Unterverzeichnis einen Eintrag an. Jeder Eintrag enthält Meta-Informationen wie Dateiname, Erstellungs- und Änderungsdatum, Dateiattribute, Größe und Start-Cluster. Bei einem Lesezugriff wird also zuerst der zur Datei zugehörige Eintrag in der Stammverzeichnistabelle gesucht und dann über die FAT auf die einzelnen Cluster der Datei zugegriffen.

Ein weiterer Vertreter der blockbasierten Dateisysteme ist das Third Extended Filesystem (ext3, [ext]), das in Linux-Betriebssystemen oft zum Einsatz kommt. Ext3 fasst mehrere Blöcke, in die der Datenträger eingeteilt ist, zu Blockgruppen zusammen. Ein Superblock am Anfang des Speicherbereichs speichert wichtige Informationen über die Konfiguration des Dateisystems. Aufgrund seiner Bedeutung existieren mehrere Kopien des Superblocks in verschiedenen Blockgruppen. Ein fundamentales Konzept des Extended Filesystem sind die Inodes. Jedes Objekt im Dateisystem, also alle Dateien und Verzeichnisse, sind durch einen Inode repräsentiert. Ein Inode besitzt Informationen zu den Metadaten des Objekts wie Zugriffsrechte, Größe und Anzahl der benutzten Blöcke. Außerdem enthält jeder Inode Zeiger auf Blöcke, in denen die Daten des Objekts gespeichert sind. Alle Daten einer Datei werden soweit möglich in einer einzelnen Blockgruppe gespeichert, um Fragmentierung zu vermeiden und die Zugriffsgeschwindigkeit zu erhöhen. Es sind 12 Zeiger vorhanden, die direkt auf die ersten 12 Datenblöcke verweisen. Daneben gibt es noch einen Zeiger auf einen Block mit direkten Blockadressen und einen Zeiger auf einen Block mit indirekten Blockadressen, wobei diese indirekten Blockadressen wiederum auf Blöcke mit direkten Blockadressen verweisen. Bei einer Blockgröße von 1 KB können somit Dateien maximal

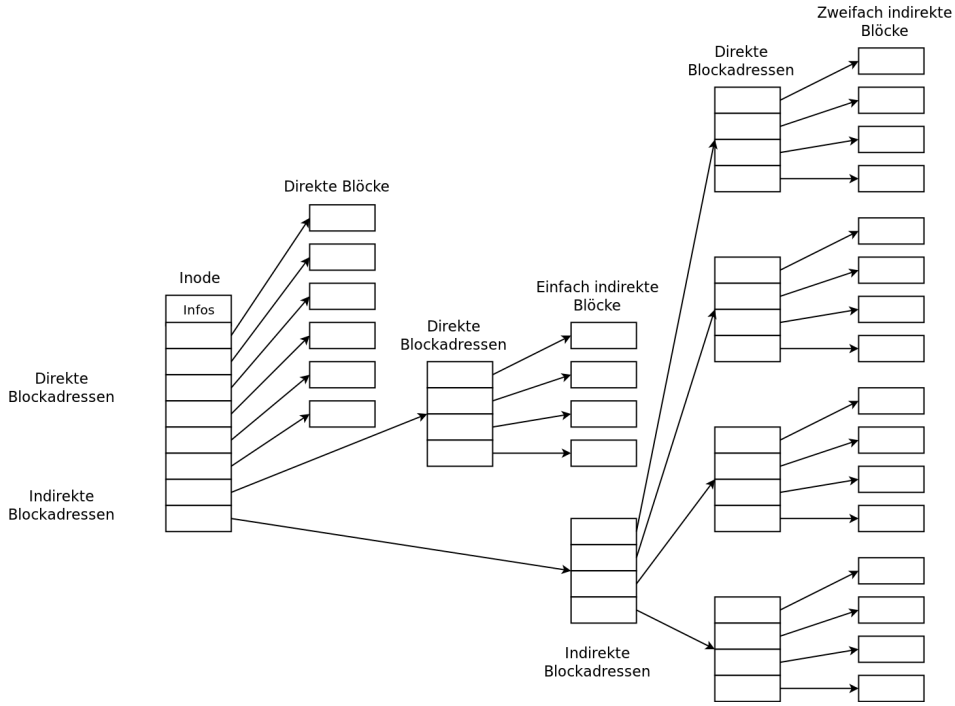


Abbildung 2.1.: Prinzip einer Inode-Struktur

eine Größe von 16 GB haben. Eine Veranschaulichung des Zeigerkonzepts bei Inodes ist in Abbildung 2.1 zu sehen. Das Journaling besitzt drei Stufen: bei gewählter Full-Option werden sowohl Metadaten als auch Dateiinhalte ins Journal geschrieben. Dies ist zeitaufwendig, allerdings erhöht es auch die Zuverlässigkeit. Die Ordered-Option schreibt nur die Metadaten ins Journal, allerdings erst nachdem der Dateiinhalt in den Speicher geschrieben wurde. Dadurch können abgebrochene Schreibvorgänge bei neuen Dateien und Schreibvorgänge, die eine bestehende Datei vergrößern, nach einem Fehler erkannt und repariert werden. Die letzte Möglichkeit, die Writeback-Option, schreibt ebenfalls nur die Metadaten ins Journal. Allerdings wird hier nicht garantiert, dass dies erst nach dem Schreiben der Dateiinhalte geschieht. Diese Möglichkeit bietet den geringsten Schutz, ist aber dafür die schnellste verfügbare Methode.

2.1.2. Netzwerk-Dateisysteme

Im Gegensatz zu den lokalen Dateisystemen, die direkten Zugriff auf einen physikalischen Speicher besitzen, stellen Netzwerk-Dateisysteme den Zugriff auf Dateien eines entfernten Servers über ein Netzwerk her. Der Client bindet also transparent für den Nutzer ein entferntes Dateisystem ein und greift mit den gleichen Systemaufrufen wie bei einem lokalen Dateisystem darauf zu, die vom VFS dann entsprechend umgesetzt und an das Netzwerk-Dateisystem weitergeleitet werden. Wenn mehrere Clients parallel auf die Daten zugreifen

wollen, übernimmt der Server die Koordination des Zugriffs. Im Gegensatz zu blockbasierten Dateisystemen ist die kleinste Einheit in den meisten Netzwerkdateisystemen eine Datei.

Ein bekanntes Beispiel für Netzwerkdateisysteme ist das Network File System (NFS, [Now89]), das auf dem Remote-Procedure-Call-Modell (RPC) basiert. Bis zu NFS Version 3 läuft die Authentifizierung auf Client-Ebene über die IP-Adresse, was bedeutet, dass den Clients die Aufgabe der Authentifizierung verschiedener Nutzer übertragen wurde. Erst ab Version 4 ist es möglich, auch auf Serverseite verschiedene Nutzer des selben Clients zu authentifizieren.

Das Web-based Distributed Authoring and Versioning Protocol (WebDAV, [web]) ist ein Standard, der die Zusammenarbeit von Nutzern erleichtern soll, die auf einen gemeinsamen Datenbestand zugreifen. Mit WebDAV, das auf dem HTTP-Protokoll basiert, kann der Nutzer auf einen Online-Speicher zugreifen und Dateien verwalten. Es wird von den meisten modernen Betriebssysteme standardmäßig unterstützt und kann wie ein normales Dateisystem eingebunden werden. WebDAV ist für den parallelen Zugriff mehrerer Clients auf eine Datei konzipiert worden. Vor einer Schreiboperation muss der Client eine Sperre für die betreffende Datei anfordern. Nach erfolgreicher Rückmeldung ist diese Datei für den Client für Schreibänderungen gesperrt und andere Clients können nicht darauf zugreifen. Nach Beendigung des Vorgangs gibt der Client die Sperre wieder frei.

Ein weiterer Vertreter aus der Klasse der Netzwerkdateisysteme ist das Internet Small Computer System Interface (iSCSI, [SMS+04]). Es basiert auf SCSI [Tec], einem Standard für die Verbindung von Peripheriegeräten mit einem Computer. iSCSI entkoppelt die direkte Verbindung von Peripheriegerät und Computer. Die SCSI-Daten werden stattdessen über ein IP-Netzwerk mit Hilfe von TCP verschickt. Damit stellt iSCSI eine Ausnahme im Bereich der Netzwerkdateisysteme dar, weil hier nicht auf der Abstraktionsebene von Dateien operiert wird, sondern wie bei herkömmlichen Dateisystemen für lokal verbundene Festplatten auf Blöcken.

Netzwerk-Dateisysteme können nicht anstelle von CloudFS für das geplante Einsatzszenario genutzt werden. Systeme wie NFS [Now89] ermöglichen zwar mehreren Clients den parallelen Zugriff über ein Netzwerk auf einen gemeinsam genutzten Speicherplatz, allerdings muss dazu auf dem Server ein entsprechender Dienst ausgeführt werden. Im geplanten Einsatzszenario für CloudFS ist es dem Nutzer nicht möglich serverseitig einen Dienst auszuführen, da ihm lediglich Online-Speicher mit einem Backend zum Datentransfer wie FTP zur Verfügung stehen. Außerdem sind im Fall von NFS die Dateisperren nicht zuverlässig implementiert worden, sodass Datenintegrität nicht gewährleistet werden kann. Dies soll jedoch bei CloudFS erreicht werden.

2.1.3. Cluster-Dateisysteme

Ein Cluster-Dateisystem ist ein Dateisystem, bei dem mehrere Clients direkt und potenziell konkurrierend auf einen gemeinsamen physischen Speicher zugreifen. Es werden mehrere Clients zu einem Cluster zusammengeschlossen, die dann direkt und ohne Vermittlung eines

Servers auf den Speicher zugreifen. Der direkte Zugriff der Clients wird meist durch ein Storage Area Network (SAN) realisiert, das den Speicherplatz wie einen lokalen Datenspeicher erscheinen lässt. Die Anbindung eines SAN geschieht in der Regel über Fibre Channel oder iSCSI. Beim parallelen Zugriff auf die selben Daten können Inkonsistenzen auftreten. Um diese zu vermeiden, müssen Maßnahmen getroffen werden, um den Zugriff zu koordinieren. Es existieren mehrere Ansätze: In der Regel übernimmt ein Distributed Lock Manager (DLM) die Koordination. Der DLM ist ein Modul, das auf jedem Client ausgeführt wird und sich mit den Instanzen anderer Clients koordiniert und den Zugriff auf die Daten regelt. Alternativ kann ein Metadaten-Server zum Einsatz kommen. Dieser speichert die zu den Daten gehörende Metadaten und übernimmt in den meisten Fällen auch die Koordination der Dateisperren, um Clients einen exklusiven Schreibzugriff zu ermöglichen. Zu den Vertretern der Cluster-Dateisysteme, die einen DLM nutzen, gehören etwa das Global File System (GFS, [gfs]) und Lustre [lus]. Auf die Dienste eines Metadaten-Servers greift dagegen zum Beispiel CXFS [Sil] zurück.

Cluster-Dateisysteme können wie Netzwerk-Dateisysteme auch nicht anstelle von CloudFS genutzt werden. Dateisysteme mit Metdadaten-Server wie CXFS [Sil] scheiden aus, da eine zentrale Kontrollinstanz erforderlich ist, um Inkonsistenzen durch den parallelen Zugriff zu vermeiden. Dies ist bei einem reinen Online-Speicher ohne weitergehende Privilegien für den Nutzer allerdings nicht zu realisieren. Cluster-Dateisysteme wie GFS [gfs] oder Lustre [lus], die per DLM die Synchronisation der Zugriffe realisieren, sind wegen der potenziellen Nutzung von NAT ebenfalls nicht geeignet für den Einsatz in der geplanten Arbeitsumgebung.

2.1.4. Kryptografie in Dateisystemen

Es sind verschiedene Lösungen verfügbar, um die auf einem Datenträger gespeicherten Daten zu verschlüsseln und sie somit vor unberechtigtem Lese- und Schreibzugriff anderer Nutzer zu schützen.

Bitlocker [Mica] ist eine Festplattenverschlüsselungsfunktion, die in verschiedenen Windows-Versionen zum Einsatz kommt. Dabei werden immer komplette Partitionen verschlüsselt, die erst nach erfolgreicher Authentifizierung wieder entschlüsselt werden können. Bitlocker schützt die gespeicherten Daten inklusive Metadaten damit vor Lesezugriffen während die verschlüsselte Partition nicht vom Betriebssystem genutzt wird. Als Verschlüsselungsalgorithmus wird der Advanced Encryption Standard (AES) mit einer Schlüssellänge von 128 Bit oder 256 Bit verwendet. Es werden drei verschiedene Authentifizierungsmethoden angeboten: Im Transparent Operation Mode wird mit Hilfe des Trusted Platform Module (TPM) im Hintergrund und transparent für den Nutzer die Entschlüsselung vorgenommen. Das TPM ist ein Chip auf dem Mainboard eines Computers, der vertrauliche Schlüssel speichern kann. In diesem wird ein Schlüssel gespeichert, mit dem die Partition entschlüsselt werden kann. Die Herausgabe des Schlüssels an das Betriebssystem erfolgt nur, nachdem die Hardware des Systems auf Änderungen überprüft und keine Änderungen festgestellt wurden. Die zweite mögliche Authentifizierungsmethode ist die Eingabe einer PIN, mit der dann auf den gespeicherten AES-Schlüssel zugegriffen werden kann. Die letzte Möglichkeit ist die Verwendung

eines USB-Geräts, das beim Start des Systems mit dem Computer verbunden sein muss und einen Schlüssel enthält, der wiederum den Zugriff auf den AES-Schlüssel erlaubt. Bitlocker ermöglicht auch die Kombination der verschiedenen Authentifizierungsmethoden.

Eine weitere Möglichkeit zur Datenverschlüsselung bietet die Software Truecrypt [tru], die für Windows, Linux und Mac OS X verfügbar ist. Die Daten können auf unterschiedliche Art und Weise verschlüsselt werden: Es kann wahlweise eine ganze Festplatte, einzelne Partitionen oder Container-Dateien, die ein eigenes von Truecrypt angelegtes Dateisystem enthalten, verschlüsselt werden. Die Daten werden dabei on-the-fly verschlüsselt, wobei als Verschlüsselungsalgorithmus Serpent, AES oder Twofish verwendet werden. Wenn das Gerät nicht eingebunden ist, wird es vom Betriebssystem als nicht initialisiert angezeigt. Erst nach dem Einbinden werden die Partitionen und Daten für den Nutzer sichtbar. Wenn die Verschlüsselung durch eine Container-Datei erfolgt, wird in dieser ein eigenes Dateisystem angelegt. Die erstellte Datei kann auch auf einer unverschlüsselten Partition abgelegt werden. Für den Lese- und Schreibzugriff müssen Container-Dateien eingebunden werden. Danach kann darauf wie auf ein reguläres Dateisystem zugegriffen werden. Container-Dateien haben keinen eigenen Datei-Header und können somit nicht als Truecrypt-Datei erkannt werden. Zusätzlich kann die Datei auf einer verschlüsselten Partition gespeichert und in dieser neben anderen unwichtigen Daten quasi versteckt werden. Dadurch unterstützt Truecrypt das Konzept der glaubhaften Abstreitbarkeit, also der Möglichkeit, bewusst Spuren zu Dateien zu verwischen und deren Existenz glaubhaft abstreiten zu können.

Für Unix-Betriebssysteme steht weiterhin zum Verschlüsseln von Daten das Dateisystem EncFS [Gou] zur Verfügung. Es werden dabei einzelne Dateien anstatt ganzer Partitionen verschlüsselt. Die Funktionsweise von EncFS basiert auf zwei Verzeichnissen, einem Quellverzeichnis und einem Arbeitsverzeichnis. Dateien, die der Nutzer im Arbeitsverzeichnis ablegt, werden on-the-fly verschlüsselt und im Quellverzeichnis abgelegt. Jede Datei im Arbeitsverzeichnis entspricht also einer verschlüsselten Datei im Quellverzeichnis, wobei der Dateiname ebenfalls verschlüsselt wird. Als Verschlüsselungsalgorithmus wird AES oder Blowfish verwendet. Die Dateien werden in einzelne, kleinere Dateien aufgespalten, die dann separat verschlüsselt werden. Die Größe dieser Dateien beträgt standardmäßig 512 Bytes. Die Daten werden mit einem Volume Key verschlüsselt, der wiederum mit einem Passwort gesichert ist. Zum Einbinden des verschlüsselten Verzeichnisses wird dieses Passwort benötigt; andernfalls wird der Ordner als leer angezeigt. Dieses Verfahren schützt allerdings nicht vor der Offenlegung von Metadaten wie Dateiberechtigungen oder der Zeit des letzten Zugriffs. Außerdem kann die Anzahl der Dateien jederzeit eingesehen werden.

2.2. Verfahren für den synchronisierten Zugriff auf gemeinsame Ressourcen

Wenn mehrere Prozesse auf eine gemeinsame Ressource zugreifen, muss der Zugriff koordiniert werden. Andernfalls können durch überlappende Schreibvorgänge Änderungsanomalien auftreten und die Konsistenz der Daten wird gefährdet. Im Bereich der Datenbanken sind Verfahren für synchronisierten Zugriff von großer Bedeutung, da dort parallel viele

Transaktionen gleichzeitig lesend und schreibend auf die Datenbank zugreifen können und trotzdem die Konsistenz der Datenbank gewahrt werden muss.

Die Synchronisationsverfahren können in zwei Klassen eingeteilt werden, die optimistischen und die pessimistischen Verfahren (siehe [HR01]). Beim pessimistischen Ansatz wird die Annahme getroffen, dass beim Start einer Transaktion mit hoher Wahrscheinlichkeit ein Konflikt mit einer anderen Transaktion auftritt. Deshalb wird die Ressource zu Beginn für einen exklusiven Schreibzugriff gesperrt und damit Inkonsistenzen vermieden. Die optimistischen Verfahren dagegen liegen der Annahme zu Grunde, dass in den meisten Fällen kein Konflikt auftreten wird. Deshalb wird erst nach Beenden des Vorgangs auf eventuell aufgetretene Konflikte geprüft und gegebenenfalls der Schreibvorgang zurückgesetzt.

Das RX-Sperrverfahren ist ein Vertreter der pessimistischen Synchronisationsverfahren. Das Prinzip dieses Verfahrens ist es, dass jede Transaktion vor einem Lese- oder Schreibzugriff eine entsprechende Sperre für das betreffende Objekt erwerben muss. Sofern noch keine Sperre auf ein Objekt besteht, kann eine Transaktion sowohl eine Lese- als auch eine Schreibsperre erwerben. Wenn schon eine Lesesperre auf einem Objekt besteht, kann nur eine weitere Lesesperre erworben werden, damit nicht die Daten schreibend verändert werden, die gerade von einer anderen Transaktion gelesen werden. Eine Schreibsperre kann nur erworben werden, wenn noch keine andere Transaktion eine Sperre für das betreffende Objekt angefordert hat. Es existieren noch diverse Erweiterungen für dieses Sperrverfahren mit zusätzlichen Sperren, um Deadlocks zu vermeiden und den Schreibdurchsatz zu erhöhen.

Ein weiteres pessimistisches Synchronisationsverfahren ist das Multiversion Concurrency Control, das konkurrierende Zugriffe ohne den Einsatz von Sperren ermöglicht und trotzdem die Konsistenz der Datenbank nicht gefährdet. Dazu werden von jedem Objekt mehrere Versionen angelegt, wobei nur eine davon die aktuellste ist und sich die Versionen durch einen Zeitstempel oder eine fortlaufende Transaktionsnummer unterscheiden. Jede Transaktion besitzt ebenfalls eine Transaktionsnummer oder einen Zeitstempel, mit dessen Hilfe sie dann bei lesendem Zugriff die passende Version finden kann. Schreibender Zugriff einer Transaktion blockiert bei diesem Verfahren andere Transaktionen nicht, da einfach eine neuere Version des Objekts angelegt wird.

Zu den optimistischen Synchronisationsverfahren zählt das Optimistic Concurrency Control-Verfahren (OCC). Dies lässt alle Transaktionen ihre Lese- und Schreibvorgänge ausführen und prüft anschließend, ob ein Konflikt aufgetreten ist und ob die Transaktion zurückgesetzt werden muss. Das Verfahren ist in drei Phasen eingeteilt: In der ersten Phase, der Read-Phase, wird die Transaktion ausgeführt, wobei Änderungen an einem Objekt auf einer privaten Kopie verbleiben. In der nächsten Phase, der Validierungsphase, wird überprüft, ob die Transaktion mit einer anderen parallel ausgeführten Transaktion in Konflikt geraten ist. In der Write-Phase werden zum Schluss bei positivem Ergebnis der Validierungsphase die Änderungen geschrieben oder bei aufgetretenem Konflikt die Transaktion zurückgesetzt. Für die Validierungsphase gibt es zwei verschiedene Ansätze für die Konflikterkennung: Es können entweder parallele Transaktionen, die am Ende der untersuchenden Transaktion schon abgeschlossen waren oder Transaktionen, die sich aktuell noch in der Read-Phase befinden, untersucht werden.

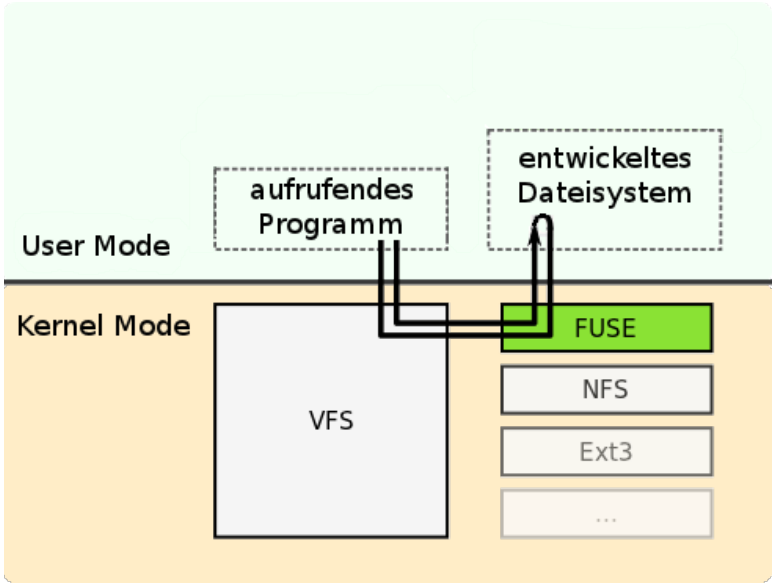


Abbildung 2.2.: Funktionsweise von FUSE (nach Vorlage von: [FUS])

2.3. FUSE

Das Filesystem in Userspace (FUSE, [FUS]) ist ein Kernel-Modul für Unix-Systeme. Es ermöglicht nicht-privilegierten Nutzern eigene Dateisysteme zu entwickeln und einzubinden, ohne den Kernel selbst verändern zu müssen. Die Funktionsweise ist in Abbildung 2.2 zu sehen: Im Gegensatz zu herkömmlichen Dateisystemtreibern befindet sich der entwickelte Dateisystemtreiber im Userspace und nicht im Kernel-Mode. Ein an dieses Dateisystem gerichteter Aufruf aus dem Userspace wird vom Virtual File System (VFS) an das FUSE-Kernel-Modul weitergeleitet, das wiederum den Aufruf an den Dateisystemtreiber im Userspace umleitet. Das Ergebnis des Aufrufs wird auf dem umgekehrten Weg zum aufrufenden Programm zurückgeschickt. Dadurch ist der Zugriff transparent für den Nutzer. Für die Realisierung eines Dateisystems müssen verschiedene Funktionen implementiert werden, die das Verhalten des Dateisystems definieren. Es muss festgelegt werden, was bei an das Dateisystem gerichtete Anfragen wie zum Beispiel dem Öffnen/Schließen einer Datei oder einem Lese-/Schreibzugriff geschehen soll. Für die Grundfunktionen eines Dateisystems wie das Lesen eines Verzeichnisses und das Anzeigen von Dateiinhalten sind nur einige wenige Funktionen zu implementieren. Ein vollständiges Dateisystem kann dagegen bis zu 30 zu implementierende Dateisystemoperation umfassen.

FUSE wird hauptsächlich dafür eingesetzt, virtuelle Dateisysteme zu entwickeln. Diese speichern Daten im Gegensatz zu herkömmlichen Dateisystemen nicht selbst, sondern bieten eine Abstraktionsschicht für andere Dateisysteme oder Datenspeicher. Es sind bereits einige Dateisystemtreiber auf der Basis von FUSE entstanden, wie etwa das GMail Filesystem over FUSE [GMa], mit dem der Speicher eines Benutzerkontos von Google Mail als Dateisystem eingebunden werden kann. Weitere Beispiele sind EncFS [Gou] (siehe Abschnitt 2.1.4), das

einzelne Dateien ohne die Einrichtung einer eigenen verschlüsselten Partition verschlüsselt oder auch davfs2 [Bau], das das Einbinden von WebDAV-Ressourcen als reguläres Dateisystem ermöglicht.

3. Systemmodell und Anforderungen an CloudFS

CloudFS soll die Datenspeicherung in Cloud-Diensten ermöglichen, auch wenn kein zentraler Kontrollprozess zur Koordination der Zugriffe vorhanden und keine direkte Kommunikationsmöglichkeit zwischen den Clients möglich ist. In diesem Kapitel wird das zugrunde liegende Systemmodell vorgestellt. Außerdem werden Anforderungen genannt, die an CloudFS gestellt werden und die bei der Implementierung berücksichtigt werden sollen.

3.1. Systemmodell

Das Systemmodell, auf dessen Annahme CloudFS entwickelt wurde, ist in Abbildung 3.1 zu sehen. Es sind drei Arten von Clients für den Einsatz von CloudFS vorgesehen: stationäre Clients, mobile Clients und Serverdienste, die in der Cloud ausgeführt werden.

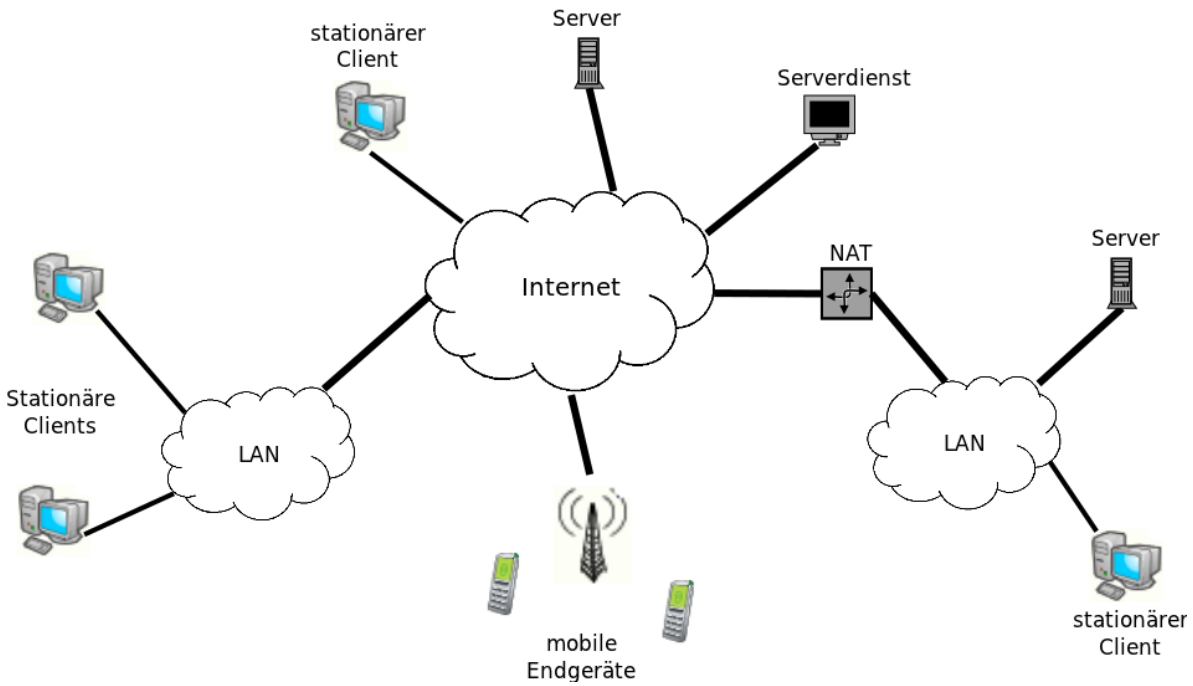


Abbildung 3.1.: Systemmodell

Stationäre Clients besitzen einen schnellen Breitbandinternetzugang, der Übertragungsgeschwindigkeiten von bis zu 100 MBit/s erlaubt. Außerdem weisen über diese Verbindung verschickte Datenpakete nur sehr geringe Verzögerungen im Bereich von weniger als 50 ms auf. Zudem sind bei dieser Art von Internetzugang nur sehr seltene bis keine Verbindungsabbrüche zu erwarten. Die zweite Art von Clients stellen mobile Geräte wie Smartphones oder Notebooks mit mobiler Internetverbindung dar. Die Übertragungsgeschwindigkeit bei dieser Verbindungsart ist in der Regel deutlich niedriger als bei Breitbandzugängen der stationären Clients. Aufgrund der Mobilfunktechnik ist außerdem mit höheren Paketlaufzeiten zu rechnen, insbesondere bei schlechter Verbindungsqualität. Im Extremfall kann die Datenverbindung auch abreißen und der mobile Client ist erst nach einer gewissen Zeit wieder in der Lage, über das Internet zu kommunizieren. Die dritte Client-Art, die CloudFS nutzt, sind Serverdienste. So ist es denkbar, dass ein selbst in einer Cloud ausgeführter Dienst auf per CloudFS bereitgestellte Daten zugreift. Die Internetverbindung solcher Serverdienste besitzt in der Regel eine sehr hohe Übertragungsgeschwindigkeit und weist nur geringe Latenzzeiten auf. Verbindungsabbrüche sind ebenfalls nur sehr selten zu erwarten.

Es besteht die Möglichkeit, dass sich manche stationäre Clients in einem LAN befinden, in dem Network Address Translation (NAT) zum Einsatz kommt. Dadurch wird ohne weitere Konfiguration eine mögliche Ende-zu-Ende-Verbindung zu Clients verhindert, die sich nicht im selben LAN befinden. Aus diesem Grund wird angenommen, dass Clients untereinander keine direkte Kommunikationsmöglichkeit besitzen und nur indirekt über den gemeinsam verwendeten Speicher kommunizieren können.

Clients, die CloudFS nutzen, greifen parallel auf einen oder mehrere Speicher zu, die über das Internet erreichbar sind. Dabei kommen unterschiedliche Protokolle zum Einsatz, mit denen der Speicherzugriff realisiert wird. Online-Speicher, die den Zugriff per WebDAV oder NFS realisieren, lassen den Nutzer auf dateibasierter Ebene auf seine Daten zugreifen. Im Gegensatz dazu ist auch ein blockbasierter Zugriff auf die Daten möglich, wie es bei lokalen Dateisystemen der Fall ist. Diese Zugriffsart wird zum Beispiel von iSCSI realisiert. Außerdem sind Server vorgesehen, die vom Nutzer selbst bereitgestellt werden. So kann der Nutzer Daten, die auf seinem Heimcomputer gespeichert sind, über ein Netzwerkdateisystem wie NFS online zur Verfügung stellen. Damit besteht dann die Möglichkeit mit anderen Clients auf diese Daten per CloudFS zuzugreifen.

Der geplante Haupteinsatzzweck von CloudFS ist der ubiquitäre Zugriff des Nutzers auf seine gespeicherten Daten. Es soll vor allem möglich sein, auf private Dokumente und Mediendaten wie Musik oder Filme von zu Hause und unterwegs zugreifen zu können. Dementsprechend wird erwartet, dass der Großteil der Dateioperationen Lesevorgänge sein werden und nur selten Daten geschrieben werden müssen. Aufgrund der potenziell sensiblen Daten muss zudem eine Möglichkeit vorgesehen werden, die gespeicherten Dateien zu verschlüsseln, um dem Anbieter des Speicherplatzes und möglichen anderen Nutzern die Daten vorenthalten zu können. Dabei sind mehrere Sicherheitsstufen vorgesehen, die zum Beispiel nur den Dateiinhalt oder auch die komplette Speicherstruktur mit Verzeichnis und Dateinamen verschlüsseln. Außerdem kann bei Bedarf nicht genutzter Speicherplatz mit Leerdaten beschrieben werden, um die Größe der gespeicherten Daten zu verschleiern.

3.2. Anforderungen an CloudFS

An das zu entwickelnde Dateisystem CloudFS werden verschiedene Anforderungen gestellt, damit ein paralleler Zugriff der Clients auf einen gemeinsamen Speicher ermöglicht werden kann. Es sind folgende Punkte zu beachten:

Koordination über Online-Speicher Netzwerkdateisysteme synchronisieren den parallelen Zugriff auf einen gemeinsam genutzten Speicher entweder über einen zentralen Kontrollprozess, der auf dem Server ausgeführt wird, oder durch direkte Kommunikation der Clients, die das Dateisystem nutzen. Aufgrund des Systemmodells kann CloudFS auf keine der beiden Möglichkeiten zurückgreifen. Es darf kein zentraler Kontrollprozess benötigt werden, um die Anforderungen an den Server zu reduzieren. Ebenso ist keine direkte Kommunikation der Clients aufgrund des möglichen Einsatzes von NAT verfügbar. Die Clients können lediglich Daten auf den gemeinsam genutzten Speicher schreiben und lesen. Aus diesen Gründen muss CloudFS die Koordination und Synchronisation der Clients über den jeweils verwendeten Online-Speicher realisieren.

Konfliktbehandlung Wie in lokalen Dateisystemen auch können verschiedene parallel ausgeführte Operationen auf dem Dateisystem zu Konflikten führen. So resultieren zum Beispiel zwei parallele Schreiboperationen zweier Clients auf der gleichen Datei in einem Konflikt. Es muss jedoch am Ende jeder Operation eine konsistente Version der Datei stehen und dies ist bei gleichzeitigem Schreibzugriff der Clients nicht gewährleistet. Deshalb müssen bei der Konzeption von CloudFS Maßnahmen getroffen werden, um Inkonsistenzen zu vermeiden, die aufgrund unverträglicher Dateisystemoperationen entstehen können.

Behandlung von Verbindungsabbrüchen Bei Mobilfunkverbindungen können sehr lange Latenzzeiten oder sogar Verbindungsabbrüche beobachtet werden. Wenn die Verbindung eines mobilen CloudFS-Clients während eines Schreibzugriffs abbricht, kann sich das Dateisystem möglicherweise in einem inkonsistenten Zustand befinden. Auch bei stationären Clients kann zum Beispiel durch einen Stromausfall die Verbindung unterbrochen werden. Deshalb muss CloudFS in der Lage sein, mit unvollständigen Schreib- und Änderungsoperationen umgehen zu können, indem es das Dateisystem wieder in einen konsistenten Zustand überführt.

Zugriffskontrolle CloudFS soll, wie viele andere Dateisysteme auch, die Möglichkeit bieten, den Zugriff auf die gespeicherten Daten zu kontrollieren. Dazu ist eine Rechteverwaltung notwendig, die für jede Datei angibt, welche Operationen ein Nutzer auf ihr ausführen darf.

Die eben genannten Anforderungen an das Dateisystem wurden beim Design von CloudFS berücksichtigt und durch verschiedene Maßnahmen umgesetzt. Die Umsetzung wird in den beiden folgenden Kapiteln beschrieben.

4. Entwurf

In diesem Kapitel wird der Entwurf des Dateisystems CloudFS beschrieben. Es wird zunächst auf die Gesamtarchitektur des Systems eingegangen. Anschließend wird die Cluster-Schicht, die den Hauptbestandteil dieser Arbeit darstellt, beschrieben. Ebenso wird auf das verwendete Verfahren zum Sperren von Dateien und Verzeichnissen eingegangen. Abschließend werden der detaillierte Aufbau des Systems und die verfügbaren Dateisystemoperationen vorgestellt. Dabei beschränkt sich diese Arbeit auf dateibasierte Online-Speicher.

4.1. Gesamtarchitektur

In Abbildung 4.1 ist die Architektur von CloudFS zu sehen. Die Architektur ist in mehrere Schichten eingeteilt. Auf der obersten Ebene sind die implementierten Dateisystemtreiber, die dem Nutzer die gespeicherten Daten der Online-Speicher zur Verfügung stellen. Es ist für jedes der drei gängigsten Betriebssysteme Windows, Linux und Mac OS ein Treiber vorgesehen. CloudFS soll auch auf Smartphones zum Einsatz kommen können, weshalb auch Treiber für iPhone OS, Android und Symbian benötigt werden. Außerdem befindet sich hier die Konfigurationsschnittstelle. Diese stellt dem Nutzer eine grafische Oberfläche zur Verfügung, über die sämtliche Konfigurationsparameter von CloudFS eingestellt werden können.

Unter den Dateisystemtreibern befindet sich die Management-Schicht. Sie besteht aus mehreren Modulen, die das Dateisystem um verschiedene Funktionalitäten erweitern. Die einzelnen Module sind:

Automatischer Cache Die Daten, auf die per CloudFS zugegriffen werden kann, befinden sich auf einem Online-Speicher. Vor allem bei mobilen Clients kann es vorkommen, dass keine Internetverbindung zur Verfügung steht. Um trotzdem Zugriff auf zumindest einen Teil der Daten zu haben, werden in einem vom Client selbstständig verwaltetem Cache auf einer lokalen Festplatte ausgewählte Dateien zwischengespeichert. Auf diesen Daten kann dann trotz fehlender Verbindung zum Online-Speicher gearbeitet werden. Sobald wieder Internetkonnektivität vorhanden ist, werden die Dateien mit dem Server abgeglichen. Durch die Verwendung eines Caches kann außerdem eine Beschleunigung des Systems erreicht werden, da häufig genutzte Daten zwischengespeichert werden können und nicht bei jedem Zugriff vom Server geladen werden müssen.

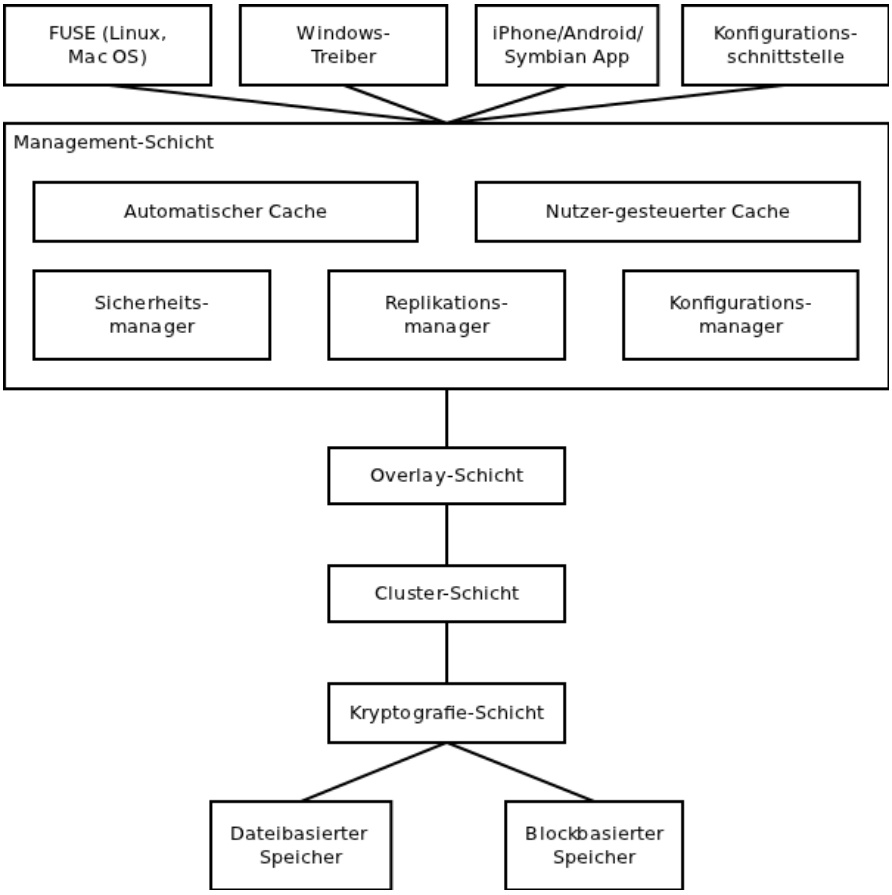


Abbildung 4.1.: Gesamtarchitektur von CloudFS

Nutzer-gesteuerter Cache Im Nutzer-gesteuerten Cache werden analog zum automatischen Cache Dateien zwischengespeichert. Er garantiert die Verfügbarkeit der betreffenden Dateien und beschleunigt den Zugriff darauf. Im Gegensatz zum automatischen Cache, der selbstständig vom System verwaltet wird, kann der Nutzer hier selbst Vorgaben erstellen, welche Dateien lokal vorgehalten werden sollen.

Sicherheitsmanager Der Sicherheitsmanager hat die Aufgabe, die vom Nutzer vergebenen Datei- und Verzeichnisrechte umzusetzen. Er muss also vor jedem Zugriff prüfen, ob der entsprechende Client ausreichend Befugnisse hat, um die gewünschte Operation durchzuführen. Die Überprüfung wird bedingt durch den fehlenden zentralen Kontrollprozess auf den Clients ausgeführt.

Replikationsmanager Es besteht die Möglichkeit, die auf einem Online-Speicher abgelegten Daten auf verschiedene andere Online-Speicher zu replizieren. Dadurch kann eine höhere Datensicherheit und Datenverfügbarkeit erreicht werden. Der Replikationsmanager ist dafür zuständig, die vom Nutzer ausgewählte Replikation auf die verschiedenen Online-Speicher abzubilden.

Konfigurationsmanager CloudFS wurde für den parallelen Einsatz auf mehreren Clients konzipiert. Für die einzelnen Clients ist die Kenntnis von bestimmten Konfigurationsparametern notwendig, um ein bestehendes CloudFS-System nutzen zu können. Damit nicht bei jedem Client, der zum ersten Mal CloudFS nutzt, alle Parameter vom Nutzer eingegeben werden müssen, werden diese in einem bestimmten Bereich im Online-Speicher hinterlegt. Der Konfigurationsmanager ist dann dafür zuständig, diese Parameter auszulesen, eine Erstkonfiguration des Clients vorzunehmen und eventuelle Änderungen wieder zurückzuschreiben.

Unter der Management-Schicht mit ihren einzelnen Modulen befindet sich die Overlay-Schicht. Sie ist dafür zuständig, dass mehrere Online-Speicher parallel von einem Client genutzt werden können. Damit ist es möglich, entweder Unterverzeichnisse auf verschiedenen Online-Speicher zu speichern oder bestimmte Verzeichnisse auf andere Online-Speicher zu replizieren.

Unterhalb der Overlay-Schicht befindet sich die Cluster-Schicht. Sie ist für die Koordination des parallelen Zugriffs der verschiedenen Clients zuständig. Die Bezeichnung der Cluster-Schicht wurde in Anlehnung an Cluster-Dateisysteme gewählt, bei denen ebenfalls die Hauptaufgabe die Koordination der parallelen Client-Zugriffe ist.

Die letzte Schicht über der tatsächlichen Speicherung der Daten auf einem Online-Speicher ist die Kryptografie-Schicht. Sie ist für die Verschlüsselung der abgelegten Daten zuständig. Es steht dem Nutzer frei, ob er eine Verschlüsselung verwenden will oder ob die Dateien unverschlüsselt gespeichert werden sollen. Der Nutzer kann dabei auf verschiedene Sicherheitsstufen zurückgreifen. Sensible Daten wie zum Beispiel Kreditkartendaten müssen zwingend vor jeglichem fremden Zugriff geschützt werden, während eine Audio-Datei nicht unbedingt diese Sicherheitsstufe erfordert. Die Wahl der Sicherheitsstufe hat auch Auswirkungen auf die Overlay-Schicht. Wenn mehrere Online-Speicher verfügbar sind, können sensible Dateien bevorzugt auf Servern gespeichert werden, denen am meisten vertraut wird. Wenn keine Verschlüsselung gewählt wurde, leitet die Kryptografie-Schicht alle Daten unverändert an die jeweiligen Online-Speicher weiter. Bei aktiver Verschlüsselung und zugrunde liegendem blockbasiertem Speicher werden die Datenblöcke beim Transfer zwischen Cluster-Schicht und Speicher je nach Richtung ver- beziehungsweise entschlüsselt.

Die unterste Schicht der Architektur von CloudFS stellen die Online-Speicher dar. Diese können entweder dateibasiert oder blockbasiert sein. Hier werden dann schließlich alle von CloudFS genutzten Daten gespeichert.

In dieser Arbeit liegt das Hauptaugenmerk auf der Cluster-Schicht, die für die Koordination des parallelen Zugriffs zuständig ist. Die anderen Komponenten sind zwar für die Gesamtarchitektur vorgesehen, werden hier jedoch nicht oder nur teilweise vorgestellt und implementiert.

4.2. Entwurf der Cluster-Schicht

Die Cluster-Schicht koordiniert den parallelen Zugriff der Clients auf die gespeicherten Dateien. Außerdem wird durch die Verwendung eines Journals sichergestellt, dass bei Auftreten von Inkonsistenzen durch Verbindungsabbrüche stets eine konsistente Version des Dateisystems rekonstruiert werden kann.

Dateien und Verzeichnisse, im Folgenden auch Objekte genannt, besitzen eine Sperre. Wenn eine Änderung an einem Objekt durchgeführt werden soll, muss zuerst die zugehörige Sperre erworben werden. Die Sperre sichert dem entsprechenden Client damit exklusiven Zugriff auf die Datei oder das Verzeichnis. Das in CloudFS verwendete Sperrverfahren wird in Abschnitt 4.2.2 beschrieben. Eine Leseoperation auf einem Objekt erfordert keine Sperre und kann immer ausgeführt werden. Dadurch ist es möglich, dass schmutzige Daten gelesen werden, wenn parallel ein anderer Client schreibend auf das Objekt zugreift. Das Sperrverfahren kann unter bestimmten Umständen einen Deadlock einer Objektsperre verursachen. In diesem Fall ist es nicht mehr möglich, dass sich ein Client den exklusiven Zugriff auf dieses Objekt sichern kann. Somit kann auf dieses Objekt nur noch lesend zugegriffen werden, bis der Deadlock beseitigt ist.

4.2.1. Journal

Das Journal ist in verschiedene Abschnitte aufgeteilt. Jeder Client muss sich einen dieser Abschnitte reservieren, bevor er das System nutzen kann. Auf dem betreffenden Abschnitt besitzt er exklusiven Schreibzugriff.

Bevor eine Änderungsoperation auf den gespeicherten Daten ausgeführt wird, legt ein Client in seinem Journal-Abschnitt einen Eintrag für die Operation an. Dieser enthält alle benötigten Informationen über die auszuführende Operation. Jeder Eintrag enthält den Namen der Operation, eine Versionsnummer und den Zeitpunkt der Änderung. Je nach Art der Operation werden weitere Informationen gespeichert (siehe Abschnitt 4.4.1 über die verfügbaren Dateisystemoperationen). Wenn ein Eintrag erfolgreich erstellt wurde, ist sichergestellt, dass die Operation im Fall eines Abbruchs während der Ausführung wiederholt werden kann. Andernfalls wird der unvollständige Eintrag verworfen und die Operation kann nicht ausgeführt werden. Somit kann immer eine konsistente Version des Dateisystems rekonstruiert werden.

Um gespeicherte Journal-Eintrage chronologisch ordnen zu können, enthalten sie eine Versionsnummer. Jede Datei und jedes Verzeichnis besitzt ebenfalls einen Versionszähler, der der Versionsnummer des zuletzt ausgeführten Journal-Eintrags entspricht. Somit kann festgestellt werden, welche Einträge bereits ausgeführt wurden und veraltet sind und welche Einträge noch ausgeschrieben werden müssen. Beim Anlegen eines neuen Journal-Eintrags muss diesem eine Versionsnummer zugewiesen werden. Dazu wird die aktuelle Version der betreffenden Datei oder des Verzeichnisses im Datenbereich gelesen. Existieren keine Journal-Einträge des Objekts, wird die gelesene Version inkrementiert und für den neuen Eintrag verwendet. Sind jedoch alte Journal-Einträge vorhanden, werden diese zusätzlich

Algorithmus 4.1 Ausschreiben von Journal-Einträgen

```
1: procedure WRITEBACK_JOURNAL_ENTRIES( )
2:     get all journal entries of all journal sections
3:     writeback changes that were not yet executed
4:     update version number
5:     delete journal entries in own journal section
6: end procedure
```

aus dem Journal gelesen und ihrer Versionsnummer entsprechend aufsteigend geordnet. Ist die höchste Version kleiner als die der Datei oder des Verzeichnisses, ist kein weiteres Vorgehen nötig. Andernfalls wird diese Version inkrementiert und für den neuen Eintrag verwendet.

Das Ausschreiben von Journal-Einträgen geschieht in der Regel direkt nach dem Erstellen des jeweiligen Eintrags. Allerdings ist gerade für mobile Clients die Möglichkeit vorgesehen, die Einträge unausgeschrieben zu hinterlassen. Diese Aufgabe kann dann zu einem späteren Zeitpunkt von einem anderen Client übernommen werden. Dieser wird dann vor dem Ausführen einer weiteren Operation auf der betreffenden Datei oder dem Verzeichnis alte Journal-Einträge in den Datenbereich ausschreiben.

Der Ausschreibevorgang ist in Algorithmus 4.1 beschrieben. Zuerst werden für die betreffende Datei oder für das betreffende Verzeichnis alle verfügbaren Journal-Einträge aus allen Journal-Bereichen zusammengestellt. Anhand der Versionsnummer der Datei oder des Verzeichnisses und der der einzelnen Einträge kann nun festgestellt werden, welche Journal-Einträge bereits ausgeführt wurden und welche noch ausgeschrieben werden müssen. Nach dem erfolgreichen Ausschreiben muss dann dementsprechend der Versionszähler der Datei oder des Verzeichnisses angepasst werden. Zum Schluss werden noch die bereits ausgeschriebenen Journal-Einträge gelöscht, die sich im Journal-Bereich des Clients befinden. Einträge, die sich in Journal-Bereichen anderer Clients befinden, können dort nicht gelöscht werden, da der ausschreibende Client nur schreibenden Zugriff auf seinen eigenen Journal-Bereich besitzt. Alte Journal-Einträge können jedoch anhand ihrer Versionsnummer gefiltert werden und werden beim nächsten Ausschreiben des Clients, der den betreffenden Journal-Bereich reserviert hat, gelöscht.

4.2.2. Sperrverfahren

Das Sperren eines Objekts für den exklusiven Zugriff kann in CloudFS nicht wie in herkömmlichen Netzwerk- oder Cluster-Dateisystemen gelöst werden. Der Grund hierfür ist das Fehlen eines zentralen Kontrollprozesses und die ausschließliche Kommunikation der Clients über die Online-Speicher. In diesem Abschnitt wird zuerst die Problematik von Sperrverfahren ohne zentrale Koordination erläutert. Anschließend wird das Sperrverfahren vorgestellt, das in CloudFS zum Einsatz kommt.

Abbildung 4.2.: Veranschaulichung des Two Army Problems

Problematik von Sperrverfahren ohne zentrale Koordination

Unter den gegebenen Voraussetzungen des Systemmodells, dass die Nachrichtenübermittlung unzuverlässig ist und ein zentraler Kontrollprozess fehlt, ist es nicht möglich, ein sicheres Sperrverfahren zu entwickeln. Einen Beweis hierfür veröffentlichten Akkoyunlu et al. [AEH75] bereits 1975. Darin wird erklärt, dass in einem verteilten System mit zwei Prozessen und unzuverlässiger Kommunikation keine Einigung über eine Transaktion erzielt werden kann, sofern kein dritter Prozess vorhanden ist, der den Gesamtstatus des Systems kennt. Akkoyunlu et al. veranschaulichten die Problemstellung durch ein Beispielszenario, bei dem Gangster eine Stadt überfallen wollen. Heute ist das Problem auch unter dem Namen „Two Army Problem“ bekannt. Es handelt dabei von einer Stadt, die umgeben ist von zwei Hügeln, auf denen jeweils eine Armee steht, wie in Abbildung 4.2 zu sehen ist. Diese Armeen wollen die Stadt angreifen, können die Stadt jedoch nur dann besiegen, wenn beide zum gleichen Zeitpunkt angreifen. Um ihre Angriffe zu koordinieren und einen Angriffszeitpunkt zu vereinbaren, schicken sie Boten durch das Tal. Diese können allerdings auf ihrem Weg durch die Stadt gefangen genommen werden.

Das Sperren eines Objekts in CloudFS lässt sich auf das Two Army Problem abbilden: Zwei Clients, die den Zugriff auf ein Objekt koordinieren wollen, stellen die beiden Armeen dar. Es muss ebenfalls wie beim Two Army Problem eine Einigung erzielt werden, im Fall von CloudFS die Übereinkunft, welcher der Clients exklusiv auf das Objekt zugreifen darf. Beim Two Army Problem besteht die Möglichkeit, dass Boten auf ihrem Weg durch das Tal gefangen genommen werden können. In CloudFS können ebenfalls Nachrichten verloren gehen: Damit zwei Clients in CloudFS kommunizieren können, müssen sie an den jeweils anderen Client adressierte Nachrichten auf dem Online-Speicher zwischenspeichern. Der Empfänger kann die Nachricht anschließend vom Online-Speicher lesen. Dieses Vorgehen ist notwendig, da eine direkte Kommunikation zwischen den Clients aufgrund des Systemmodells nicht möglich ist. Wenn beide Clients den gleichen Bereich nutzen, um dort ihre Nachrichten abzulegen, besteht die Möglichkeit, dass sie gegenseitig die Nachrichten überschreiben. Das Resultat ist der Verlust der überschriebenen Nachricht. Selbst wenn dies ausgeschlossen werden kann und beide Clients über einen eigenen Bereich verfügen, können Datenpakete auf ihrem Weg von und zum Server verloren gehen, was speziell bei Mobilfunkverbindungen häufiger der Fall ist. Somit kann in CloudFS ebenfalls wie beim Two Army Problem eine Nachricht auf dem Weg zwischen den beteiligten Parteien verloren gehen. Es fehlt beim Two

Army Problem ebenso wie in CloudFS ein zentraler Kontrollprozess. Somit lässt sich das Problem der Einigung über den exklusiven Zugriff auf ein Objekt in CloudFS direkt auf das Two Army Problem abbilden.

Akkoyunlu et al. bewiesen, dass das Two Army Problem in endlicher Zeit nicht zu lösen ist: Das Protokoll zur Lösung enthält mindestens eine Nachricht. Angenommen, es existiert eine endliche Sequenz von Nachrichten zur sicheren Übereinkunft, so muss es auch eine kürzeste Sequenz geben mit $n \geq 1$, wobei n die Anzahl der gesendeten Nachrichten ist. Da dies die kürzeste Sequenz ist, muss die letzte Nachricht wichtig sein für den Protokollablauf. Weil diese Nachricht aber verloren gehen kann, muss der Empfänger sie bestätigen, weshalb die Sequenz mindestens $n + 1$ Nachrichten benötigt. Dies ist ein Widerspruch zur Annahme und somit ist bewiesen, dass man kein endliches Protokoll für dieses Problem finden kann.

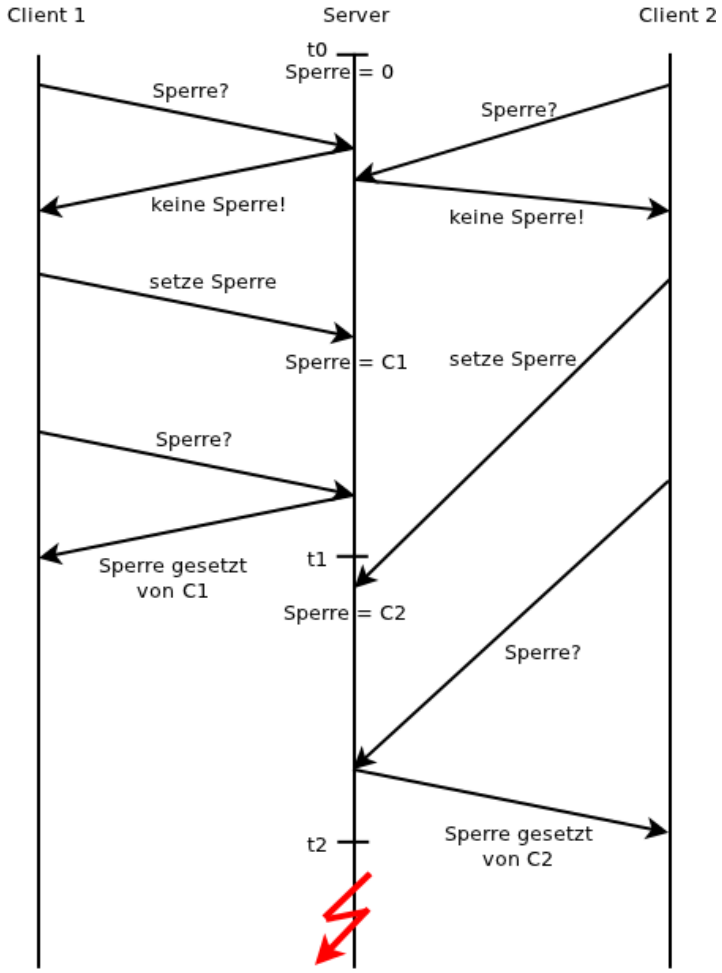


Abbildung 4.3.: Veranschaulichung des Problems der indirekten Kommunikation

Ein Beispielprotokoll (siehe Abbildung 4.3) verdeutlicht das Problem der fehlenden zentralen Koordination bei Sperrverfahren. Beide Clients versuchen, die Sperre für den Abschnitt zu setzen. Dies geschieht durch Schreiben der jeweiligen Client-ID, die jeden Client eindeutig

identifiziert. Zu Beginn bei Zeitpunkt t_0 ist die Sperre nicht vergeben (Sperre = 0). Beide Clients führen dann folgende Schritte aus:

1. Prüfe, ob die Sperre schon gesetzt ist. Wenn ja, breche ab, sonst weiter mit Schritt 2.
2. Setze die Sperre durch Schreiben der eigenen Client-ID.
3. Lese Status der Sperre vom Server. Wenn die Sperre gesetzt ist und die Client-ID mit der eigenen übereinstimmt, betrachte Objekt als gesperrt und beginne Operation auf Objekt.

Zum Zeitpunkt t_1 geht Client 1 davon aus, dass er die Sperre erhalten hat. Durch die hohe Verzögerung bei der Übermittlung der Nachrichten trifft allerdings Client 2 zum Zeitpunkt t2 ebenfalls die Annahme, dass er erfolgreich die Sperre beantragt hat. Somit hat dieses Sperrverfahren versagt, beide Clients könnten mit einem Schreibvorgang beginnen und Inkonsistenzen des Dateisystems wären die Folge.

Einfacher Sperralgorithmus

Es wurde bewiesen, dass ohne zentralen Koordinationsprozess keine Einigung über eine Transaktion erzielt werden kann. Mit der Annahme einer maximalen Nachrichtenlaufzeit t_{max} ist es allerdings auch im vorliegenden Fall möglich, ein sicheres Sperrverfahren zu erreichen. Die Annahme einer maximalen Paketlaufzeit ist für das gegebene Systemmodell zulässig, da die meisten zugrunde liegenden Speicherprotokolle wie NFS (seit Version 4) als Transportprotokoll TCP verwenden. Dieses spezifiziert die Maximum Segment Lifetime (MSL), die angibt, wie lange ein Segment (Paket) im Netzwerk verbringen kann, bevor es verworfen wird. Diese beträgt in den meisten Fällen zwei Minuten. Somit kann die maximale Paketlaufzeit für CloudFS durch die MSL angegeben werden. Falls UDP als Transportprotokoll zum Einsatz kommt, muss das Speicherprotokoll dafür Sorge tragen, dass Pakete eine maximale Lebensdauer besitzen, nach deren Ablauf sie verworfen werden.

Ein einfaches Sperrprotokoll, das nicht auf einem zentralen Koordinationsprozess oder direkter Kommunikation basiert und die Annahme einer maximalen Paketlaufzeit ausnutzt, ist in Algorithmus 4.2 zu sehen. Um eine Sperre zu erhalten, liest ein Client zuerst den aktuellen Status einer Objektsperre. Wird diese nicht bereits von einem anderen Client gehalten, schreibt der betreffende Client seine eigene eindeutige ID in das Feld der Objektsperre. Daraufhin wartet er $2 * t_{max}$, bis er nochmals den Status der Objektsperre liest. Im Worst Case überprüft ein zweiter Client kurz vor dem Setzen der Sperre des ersten Clients den Status und wird annehmen, die Sperre sei noch nicht gesetzt. Anschließend wird er versuchen, selbst die Sperre setzen. Das Lesen und Setzen des zweiten Clients dauert maximal $2 * t_{max}$. Aus diesem Grund muss ein Client diese Zeit warten, bis er nochmals den Status der Sperre überprüft. Steht nach Ablauf dieser Zeitspanne immer noch seine eigene ID in der Objektsperre, kann er sich sicher sein, dass kein anderer Client ebenfalls der Annahme ist, dass er die Sperre erfolgreich beantragt hat.

Algorithmus 4.2 Einfacher Algorithmus zur Objektsperrung

```
1:  procedure getLock( )
2:      read Lock from server
3:      if (Lock ≠ 0 AND Lock ≠ own Client-ID) then
4:          abort without success
5:      end if
6:      set Lock
7:
8:      wait for 2 * t_max
9:
10:     read Lock from server
11:     if (Lock ≠ own Client-ID) then
12:         abort without success
13:     else
14:         return success
15:     end if
16: end procedure
```

Das eben vorgestellte Sperrverfahren eignet sich nur bedingt für den Einsatz in einem Dateisystem. Bei jedem Schreibvorgang oder ähnlichen Operationen, bei denen exklusiver Zugriff auf ein Objekt verlangt wird, müsste der Client mindestens vier Minuten warten (wenn eine maximale Paketlaufzeit von zwei Minuten angenommen wird), bis er die eigentliche Dateioperation durchführen kann. Selbst für die Übertragung einer Datei mit einer Größe von nur wenigen Bytes würde der Client diese Zeit benötigen. Aus diesem Grund wird im normalen Betrieb von CloudFS zum Sperren von Dateien und Verzeichnissen ein anderes Sperrverfahren verwendet und nur in Ausnahmefällen auf das eben vorgestellte Verfahren zurückgegriffen.

Modifizierter Sperralgorithmus

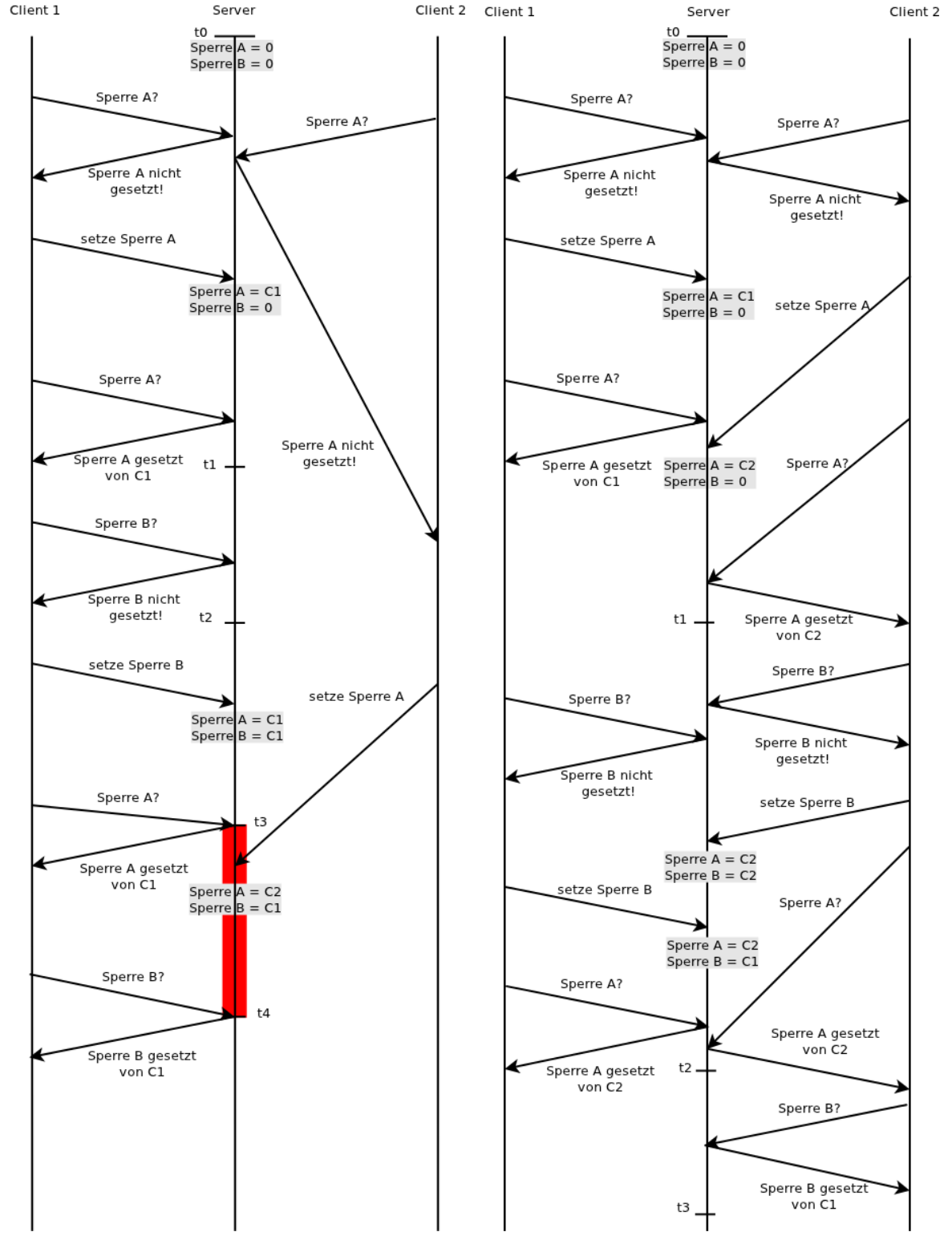
Für das Sperrverfahren, mit dem Dateien, Verzeichnisse und Journal-Abschnitte für den exklusiven Zugriff gesperrt werden können, wird die Annahme einer maximalen Paketlaufzeit wieder fallen gelassen. Der Algorithmus 4.3 zeigt den Ablauf des Sperrverfahrens. Es wird eine weitere Sperre eingeführt, sodass nun zwei Sperren pro Objekt existieren. Beide Sperren müssen von einem Client gehalten werden, um erfolgreich die Objektsperre beantragt zu haben. Ein Beispielablauf des Sperrprotokolls ist in Abbildung 4.4(a) zu sehen. Zu Beginn versucht ein Client Sperre A zu setzen, indem er dort seine Client-ID einträgt. Wenn Sperre A erfolgreich gesetzt wurde, versucht er auch Sperre B zu setzen, sofern dies nicht bereits durch einen anderen Client geschehen ist. Am Ende wird überprüft, ob er immer noch Sperre A hält und auch Sperre B wird nochmals überprüft. Sollte ein Client bei der Überprüfung am Ende des Protokolls für beide Sperren eine erfolgreiche Rückmeldung bekommen, so betrachtet er diese Objektsperre als erfolgreich reserviert und setzt ein Flag,

Algorithmus 4.3 Modifizierter Algorithmus zur Objektsperrung

```
 1: procedure getLockOnObject( )
 2:     read Lock_A from server
 3:     if (Lock_A ≠ 0 AND Lock_A ≠ own Client-ID) then
 4:         call ClearPossibleDeadlock() and return
 5:     end if
 6:     set Lock_A
 7:
 8:     read Lock_A from server
 9:     if (Lock_A ≠ own Client-ID) then
10:         call ClearPossibleDeadlock() and return
11:     end if
12:
13:     read Lock_B from server
14:     if (Lock_B ≠ 0 AND Lock_B ≠ own Client-ID) then
15:         call ClearPossibleDeadlock() and return
16:     end if
17:     set Lock_B
18:
19:     read Lock_A from server
20:     if (Lock_A ≠ own Client-ID) then
21:         call ClearPossibleDeadlock() and return
22:     end if
23:     read Lock_B from server
24:     if (Lock_B ≠ own Client-ID) then
25:         call ClearPossibleDeadlock() and return
26:     end if
27:
28:     set owner flag
29: end procedure
```

das ihn als Besitzer der Objektsperre ausweist. Ein eventuelles Ändern der einzelnen Sperren durch einen anderen Client hat darauf keine Auswirkung, da dieser später bei seiner Überprüfung feststellen wird, dass er nicht mehr beide Sperren hält. Nach Beenden der Operationen auf dem Objekt setzt der Client die Sperren wieder zurück. Anschließend kann ein anderer Client wieder versuchen, die Objektsperre zu erlangen.

Die Korrektheit des Reservierungsprotokolls kann durch Ausschlussverfahren bewiesen werden: Zu Beginn sind beide Sperren nicht gesetzt. O.b.d.A. wird angenommen, dass Client 1 zuerst Sperre A und Sperre B setzt und, wie in Abbildung 4.4(a) zu sehen, zum Zeitpunkt t_3 nochmals Sperre A überprüft. Nun kann der zweite Client zu unterschiedlichen Zeitpunkten ebenfalls beginnen und versuchen, die Sperre zu erlangen. Wenn der zweite Client bis zum Zeitpunkt t_3 Sperre A setzt, bemerkt dies Client 1 spätestens bei der zweiten Überprüfung von Sperre A direkt nach t_3 und zieht sich zurück. Wenn der zweite Client erst



(a) Ablauf 1 (b) Ablauf 2

Abbildung 4.4.: Beispielhafte Abläufe des Sperrverfahrens

nach dem Zeitpunkt t_3 Sperre A setzt, so hat Client 1 bereits Sperre B gesetzt, was der zweite Client auch durch Überprüfung bemerken wird und sich dementsprechend zurückzieht. Damit ist ausgeschlossen, dass zwei Clients gleichzeitig der Annahme sind, dass sie sich die Objektsperre erfolgreich gesichert haben.

Aufgrund der unterschiedlichen Verbindungsarten, vor allem bei mobilen Clients, kann es zu sehr unterschiedlichen Paketlaufzeiten kommen, sodass bei zwei gleichzeitig agierenden Clients eine fast beliebige Reihenfolge der am Server ankommenden Nachrichten entstehen kann. Das Gleiche gilt für Pakete, die vom Server zu einem der Clients gesendet werden. Es wird die Annahme getroffen, dass die Nachrichten in FIFO-Ordnung gesendet und empfangen werden, also dass zwei Nachrichten entsprechend ihrem Versandzeitpunkt auch in dieser Reihenfolge beim Empfänger ankommen. Die chronologisch geordnete Reihenfolge der Nachrichten ist notwendig, da andernfalls die Überprüfungen der Zustände der einzelnen Sperren ausgehebelt werden könnten und das Protokoll nicht mehr für die Konsistenz des Dateisystems garantieren könnte. Die Annahme setzt entweder voraus, dass die zugrunde liegenden Speicherprotokolle TCP als Transportprotokoll verwenden, da dort diese Funktionalität bereits implementiert ist, oder das Speicherprotokoll selbst muss die FIFO-Ordnung gewährleisten.

Deadlock-Beseitigung

Das modifizierte Sperrverfahren sichert immer höchstens einem Client die Objektsperre zu. Es kann allerdings der Fall eintreten, dass zwei Clients jeweils eine der beiden Einzelsperren erwerben und sich anschließend zurückziehen. Das Resultat ist ein Deadlock der Objektsperre. Durch Verwendung der Annahme einer maximalen Nachrichtenlaufzeit kann dieser wieder aufgelöst werden.

Eine mögliche Nachrichtenreihenfolge, die einen Deadlock verursacht, ist in Abbildung 4.4(b) zu sehen. Zu Beginn überprüfen beide Clients den Status von Sperre A, die nicht gesetzt ist. Beide Clients versuchen Sperre A zu setzen und überprüfen anschließend, ob das Sperren erfolgreich war. Durch die lange Übertragungszeit der Nachricht von Client 2 zum Setzen der Sperre A hat Client 1 bereits die Sperre schon gesetzt und erneut den Status der Sperre gelesen. Dadurch überschreibt Client 2 Sperre A mit seiner Client-ID. Anschließend lesen beide Clients den Status von Sperre B, die zu diesem Zeitpunkt noch nicht gesetzt ist. Daraufhin versuchen beide, Sperre B zu setzen, wobei die Nachricht von Client 2 zum Setzen der Sperre vor der von Client 1 am Server ankommt. Somit hält Client 2 für kurze Zeit Sperre B, die allerdings kurz darauf von Client 1 überschrieben wird. Zum Schluss überprüfen beide Clients jeweils den Status von Sperre A und Sperre B. Beide Clients stellen fest, dass sie nicht erfolgreich beide Sperren gesetzt haben und ziehen sich zurück. Somit ist ein Deadlock entstanden, der es keinem Client mehr ermöglicht, dieses Objekt für sich zu sperren.

Sobald ein Client bei der Reservierung einer Objektsperre einen möglichen Deadlock erkennt, führt er den Algorithmus 4.4 zur Deadlock-Erkennung und -Beseitigung aus. Dabei wird dann untersucht, ob ein Deadlock einer Objektsperre vorliegt. Wenn dies der Fall

Algorithmus 4.4 Algorithmus zur Deadlock-Erkennung und -Auflösung

```
 1: procedure ClearPossibleDeadlock( )
 2:     read owner flag from server
 3:     if (owner flag is set) then
 4:         object is not deadlocked -> return
 5:     end if
 6:     read Lock_A from server
 7:     read Lock_B from server
 8:
 9:     if (Locks are not set OR Lock_A = Lock_B) then
10:         object is not deadlocked -> return
11:     end if
12:
13:     waiting time = (2 * t_max + max clock skew) - (time since modification of locks)
14:
15:     if (Client was involved in deadlock) then
16:         wait for (waiting time + random backoff time)
17:     else if (Client was NOT involved in deadlock ) then
18:         if (waiting time > 0) then
19:             leave deadlock unhandled and return
20:         end if
21:     end if
22:
23:     lock parent directory
24:     if (parent directory cannot be locked) then
25:         leave deadlock unhandled and return
26:     end if
27:
28:     if (Locks were modified during waiting time) then
29:         object is not deadlocked -> return
30:     end if
31:
32:     clear deadlock by setting Lock_A, Lock_B and owner flag with own Client-ID
33: end procedure
```

ist, versucht der Client den Deadlock durch Setzen der beiden einzelnen Sperren und des Besitzer-Flags aufzuheben. Ein Deadlock einer Objektsperre liegt genau dann vor, wenn das Besitzer-Flag nicht gesetzt ist und die beiden einzelnen Sperren unterschiedliche eingetragene Client-IDs besitzen. Allerdings muss ein Client den Status der Objektsperre zwei Mal im Abstand von mindestens $2 * t_{\max}$ überprüfen, um sicher den Zustand der Objektsperre bestimmen zu können. Der Grund hierfür sind Nachrichten, die sich aufgrund von hoher Verbindungslatenzen noch im Transit befinden können und den Zustand der Sperre ändern können. Nur wenn sich der Zustand der Sperre nach Ablauf dieser Zeit nicht geändert hat, kann sicher von einem Deadlock ausgegangen werden. Es ist wichtig, dass ein Client nicht fälschlicherweise einen Deadlock annimmt und diesen zu beseitigen versucht, da er dabei die regulär erworbene Objektsperre eines zweiten Clients überschreiben könnte und somit die Gefahr von Inkonsistenzen entstünde.

Der Algorithmus zur Erkennung und Beseitigung eines eventuellen Deadlocks wird aufgerufen, wenn beim Überprüfen des Status einer Einzelsperre festgestellt wird, dass ein anderer Client diese Sperre bereits gesetzt hat. Zuerst wird überprüft, ob das Besitzer-Flag gesetzt ist. Ist dies der Fall, kann kein Deadlock vorliegen, da die Objektsperre einen Besitzer ausweist. Anschließend wird der Zustand der beiden einzelnen Sperren gelesen und überprüft. Ist eine der beiden nicht gesetzt oder sind beide der gleichen Client-ID zugeordnet, liegt ebenfalls noch kein Deadlock vor und die Erkennung wird beendet. Andernfalls wird die Zeit berechnet, die noch gewartet werden muss, bis der Zustand eines Deadlocks sicher bestimmt werden kann. Dazu werden die Zeitstempel der letzten Änderung der Einzelsperren gelesen und die seit diesem Zeitpunkt verstrichene Zeit berechnet. Dabei wird der Stempel der Sperre verwendet, die als letztes geändert wurde. Die berechnete Zeit wird von der mindestens benötigten Wartezeit, die $2 * t_{\max}$ beträgt, abgezogen. Da die Systemuhren der Clients variieren können und dennoch sichergestellt sein muss, dass frühestens nach $2 * t_{\max}$ ein Deadlock angenommen wird, muss zu der berechneten Wartezeit noch eine Konstante addiert werden. Diese gibt die maximale Abweichung aller Systemuhren der das System nutzenden Clients von der Systemuhr des Servers an. Diese Maßnahme ist notwendig, da die Zeitspanne seit der letzten Änderung einer Sperre mit Hilfe des Zeitstempels berechnet wird, der entsprechend der Systemzeit auf dem Server gesetzt wird. Bei einem Client mit einer Systemuhr, die einen negativen Drift besitzt, könnte ohne die zusätzlich addierte Zeit zu früh ein Deadlock angenommen werden. Der Wert der Konstante wird vom Nutzer angegeben. Wenn dieser dafür Sorge trägt, dass alle Uhren im System synchronisiert sind, kann er einen sehr niedrigen Wert für die Konstante wählen, was die Deadlock-Erkennung und -Beseitigung beschleunigt.

Wenn ein Client beim Versuch, die Objektsperre zu setzen, eine der beiden einzelnen Sperren gesetzt hat und somit zum eventuellen Deadlock beigetragen hat, wartet er anschließend für die berechnete Zeitspanne. Danach überprüft er die Sperren nochmals. Falls er dagegen bereits den Fall vorfand, dass die einzelnen Sperren schon gesetzt waren, bricht er den Algorithmus ab, sofern er für die sichere Deadlock-Erkennung warten müsste. Somit werden Clients, die nicht an der Verursachung des potenziellen Deadlocks beteiligt waren, nicht versuchen, diesen zu beheben, sofern er noch nicht älter als die Mindestwartezeit ist. Dadurch wird die Zahl der Clients, die sich bei der Deadlock-Behebung gegenseitig behindern könnten, nicht noch zusätzlich erhöht, sondern beschränkt sich auf die am Deadlock beteiligten

Clients. Falls während der Wartezeit die Sperren verändert wurden, wird die Deadlock-Erkennung abgebrochen. Um zu bestimmen, welcher von möglicherweise mehreren Clients den Deadlock beseitigen und die Sperre für sich reservieren darf, ist eine Koordination der Clients nötig. Dies geschieht durch das Sperren des Verzeichnisses, in dem sich das Objekt befindet. Nur der Client, der einen Deadlock erkannt und das Elternverzeichnis erfolgreich gesperrt hat, darf den Deadlock auch aufheben. Falls der Deadlock bei einem Reservierungsvorgang aufgetreten ist, bei dem mehrere Clients parallel versucht haben die Sperre zu beantragen, so werden diese Clients anschließend auch zur gleichen Zeit versuchen, das Eltern-Verzeichnis zu sperren. Um die Beantragung der Verzeichnissperre der einzelnen Clients zu desynchronisieren, wird zu der berechneten Wartezeit eine kurze, zufällige Zeitspanne addiert. Dadurch soll die Wahrscheinlichkeit eines weiteren Deadlocks verringert werden. Es kann jedoch bereits ein Deadlock der Sperre des Elternverzeichnisses bestehen oder trotz der Desynchronisation auftreten. In diesem Fall wird der Algorithmus rekursiv für eine Verzeichnisebene höher ausgeführt. Dieser Vorgang bricht spätestens beim Wurzelverzeichnis ab, bei dem kein Elternverzeichnis mehr gesperrt werden kann, um den Deadlock eines enthaltenen Verzeichnisses oder einer Datei aufzuheben. In diesem Fall ist es nötig, dass der einfache, aber langsamere Sperralgorithmus verwendet wird, der in Algorithmus 4.2 beschrieben wurde. Hier ist zusätzlich zu der Wartezeit zur Deadlock-Erkennung eine Wartezeit nötig, um das Resultat einer beantragten Sperre bestimmen zu können, weshalb er nur bei Deadlocks des Wurzelverzeichnisses verwendet wird.

4.3. Details des Dateisystems

CloudFS kann sowohl auf blockbasierten als auch auf dateibasierten Online-Speichern ausgeführt werden. In dieser Arbeit werden ausschließlich dateibasierte Speicher betrachtet. CloudFS wurde als Journaling-Dateisystem entworfen, damit bei Auftreten von Inkonsistenzen stets eine konsistente Version des Dateisystems rekonstruiert werden kann. Eine strukturelle Übersicht des Systems auf einem dateibasiertem Speicher ist in Abbildung 4.5 zu sehen. Der gesamte Speicherplatz ist mit dem Journal-Bereich und dem Datenbereich in zwei große Teile partitioniert, die jeweils durch einen eigenen Ordner repräsentiert werden. Der Datenbereich beinhaltet eine normale Ordnerstruktur, in der der Nutzer Dateien und Verzeichnisse ablegen kann. Der Journal-Bereich ist in eine festgelegte Anzahl von Abschnitten unterteilt, wobei für jeden Abschnitt ein eigener Ordner existiert. Jeder Journal-Abschnitt enthält Metadaten-Dateien und Ordner, die für die Verwaltung des jeweiligen Abschnitts notwendig sind. Dazu gehören:

Sperre A / Sperre B / Besitzer / Alternativsperre Diese vier Dateien sind für die Reservierung eines Abschnitts durch einen Client notwendig. Die Funktionsweise und Nutzung dieser Dateien für das Sperrverfahren werden in Abschnitt 4.2.2 beschrieben.

Änderungsliste Für jede Änderung, die ein Client im Datenbereich vornehmen will, muss er zuerst in seinem Journal-Abschnitt einen Journal-Eintrag anlegen. In dieser Liste werden alle angelegten Einträge gespeichert.

Änderungsordner In diesem Ordner werden Dateien zwischenspeichert, die im Datenbereich neu angelegt werden sollen. Zum anderen werden hier auch Delta Update-Dateien abgelegt, die Daten enthalten, die in bereits bestehende Dateien übernommen werden sollen.

Jeder Client muss vor der Nutzung des Dateisystems zuerst einen dieser Bereiche für sich reservieren. Auf diesen Abschnitt kann dann exklusiv nur er schreibend zugreifen. Dort werden alle Schreibänderungen, die er an Dateien und Verzeichnissen vornehmen möchte, zwischengespeichert. Anschließend kann er die Einträge direkt ausschreiben und in den Datenbereich übertragen oder dies wahlweise von einem anderen Client erledigen lassen. Diese Funktionalität ist speziell bei mobilen Clients von Vorteil, bei denen das Ausschreiben aufgrund der Verbindungseigenschaften sehr lange dauern könnte. Die Nutzung eines Journals ist für CloudFS unabdingbar, da speziell bei Mobilfunkverbindungen eine erhöhte Wahrscheinlichkeit eines Verbindungsabbruchs besteht. Bei einem Schreibvorgang im

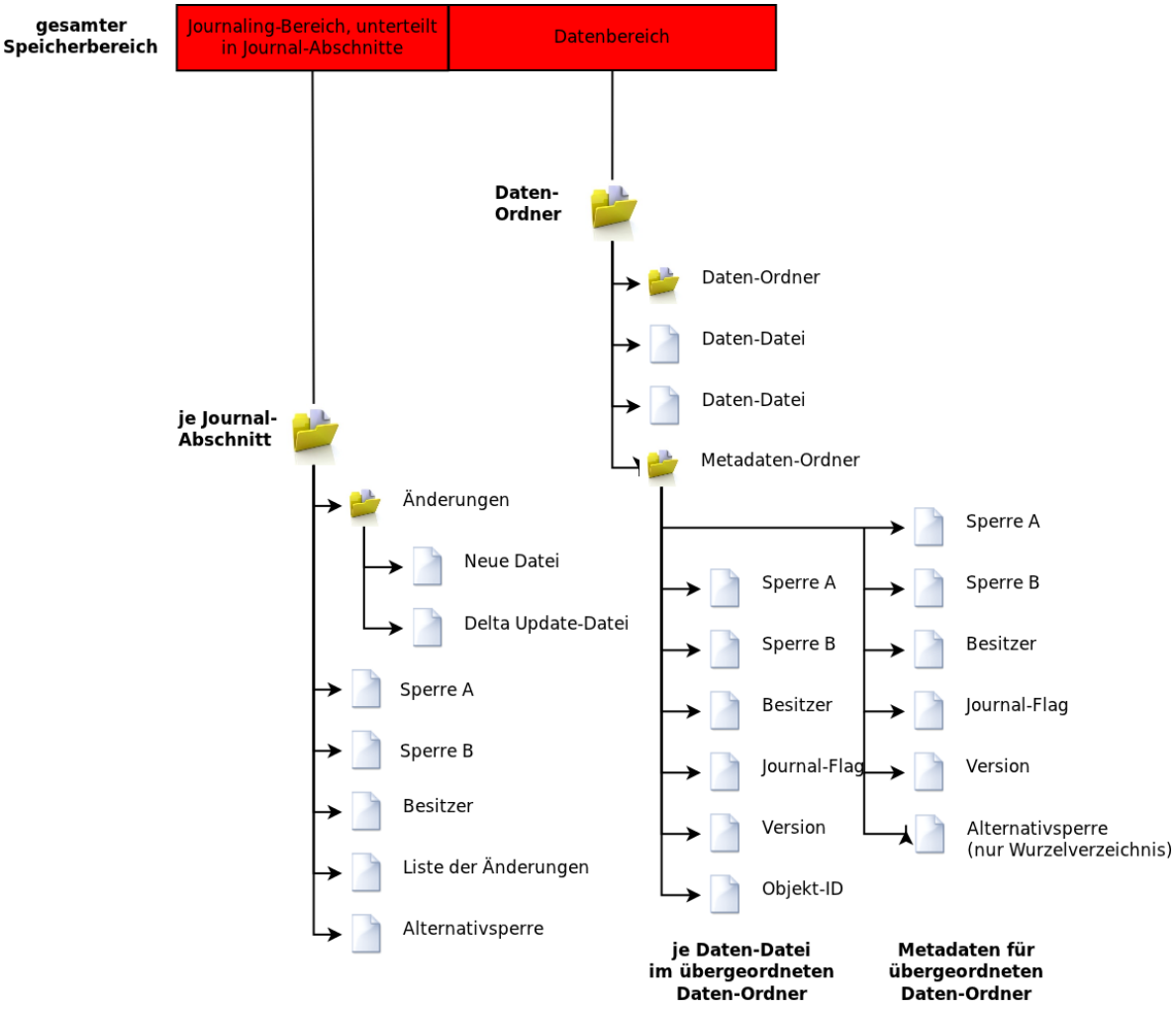


Abbildung 4.5.: Struktur des Dateisystems bei dateibasierten Speichern

Datenbereich ohne Journal wäre die zu ändernde Datei bei Abbruch der Verbindung in einem inkonsistenten Zustand, der nicht behoben werden kann. Ein Datenverlust wäre die mögliche Folge. Durch Nutzung der Journal-Funktion kann jedoch eine konsistente Version der Daten rekonstruiert werden.

Der Datenbereich beinhaltet neben den vom Nutzer angelegten Daten in jedem Verzeichnis einen für den Nutzer nicht sichtbaren Metadaten-Ordner. Darin werden für die Verwaltung der Dateien und Ordner benötigte Metadaten gespeichert. Metadaten wie Größe einer Datei oder das Änderungsdatum, die der zugrunde liegende Online-Speicher bereits direkt in einer Datei speichert und zur Verfügung stellt, werden direkt von diesem übernommen. Jeder Metadaten-Ordner enthält die Metadaten des Verzeichnisses, in dem er sich befindet, und die der darin enthaltenen Dateien. Dazu gehören:

Sperre A / Sperre B / Alternativsperre / Besitzer Diese Dateien sind für die Sperrung des jeweiligen Ordners oder der jeweiligen Datei notwendig und werden vom Sperrverfahren verwendet. Wenn das zugehörige Verzeichnis das Wurzelverzeichnis ist, befindet sich im Metadaten-Ordner zusätzlich zu den normalen Sperr-Dateien die Alternativsperre, mit deren Hilfe ein Deadlock des Wurzelverzeichnisses aufgehoben werden kann.

Version Diese Datei repräsentiert einen Versionszähler der Datei oder des Verzeichnisses. Nach jeder übernommenen Änderung aus dem Journal in den Datenbereich wird der Zähler inkrementiert.

Journal-Flag Dieses Flag gibt an, ob sich in einem der Journal-Abschnitte Änderungen für das Objekt befinden, die noch nicht in den Datenbereich übernommen wurden.

Objekt-ID Jeder Datei wird eine eindeutige ID zugewiesen. Mit Hilfe dieser ID werden Einträge im Journal den Dateien im Datenbereich zugeordnet.

Für jede Dateioperation auf einem CloudFS-Speicher müssen je nach Art der Operation verschiedene Schritte wie etwa das Setzen einer Objektsperre oder das Anlegen eines Journal-Eintrags ausgeführt werden. In Kapitel 4.3.1 werden die von CloudFS zur Verfügung gestellten Dateisystemfunktionen dargestellt. Der prinzipielle Ablauf ist bei den meisten Operationen der gleiche: Zuerst wird überprüft, ob für die betreffende Datei oder das betreffende Verzeichnis Journal-Einträge existieren, die noch nicht in den Datenbereich übernommen wurden. Wenn dies der Fall ist und der Client das Ausschreiben von Journal-Einträgen unterstützt, wird versucht die entsprechende Sperre zu setzen und die Datei oder das Verzeichnis im Datenbereich auf den aktuellen Stand zu bringen. Anschließend wird, falls notwendig und noch nicht beim Ausschreiben von Journal-Einträgen geschehen, die Objektsperre gesetzt. Falls die Sperre nicht gesetzt werden konnte, wird die Operation mit einer Fehlermeldung abgebrochen. Nach dem Setzen der Sperre kann die eigentliche Dateioperation erfolgen, indem ein neuer Journal-Eintrag angelegt wird und eventuelle Änderungen der Datei auf den Online-Speicher hochgeladen werden. Je nach Nutzereinstellung werden die Änderungen sofort in den Datenbereich übertragen oder bleiben unausgeschrieben. Beim Ausschreiben wird die Datei oder das Verzeichnis im Datenbereich entsprechend des Journal-Eintrags verändert, die Versionsnummer inkrementiert und anschließend der Journal-Eintrag gelöscht. Am Ende werden gesetzte Sperren wieder freigegeben und die Operation ist beendet.

Schreibänderungen einer Datei werden bis zum Schließen der Datei lokal auf der Festplatte des Clients in einem Cache-Verzeichnis gespeichert. Ebenso wird dieses Verzeichnis benutzt, wenn für eine Datei unausgeschriebene Journal-Einträge existieren und eine Leseoperation auf ihr ausgeführt werden soll. In diesem Fall wird die aktuelle Version der Datei im Cache rekonstruiert.

Jedem Client wird eine 128-Bit-ID zugewiesen, die ihn eindeutig im System identifiziert. Mit deren Hilfe wird dann ein Journal-Abschnitt oder eine Datei- oder Verzeichnissperre einem Client zugeordnet. Im Normalfall gibt ein Client vor Beendigung seiner Nutzung des Dateisystems alle von ihm gehaltenen Datei- und Verzeichnissperren sowie den reservierten Journalbereich wieder auf. Sollte der Fall eintreten, dass ein Client beispielsweise aufgrund von Verbindungsproblemen das System ungeordnet verlässt, so bleiben auch gesetzte Sperren im System erhalten und die betroffenen Dateien und Verzeichnisse sind nur noch für Leseoperationen, aber nicht mehr für Änderungsoperationen verfügbar. Grund dafür ist die fehlende Möglichkeit von außenstehenden Clients zwischen reiner Inaktivität und einem Verbindungsabbruch zu unterscheiden. Nun besteht die Möglichkeit, dass der betroffene Client nach einer gewissen Zeit zurückkehrt und selbst die gesetzten Sperren wieder aufgibt. Es wurde ebenfalls ein Administrationstool entwickelt, das alle Sperren eines bestimmten Clients im System wieder freigibt. Die Nutzung des Tools ist für den Fall gedacht, dass ein Client das System in absehbarer Zeit nicht mehr nutzen wird, aber trotzdem schreibender Zugriff auf die gesperrten Dateien und Verzeichnisse möglich sein soll. Die Ausführung dieses Programms obliegt dem Administrator des Systems, da sichergestellt sein muss, dass der betroffene Client zum Zeitpunkt der Ausführung das Dateisystem nicht nutzt. Andernfalls könnten Inkonsistenzen auftreten. Das Freigeben der Sperren des betreffenden Clients kann nicht automatisch geschehen, weil ein anderer Client nicht entscheiden kann, ob der Client, der die Sperren hält, nur inaktiv ist oder das System verlassen hat. Zum Ausführen des Administrationstools muss der Administrator den Client angeben, dessen Sperren freigegeben werden sollen. Optional kann der Nutzer einem Client zur leichteren Identifikation einen frei wählbaren Bezeichner (zum Beispiel „Heim-PC“) zuordnen, der jedoch im System eindeutig sein muss. Der Administrator kann bei der Angabe eines Clients auf diesen Namen zurückgreifen und muss nicht die eher unhandliche 128-Bit-Client-ID verwenden.

Die für jede Datei eindeutige Objekt-ID wird beim Erstellen der Datei generiert. Sie besteht aus der Konkatenation der ID des erstellenden Clients und dem aktuellen Zeitstempel. Durch die Verwendung der Client-ID ist ausgeschlossen, dass zwei verschiedene Clients die gleiche Objekt-ID generieren. Der Zeitstempel sorgt für Eindeutigkeit der IDs aller erstellten Dateien eines bestimmten Clients. Allerdings muss ausgeschlossen werden, dass Zeitstempel mehrfach verwendet werden, wenn mehrere Dateien kurz hintereinander erstellt werden. Dazu wird der zuletzt genutzte Zeitstempel gespeichert. Wenn bei der Generierung der nächsten Objekt-ID der selbe Zeitstempel verwendet werden müsste, wird dieser inkrementiert, damit die Eindeutigkeit gewahrt bleibt.

OPERATION	FUNKTION
CREATE	Erstellt eine neue Datei und öffnet sie
OPEN	Öffnet eine existierende Datei
READ	Liest aus einer geöffneter Datei
WRITE	Schreibt in eine geöffnete Datei
FSYNC	Synchronisiert eine geöffnete Datei
RELEASE	Schließt eine geöffnete Datei
RENAME	Verschiebt eine Datei oder benennt sie um
UNLINK	Löscht eine Datei
UTIMENS	Setzt die Zeitstempel einer Datei
CHOWN	Ändert den Besitzer einer Datei
CHMOD	Ändert die Rechte einer Datei
TRUNCATE	Ändert die Größe einer Datei
MKDIR	Erstellt ein neues Verzeichnis
OPENDIR	Öffnet ein existierendes Verzeichnis
READDIR	Liest den Inhalt eines geöffneten Verzeichnis
RELEASEDIR	Schließt ein geöffnetes Verzeichnis
RMDIR	Löscht ein Verzeichnis
RENAME	Verschiebt ein Verzeichnis oder benennt es um
GETATTR	Liest die Attribute einer Datei oder eines Verzeichnisses

Tabelle 4.1.: Verfügbare Dateisystemoperationen in CloudFS

4.3.1. Verfügbare Dateisystemoperationen in CloudFS

In diesem Abschnitt werden die Operationen vorgestellt, die CloudFS zur Nutzung des Dateisystems anbietet. Dazu gehören insgesamt 19 Funktionen, mit denen Dateien und Verzeichnisse gelesen und bearbeitet werden können. Eine Übersicht der verfügbaren Dateisystemoperationen ist in Tabelle 4.1 zu sehen.

CREATE - Datei erstellen

Beim Aufruf von CREATE zum Erstellen einer neuen Datei wird diese sofort und ohne Umweg über einen Journal-Eintrag im Datenbereich erstellt. CREATE stellt somit eine Ausnahme dar, da die meisten anderen Funktionen die Änderungen zuerst im Journal ablegen. Wie in Algorithmus 4.5 zu sehen ist, wird zuerst überprüft, ob die Datei schon vorhanden ist. Wenn dies der Fall ist, wird CREATE abgebrochen und ein Fehler zurückgegeben. Es muss auch überprüft werden, ob die Datei eventuell durch einen Journal-Eintrag gelöscht worden ist, dieser aber noch nicht in den Datenbereich übernommen wurde. Sollte das der Fall sein, kann die Datei trotzdem erstellt werden. Wenn für die alte, bereits gelöschte Datei noch unausgeschriebene Journal-Einträge einer Umbenennungsoperation existieren, werden diese ausgeschrieben, sofern der Client das Ausschreiben von Journal-Einträgen unterstützt. Journal-Einträge, die nur die zu erstellende Datei betreffen, werden ignoriert. Um die Einträge auszuschreiben, muss die Sperre der Datei erworben werden.

Algorithmus 4.5 Datei erstellen

```
 1: procedure CREATE_FILE( )
 2:     if (file already exists AND file is not deleted by journal entry) then
 3:         return ERROR
 4:     end if
 5:     if (file does not yet exist) then
 6:         try to lock parent directory
 7:         if (Lock not successfully set) then
 8:             return ERROR
 9:         end if
10:         create empty file in content section
11:         release directory lock
12:     else if (journal flag is set AND writeback = TRUE) then
13:         try to lock file
14:         if (Lock successfully set) then
15:             writeback journal entries
16:         end if
17:     end if
18:     try to lock file
19:     if (Lock not successfully set) then
20:         return ERROR
21:     end if
22:     create journal entry
23:     create working copy in cache
24: end procedure
```

Wenn die Datei noch nicht existiert, muss zuerst die Sperre des Elternverzeichnisses erworben werden. Damit wird ausgeschlossen, dass nicht zwei Clients zur gleichen Zeit in einem Ordner eine Datei des gleichen Namens erstellen können. Anschließend wird dann eine leere Datei mit dem gewünschten Dateinamen samt Metadaten im zugehörigen Metadaten-Ordner erstellt. Die Verzeichnissperre kann nun wieder freigegeben werden. Wenn die Datei bereits besteht, muss sie lediglich als nicht mehr gelöscht markiert werden. Dies geschieht durch das Erstellen des Journal-Eintrag „New_prepare“, der außer den Standardwerten eines Journal-Eintrags keine weiteren Informationen enthält. Die neu erstellte Datei muss anschließend gesperrt werden, da der Aufruf von CREATE automatisch auch das Öffnen der Datei und das Vorbereiten eines eventuell folgenden Schreibvorgangs beinhaltet. Zum Schluss wird im Cache-Verzeichnis des Clients eine weitere Datei erstellt, in die eventuelle Änderungen geschrieben werden, bevor diese ins Journal übernommen werden.

OPEN - Datei öffnen

Um eine Datei bearbeiten oder lesen zu können, muss diese zuerst geöffnet werden. Der Ablauf von OPEN ist in Algorithmus 4.6 zu sehen. Wenn die Datei nicht im Datenbereich

existiert, wird ein Fehler zurückgegeben. Andernfalls werden, sofern vom Client unterstützt, alte Journal-Einträge ausgeschrieben. Wenn die Datei zum Schreiben geöffnet werden soll, muss sie zuerst gesperrt werden. Ist dies nicht möglich, kann die Datei nicht geöffnet werden und es wird ein Fehler zurückgegeben. Anschließend wird im Cache-Verzeichnis eine Datei erstellt, in der etwaige Änderungen gespeichert werden. Dabei kommen Delta Updates zum Einsatz, das heißt, es werden nur die geänderten Teile einer Datei gespeichert und nicht die gesamte Datei. Dadurch wird die Änderungsdatei klein gehalten, was gerade Clients mit einer langsameren Internetverbindung zugute kommt.

Algorithmus 4.6 Datei öffnen

```
1:  procedure OPEN_FILE( )
2:      if (file not found OR file is deleted by journal entry) then
3:          return ERROR
4:      end if
5:      if (journal flag is set AND writeback = TRUE) then
6:          try to lock file
7:          if (Lock successfully set) then
8:              writeback journal entries
9:          end if
10:     end if
11:     if (write access is demanded) then
12:         try to lock file
13:         if (Lock not successfully set) then
14:             return ERROR
15:         end if
16:         create local delta update file in cache
17:     end if
18:     if (read access is demanded AND journal entries are available) then
19:         reconstruct local working copy in cache
20:     end if
21: end procedure
```

Wenn die Datei nur gelesen werden soll, ist kein Sperren notwendig. Ein Lesevorgang greift standardmäßig direkt auf die Datei im Datenbereich zu. Wenn allerdings noch unausgeschriebene Journal-Einträge existieren, muss die aktuelle Version der Datei im Cache-Verzeichnis rekonstruiert werden. Jeder Lesezugriff findet dann auf der Cache-Datei statt.

READ - Datei lesen

Nachdem eine Datei geöffnet wurde, kann sie gelesen werden. Dazu wird READ (siehe Algorithmus 4.7) mit den benötigten Parametern wie Offset und Länge des Dateibereichs, der gelesen werden soll, aufgerufen. Wenn die aktuelle Version im Cache rekonstruiert werden musste, so wird der Lesevorgang dort ausgeführt. Andernfalls wird direkt von der Datei im Datenbereich gelesen.

Algorithmus 4.7 Datei lesen

```
1: procedure READ_FILE( )
2:     if (Local working copy is present) then
3:         read from working copy
4:     else
5:         read from content file
6:     end if
7: end procedure
```

WRITE - Datei schreiben

Durch den Aufruf von WRITE (siehe Algorithmus 4.8) können Änderungen in eine Datei geschrieben werden. Diese werden zunächst in einer lokalen Datei im Cache gespeichert, bevor sie beim Schließen der Datei ins Journal übertragen werden. Wenn eine Datei neu erstellt wurde, wird die vollständige Datei im Cache zwischengespeichert. Beim Bearbeiten einer bereits bestehenden Datei dagegen werden lediglich Delta Updates gespeichert.

Algorithmus 4.8 Datei schreiben

```
1: procedure WRITE_FILE( )
2:     write changes to local change file
3: end procedure
```

FSYNC - Datei synchronisieren

Manchmal kann es notwendig sein, Änderungen schon vor dem Schließen einer Datei ins Journal zu übertragen. Dazu wird beim Aufruf von FSYNC (siehe Algorithmus 4.9) zuerst geprüft, ob die Datei geändert wurde. Anschließend wird die Änderungsdatei ins Journal hochgeladen und ein „Write“-Journal-Eintrag wird erstellt. Dieser enthält neben den Standardwerten die Größe der Datei nach der Änderung.

Algorithmus 4.9 Datei synchronisieren

```
1: procedure SYNC_FILE( )
2:     if (File was changed) then
3:         upload new file / delta update file
4:         create journal entry
5:     end if
6: end procedure
```

RELEASE - Datei schließen

Nachdem die Bearbeitung einer Datei abgeschlossen ist, muss diese durch den Aufruf von RELEASE (siehe Algorithmus 4.10) wieder geschlossen werden. Sofern Änderungen an der Datei vorgenommen wurden, muss die Änderungsdatei ins Journal geladen werden. Bei einer neuen Datei wird ein „New“-Eintrag angelegt, bei einem Delta Update einer bereits existierenden Datei ein „Write“-Eintrag. Beide enthalten zusätzlich zu den Standardwerten eines Eintrags die Größe der Datei nach der Änderung. Falls die Sperre der Datei gehalten wird, muss sie wieder freigegeben werden. Zum Schluss werden noch die eventuell im Cache angelegten Dateien, die zum Lesen und Schreiben notwendig waren, gelöscht.

Algorithmus 4.10 Datei schließen

```
 1: procedure CLOSE_FILE( )
 2:     if (File was changed) then
 3:         upload new file / delta update file
 4:         create journal entry
 5:     end if
 6:     if (Write access was demanded) then
 7:         release file lock
 8:     end if
 9:     delete cache files
10: end procedure
```

RENAME - Datei umbenennen oder verschieben

Durch den Aufruf von RENAME (siehe Algorithmus 4.11) kann eine Datei sowohl umbenannt als auch verschoben werden. Eine Umbenennung liegt vor, wenn sich sowohl Quell- als auch Zieldatei im gleichen Verzeichnis befinden. Verweisen die angegebenen Pfade auf Dateien in unterschiedlichen Verzeichnissen, wird eine Verschiebung durchgeführt. Wenn die Zieldatei bereits existiert, wird diese ohne Nachfrage überschrieben.

Vor dem Ausführen der Operation muss zuerst geprüft werden, ob die umzubenennende Datei existiert. Anschließend wird versucht, die Sperre für die Datei zu erlangen. Gelingt dies nicht, wird der Vorgang abgebrochen und es wird ein Fehler zurückgegeben. Nun werden zuerst alte Journal-Einträge der Datei ausgeschrieben, sofern der Client diese Funktion unterstützt. Das weitere Vorgehen ist von der Zieldatei abhängig: Existiert diese bereits, so muss sie auch gesperrt werden. Kann entweder die Sperre von Quell- oder Zieldatei nicht reserviert werden, bricht RENAME mit einer Fehlermeldung ab. Danach werden gegebenenfalls alte Journal-Einträge der Zieldatei ausgeschrieben, wenn der Client dies unterstützt. Existiert die Zieldatei noch nicht, muss äquivalent zum Erstellen einer Datei zuerst das Elternverzeichnis gesperrt werden, damit die Zieldatei erstellt werden kann. Nachdem Quell- und Zieldatei erfolgreich gesperrt worden sind, werden für beide Dateien Journal-Einträge erstellt. Für die Quelldatei wird ein „Deleted_by_Rename“-Eintrag angelegt. Dieser enthält

Algorithmus 4.11 Datei umbenennen / verschieben

```
1:  procedure RENAME_FILE( )
2:      if (file not found OR file is deleted by journal entry) then
3:          return ERROR
4:      end if
5:      try to lock source file
6:      if (Lock not successfully set) then
7:          return ERROR
8:      end if
9:      if (journal flag of source file is set AND writeback = TRUE) then
10:         writeback journal entries of source file
11:     end if
12:     if (target file already exists) then
13:         try to lock target file
14:         if (Lock not successfully set) then
15:             return ERROR
16:         end if
17:         if (journal flag of target file is set AND writeback = TRUE) then
18:             writeback journal entries of target file
19:         end if
20:     else
21:         try to lock directory of target file
22:         if (Lock not successfully set) then
23:             return ERROR
24:         end if
25:         create empty target file
26:         release directory lock
27:     end if
28:     create journal entry for source file
29:     create journal entry for target file
30:     if (writeback = TRUE) then
31:         writeback made changes
32:     end if
33:     release file locks
34: end procedure
```

zusätzlich zu den Standardwerten den Pfad zur Zieldatei sowie die Versionsnummer des zugehörigen „Rename“-Eintrags der Zieldatei. Für die Zieldatei wird ein „Rename“-Eintrag angelegt. Dieser enthält analog zum „Deleted_by_Rename“-Eintrag neben den Standardwerten den Pfad zur Quelldatei sowie die Versionsnummer des „Deleted_by_Rename“-Eintrags der Quelldatei. Die Quelldatei wird durch den Journal-Eintrag als gelöscht markiert. Bei dessen Ausführung wird die Datei in eine für den Nutzer nicht sichtbare Datei umbenannt. Dementsprechend wird bei der Ausführung des Journal-Eintrags der Zieldatei die erstellte temporäre Datei in die Zieldatei umbenannt und der Umbenennungs- beziehungsweise Verschiebevorgang ist abgeschlossen.

UNLINK - Datei löschen

Durch die Ausführung von UNLINK (siehe Algorithmus 4.12) wird die als Parameter übergebene Datei gelöscht. Zum Löschen einer Datei muss die zugehörige Dateisperre gehalten werden, ansonsten wird ein Fehler zurückgegeben. Sofern der Client es unterstützt, werden vor dem eigentlichen Löschvorgang noch nicht ausgeschriebene Journal-Einträge in den Datenbereich übertragen. Anschließend wird ein neuer „Deleted“-Eintrag erstellt, der die Datei als gelöscht markiert und neben den Standardwerten keine weiteren Informationen enthält. Beim Ausschreiben des Eintrags wird die Datei samt zugehöriger Metadaten-Dateien gelöscht.

Algorithmus 4.12 Datei löschen

```
1:  procedure DELETE_FILE( )
2:      if (file not found OR file is deleted by journal entry) then
3:          return ERROR
4:      end if
5:      try to lock file
6:      if (Lock not successfully set) then
7:          return ERROR
8:      end if
9:      if (journal flag is set AND writeback = TRUE) then
10:         writeback journal entries
11:     end if
12:     create journal entry
13:     if (writeback = TRUE) then
14:         writeback made changes
15:     end if
16:     release file lock
17: end procedure
```

UTIMENS - Zeitstempel einer Datei setzen

Beim Aufruf von UTIMENS (siehe Algorithmus 4.13) zum Setzen der Zeitstempel einer Datei werden der letzte Zugriffszeitpunkt und der Zeitpunkt der letzten Modifikation der Datei auf einen vom Nutzer bestimmten Zeitpunkt gesetzt. Zuerst muss die zugehörige Dateisperre gesetzt werden, andernfalls wird ein Fehler zurückgegeben. Anschließend werden nicht ausgeschriebene Journal-Einträge in den Datenbereich übertragen, sofern der Client dies unterstützt. Danach wird ein „Utimens"-Eintrag im Journal angelegt, der neben den Standardwerten die zu setzenden Zeitstempel enthält. Der Eintrag wird sofort ausgeschrieben, wenn der Client das Zurückschreiben unterstützt.

Algorithmus 4.13 Zeitstempel einer Datei setzen

```
 1: procedure SET_FILE_TIMESTAMP( )
 2:     if (file not found OR file is deleted by journal entry) then
 3:         return ERROR
 4:     end if
 5:     try to lock file
 6:     if (Lock not successfully set) then
 7:         return ERROR
 8:     end if
 9:     if (journal flag is set AND writeback = TRUE) then
10:         writeback journal entries
11:     end if
12:     create journal entry
13:     if (writeback = TRUE) then
14:         writeback made changes
15:     end if
16:     release file lock
17: end procedure
```

CHOWN - Dateibesitzer ändern

Durch den Aufruf von CHOWN (siehe Algorithmus 4.14) kann der Nutzer den Besitzer und die Gruppe einer Datei ändern. Dazu muss der Client die zugehörige Dateisperre beantragen. Ist dies nicht erfolgreich, wird eine Fehlermeldung zurückgegeben. Nun werden noch nicht ausgeschriebene Journal-Einträge in den Datenbereich übertragen, sofern der Client dies unterstützt. Danach wird ein „Chown"-Eintrag im Journal angelegt. Dieser enthält neben den Standardwerten den neuen Besitzer und die Gruppe der Datei. Der Eintrag wird anschließend sofort ausgeschrieben, wenn der Client das Zurückschreiben unterstützt.

Algorithmus 4.14 Dateibesitzer ändern

```
1:  procedure CHANGE_OWNER_OF_FILE( )
2:      if (file not found OR file is deleted by journal entry) then
3:          return ERROR
4:      end if
5:      try to lock file
6:      if (Lock not successfully set) then
7:          return ERROR
8:      end if
9:      if (journal flag is set AND writeback = TRUE) then
10:         writeback journal entries
11:     end if
12:     create journal entry
13:     if (writeback = TRUE) then
14:         writeback made changes
15:     end if
16:     release file lock
17: end procedure
```

CHMOD - Dateirechte ändern

Mit dem Aufruf von CHMOD (siehe Algorithmus 4.15) kann der Nutzer einer Datei neue Zugriffsrechte zuweisen. Um die Operation durchführen zu können, muss zuerst die Dateisperre gesetzt werden. Ist dies nicht erfolgreich, bricht CHMOD ab und es wird eine Fehlermeldung zurückgegeben. Anschließend werden noch nicht ausgeschriebene Journal-Einträge in den Datenbereich übernommen, falls der Client das Ausschreiben von Einträgen unterstützt. Nun wird ein neuer „Chmod“-Eintrag im Journal angelegt, der zusätzlich zu den Standardwerten die neuen Dateirechte beinhaltet. Der Journal-Eintrag wird anschließend sofort ausgeschrieben, sofern der Client dies unterstützt.

TRUNCATE - Dateigröße ändern

Mit Hilfe von TRUNCATE (siehe Algorithmus 4.16) kann die Größe einer Datei verändert werden. Typischerweise wird dies genutzt, um eine Datei zu verkleinern. Es ist auch möglich, durch den Aufruf von TRUNCATE die Datei zu vergrößern, jedoch wird dies in der Praxis kaum eingesetzt. Um die Größenänderung der Datei durchführen zu können, muss zuerst die zugehörige Dateisperre gesetzt werden. Gelingt dies nicht, bricht TRUNCATE ab und es wird eine Fehlermeldung zurückgegeben. Anschließend werden noch nicht ausgeschriebene Journal-Einträge in den Datenbereich übertragen, sofern der Client dies unterstützt. Danach wird ein neuer „Truncate“-Eintrag angelegt. Er enthält neben den Standardwerten die neue Größe der Datei. Er wird sofort ausgeschrieben, falls der Client das Ausschreiben von Journal-Einträgen unterstützt.

Algorithmus 4.15 Dateirechte ändern

```
1: procedure CHANGE_FILE_MODE( )
2:     if (file not found OR file is deleted by journal entry) then
3:         return ERROR
4:     end if
5:     try to lock file
6:     if (Lock not successfully set) then
7:         return ERROR
8:     end if
9:     if (journal flag is set AND writeback = TRUE) then
10:        writeback journal entries
11:    end if
12:    create journal entry
13:    if (writeback = TRUE) then
14:        writeback made changes
15:    end if
16:    release file lock
17: end procedure
```

Algorithmus 4.16 Dateigröße ändern

```
1: procedure TRUNCATE_FILE( )
2:     if (file not found OR file is deleted by journal entry) then
3:         return ERROR
4:     end if
5:     try to lock file
6:     if (Lock not successfully set) then
7:         return ERROR
8:     end if
9:     if (journal flag is set AND writeback = TRUE) then
10:        writeback journal entries
11:    end if
12:    create journal entry
13:    if (writeback = TRUE) then
14:        writeback made changes
15:    end if
16:    release file lock
17: end procedure
```

MKDIR - Verzeichnis erstellen

Das Erstellen eines neuen Verzeichnisses durch MKDIR (siehe Algorithmus 4.17) wird, analog zum Erstellen einer neuen Datei, ohne den Umweg über das Journal sofort ausgeführt. Damit soll auch hier vermieden werden, dass zwei Clients zur gleichen Zeit das selbe Verzeichnis erstellen.

Algorithmus 4.17 Verzeichnis erstellen

```
1:  procedure CREATE_DIRECTORY( )
2:      if (directory already exists AND directory is not deleted by journal entry) then
3:          return ERROR
4:      end if
5:      if (journal flag is set AND writeback = TRUE) then
6:          try to lock directory
7:          if (Lock successfully set) then
8:              writeback journal entries
9:          end if
10:     end if
11:     if (directory does not yet exist) then
12:         try to lock parent directory
13:         if (Lock not successfully set) then
14:             return ERROR
15:         end if
16:         create empty directory in content section
17:         release parent directory lock
18:     else
19:         increment version counter
20:     end if
21: end procedure
```

Zuerst muss geprüft werden, ob das Verzeichnis bereits existiert. Sollte dies der Fall sein, so werden die zugehörigen Journal-Einträge überprüft, ob es zwischenzeitlich bereits gelöscht wurde, aber die Änderungen noch nicht in den Datenbereich übernommen wurden. Nur wenn das Verzeichnis noch nicht existiert oder bereits gelöscht wurde kann es neu erstellt werden. Andernfalls wird MKDIR abgebrochen und eine Fehlermeldung zurückgegeben. Sollten noch unausgeschriebene Journal-Einträge bestehen, die der Client ausschreiben könnte, so wird versucht das Verzeichnis zu sperren und diese dann in den Datenbereich zu übertragen.

Wenn das Verzeichnis bereits besteht, wird anschließend lediglich der Versionszähler inkrementiert. Das hat zur Folge, dass alle bisher bestehenden Journal-Einträge nun veraltet sind und das Verzeichnis nicht mehr durch diese gelöscht werden kann. Ist das Verzeichnis dagegen noch nicht vorhanden, muss analog zum Erstellen einer Datei zuerst das zugehörige Elternverzeichnis gesperrt werden, damit anschließend das neue Verzeichnis samt Metadaten erstellt werden kann.

OPENDIR - Verzeichnis öffnen

Um den Inhalt eines Verzeichnisses zu lesen, muss es zuerst durch den Aufruf von OPENDIR geöffnet werden. OPENDIR nimmt keine Änderungen an den Daten vor oder liest von diesen, sondern ist lediglich zu Verwaltungszwecken vorhanden. Dennoch muss OPENDIR vor dem Lesevorgang eines Verzeichnisses aufgerufen werden.

READDIR - Verzeichnis lesen

Mit Hilfe von READDIR werden die in dem betreffenden Verzeichnis enthaltenen Dateien und Verzeichnisse gelesen. Sollten beim Lesevorgang Dateien oder Verzeichnisse angetroffen werden, die durch unausgeschriebene Journal-Einträge bereits gelöscht sind, werden diese Änderungen in den Datenbereich übernommen, sofern der Client dies unterstützt.

RELEASEDIR - Verzeichnis schließen

Nachdem der Inhalt eines Verzeichnisses gelesen wurde, muss es durch RELEASEDIR wieder geschlossen werden. Analog zum Öffnen eines Verzeichnisses nimmt diese Funktion keine Änderungen an den Daten vor, sondern ist nur zu Verwaltungszwecken vorhanden.

Algorithmus 4.18 Verzeichnis löschen

```
1:  procedure DELETE_DIRECTORY( )
2:      if (directory doesnt exist OR directory is deleted by journal entry) then
3:          return ERROR
4:      end if
5:      try to lock subtree
6:      if (Locks not successfully set) then
7:          return ERROR
8:      end if
9:      create journal entry for all contained directories and files
10:     if (writeback = TRUE) then
11:         writeback made changes
12:     end if
13: end procedure
```

RMDIR - Verzeichnis löschen

Das Löschen eines Verzeichnisses durch RMDIR (siehe Algorithmus 4.18) ist eine deutlich aufwändigere Operation als das Löschen einer Datei, da für alle darin enthaltenen Dateien und Unterverzeichnisse Journal-Einträge erstellt werden müssen. Bevor dies jedoch geschehen kann, muss für jedes Objekt des Unterbaums die zugehörige Sperre gesetzt werden.

Erst wenn dies gelungen ist, werden für alle enthaltenen Dateien „Delete“-Einträge und analog für Verzeichnisse „Delete_Dir“-Einträge im Journal erstellt. Andernfalls werden bereits gesetzte Sperren wieder freigegeben und die Operation wird mit einer Fehlermeldung abgebrochen.

RENAMEDIR - Verzeichnis umbenennen/verschieben

Das Umbenennen oder Verschieben eines Verzeichnisses durch RENAMEDIR (siehe Algorithmus 4.19) ist äquivalent zu dem einer Datei. Das bedeutet, dass die Operation je nach Angabe des Zielverzeichnisses ein Umbenennen oder Verschieben des Verzeichnisses darstellt.

Zuerst wird überprüft, ob das Verzeichnis vorhanden ist und ein Journal-Eintrag vorliegt, der es bereits als gelöscht markiert hat. Sollte dies der Fall sein, wird RENAMEDIR abgebrochen und es wird eine Fehlermeldung zurückgegeben. Anschließend wird die Sperre für das umzubenennende Verzeichnis beantragt und eventuell vorhandene Journal-Einträge ausgeschrieben, sofern der Client dies unterstützt. Nun muss für das Zielverzeichnis, sofern es bereits existiert, ebenfalls die Sperre gesetzt werden. Wenn das Zielverzeichnis noch nicht vorhanden ist, wird versucht, das zugehörige Elternverzeichnis zu sperren. Anschließend wird das Zielverzeichnis erstellt und ebenfalls gesperrt. Sollte das Setzen einer der benötigten Sperren nicht erfolgreich sein, wird RENAMEDIR mit einer Fehlermeldung abgebrochen. Nachdem dann erfolgreich Quell- und Zielverzeichnis gesperrt worden sind, wird versucht, jedes Objekt der jeweiligen Unterbäume der beiden Verzeichnisse zu sperren. Wenn nicht alle enthaltenen Dateien und Verzeichnisse erfolgreich gesperrt werden konnten, werden schon gesetzte Sperren wieder freigegeben und RENAMEDIR wird mit einer Fehlermeldung abgebrochen. Nachdem die jeweiligen Objekte der Unterbäume gesperrt worden sind, werden im Anschluss Journal-Einträge erstellt (für Einträge bei Dateien siehe RENAME). Bei Verzeichnissen wird das Quellverzeichnis durch einen „Delete_Dir“-Eintrag im Journal als gelöscht markiert. Hier ist im Gegensatz zum Verfahren bei Dateien keine Umbenennung in ein unsichtbares, temporäres Verzeichnis notwendig, da das Verzeichnis selbst außer dem Namen keine Informationen beinhaltet. Beim Zielverzeichnis muss lediglich äquivalent zum Erstellen eines Verzeichnisses der Versionszähler erhöht werden, sofern es bereits existiert. Andernfalls sind keine weiteren Maßnahmen notwendig.

Algorithmus 4.19 Verzeichnis umbenennen / verschieben

```
1: procedure RENAME_DIRECTORY( )
2:     if (directory not found OR directory is deleted by journal entry) then
3:         return ERROR
4:     end if
5:     try to lock source directory
6:     if (Lock not successfully set) then
7:         return ERROR
8:     end if
9:     if (journal flag of source directory is set AND writeback = TRUE) then
10:        writeback journal entries of source directory
11:    end if
12:    if (target directory already exists) then
13:        try to lock target directory
14:        if (Lock not successfully set) then
15:            return ERROR
16:        end if
17:        increment version counter
18:    else
19:        try to lock parent directory of target directory
20:        if (Lock not successfully set) then
21:            return ERROR
22:        end if
23:        create empty target directory
24:        release parent directory lock
25:        try to lock target directory
26:        if (Lock not successfully set) then
27:            return ERROR
28:        end if
29:    end if
30:    try to lock subtree of source directory
31:    if (Locks not successfully set) then
32:        return ERROR
33:    end if
34:    try to lock subtree of target directory
35:    if (Locks not successfully set) then
36:        return ERROR
37:    end if
38:    create journal entries for subtree of source + target directory
39:    if (writeback = TRUE) then
40:        writeback made changes
41:    end if
42:    release all locks
43: end procedure
```

GETATTR - Attribute von Objekten abfragen

Vor dem Ausführen einer Datei- oder Verzeichnisoperation werden in der Regel die zugehörigen Attribute wie etwa die Dateigröße durch GETATTR (siehe Algorithmus 4.20) abgefragt. Außerdem kann der Aufruf von GETATTR dazu benutzt werden um herauszufinden, ob die Datei oder das Verzeichnis existiert.

Algorithmus 4.20 Attribute von Datei/Verzeichnis abfragen

```
1: procedure GET_ATTRIBUTES( )
2:     if (file not found OR file is deleted by journal entry) then
3:         return ERROR
4:     end if
5:     read attributes from file
6:     if (journal entries available) then
7:         reconstruct current attribute values from journal entries
8:     end if
9: end procedure
```

Zuerst wird überprüft, ob die Datei oder das Verzeichnis existiert. Wenn dies nicht der Fall ist, wird GETATTR mit einer Fehlermeldung abgebrochen. Andernfalls werden die aktuellen Attribute des Objekts aus dem Datenbereich gelesen. Anschließend wird überprüft, ob Journal-Einträge für die Datei oder das Verzeichnis vorliegen. Sollten Einträge gefunden werden, muss anhand dieser Einträge der aktuelle Wert der einzelnen Attribute rekonstruiert werden, die dann als Ergebnis von GETATTR zurückgegeben werden.

5. Implementierung

Für das zuvor entwickelte Dateisystem wurde ein Linux-Treiber implementiert. Dazu wurde das Framework Filesystem in Userspace (FUSE) verwendet, das die Entwicklung und Ausführung von Dateisystemen im Userspace ohne Root-Rechte ermöglicht. Im Folgenden werden die Implementierung und verwendete Parameter beschrieben.

5.1. Übersicht

Zur Implementierung des Dateisystems wurde FUSE verwendet, das auf der Programmiersprache C basiert. Die Implementierung ist in mehrere Komponenten aufteilt. Eine Übersicht über diese Komponenten ist in Abbildung 5.1 zu sehen. Die zentrale Komponente ist die Datei `main.c`. Sie enthält die Schnittstelle von CloudFS zu FUSE. Daneben existieren mehrere Komponenten, die Hilfsfunktionen implementieren. So enthält die Komponente `cloudfs_protocol.h` Funktionen, die direkt am Ablauf der Dateioperationen beteiligt sind. Diese werden von den Funktionen, die die nach außen sichtbaren Dateisystemoperationen implementieren, aufgerufen und realisieren zum Beispiel das Erstellen von Journal-Einträgen oder das Auschreiben dieser Einträge. Die Komponente `cloudfs_metadata_handling.h` implementiert die Zugriffe auf die Metadaten von Dateien und Verzeichnissen. Außerdem wird in dieser Komponente der Sperralgorithmus implementiert. Die Komponente `cloudfs_helper.h` stellt Hilfsfunktionen zur Verfügung, die oft verwendet werden, allerdings keinen direkten Bezug zum Ablauf einer Dateioperation haben. Es werden zum Beispiel oft Umwandlungen von Dateipfaden benötigt, die dort realisiert werden. Bei Programmstart muss eine Konfigurationsdatei eingelesen werden, die wichtige Parameter für den Ablauf der Dateioperationen enthält. Die Komponente `cloudfs_config_parser.h` enthält eine Funktion, die diese Aufgabe realisiert. Die eingelesenen Parameter werden neben einer Reihe weiterer globaler Variablen in der Komponente `cloudfs_global_params.h` den anderen Komponenten zur Verfügung gestellt. Schließlich wird die externe Komponente `uthash.h` [Han] verwendet, die den Datentyp einer Hashtabelle sowie den einer verketteten Liste zur Verfügung stellt, die in der Programmiersprache C nicht nativ zur Verfügung stehen.

Ein Beispiel einer Verzeichnisstruktur des Online-Speichers ist in Abbildung 5.2 zu sehen. Die Struktur wurde bereits im Kapitel 4 in Abbildung 4.5 vorgestellt. Hier ist nun die Benennung der einzelnen Dateien und Verzeichnisse zu sehen. Die einzelnen Journal-Abschnitte werden fortlaufend nummeriert benannt. In der Datei `Changelist` eines jeden Abschnitts werden die Journal-Einträge gespeichert. Änderungsdateien, die eine neue Datei beinhalten, werden nach dem Format „#NF#<Versionsnummer>#Pfad_zur_Datei“ benannt. Dabei werden

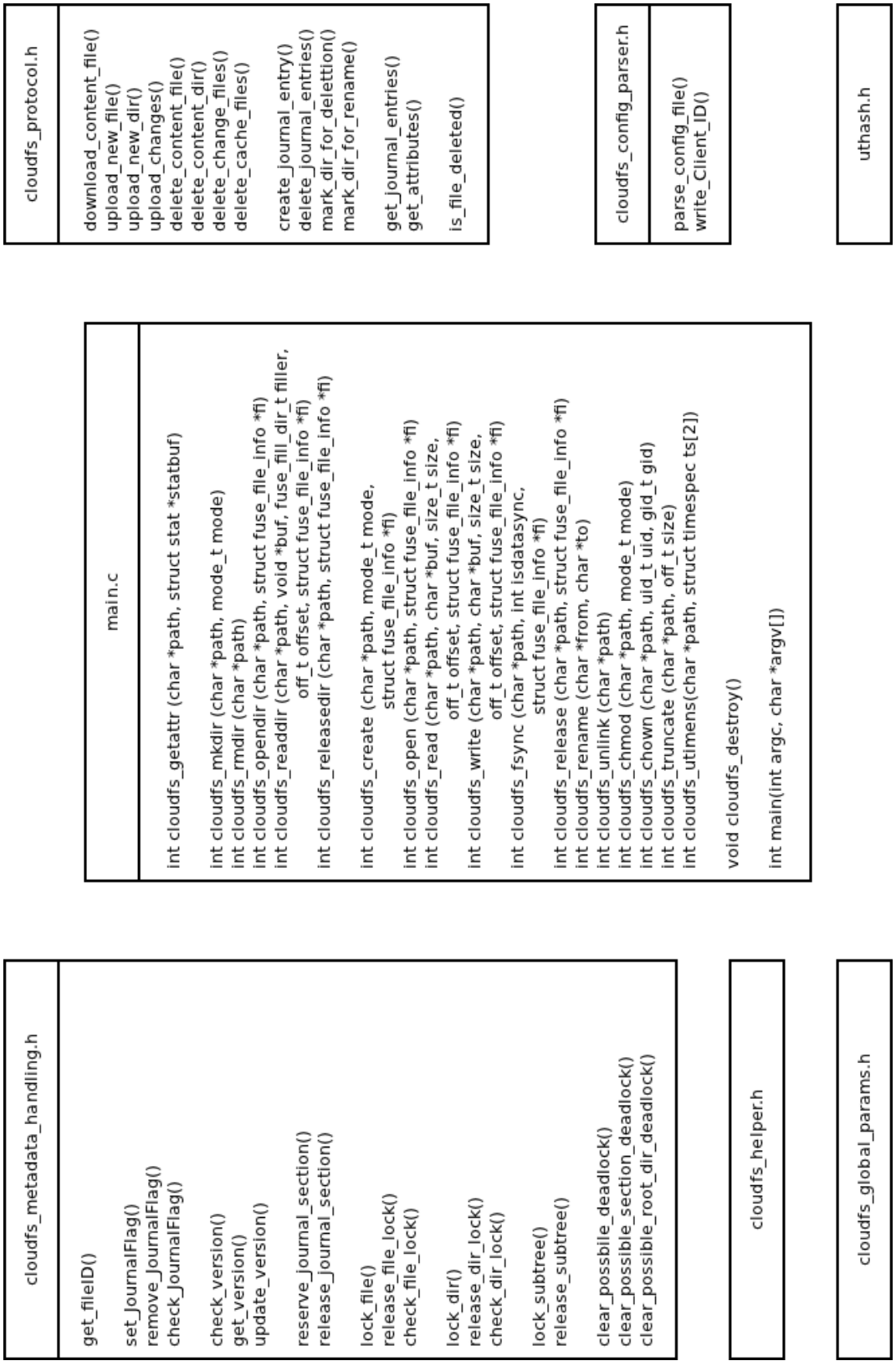


Abbildung 5.1.: Komponentenübersicht

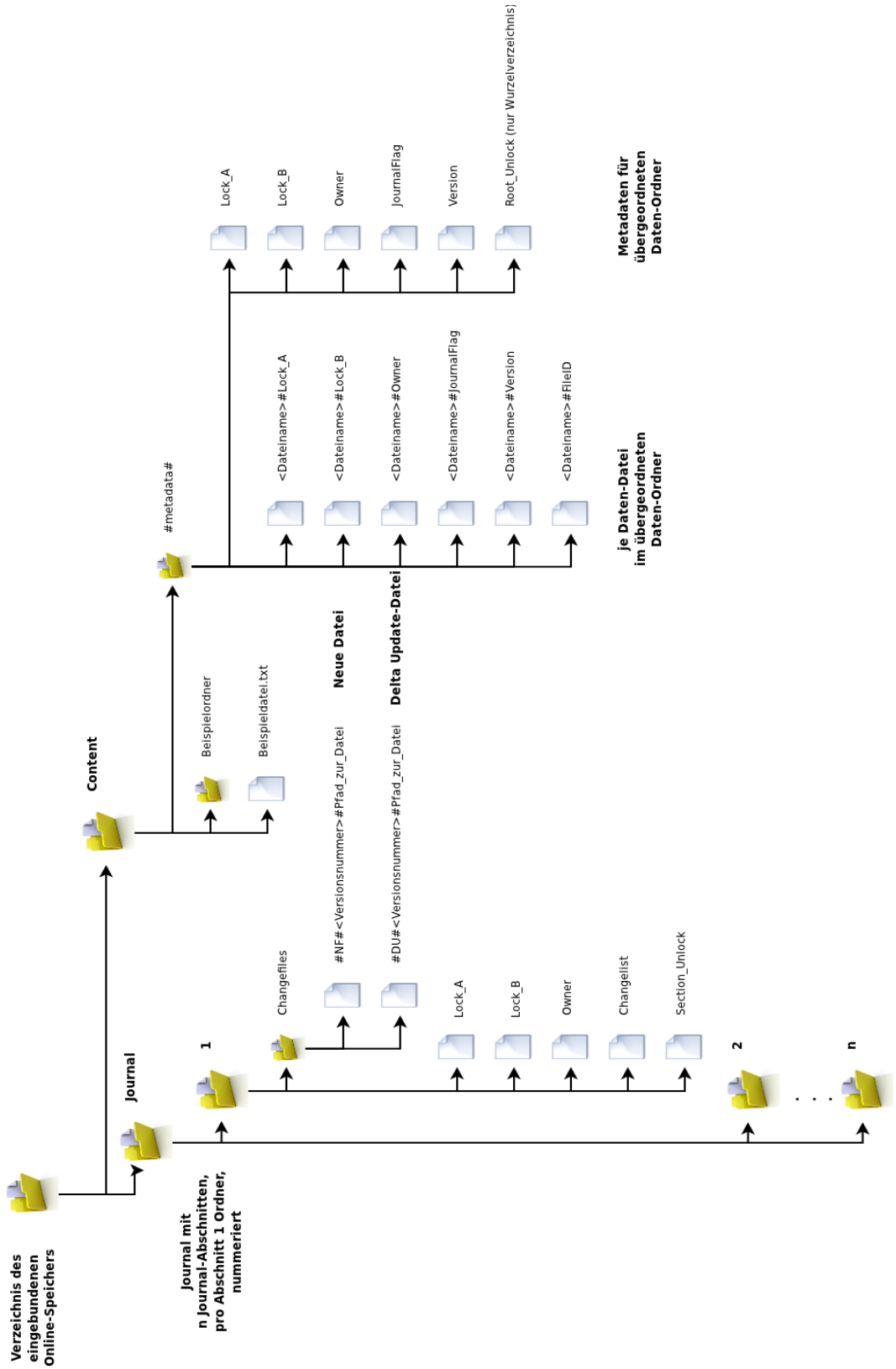


Abbildung 5.2.: Verzeichnisstruktur eines Online-Speichers mit CloudFS

Schrägstriche, die unter Linux als Trennzeichen zwischen Verzeichnissen eines Pfades genutzt werden, durch ein „#“ ersetzt, damit die Änderungsdatei einen erlaubten Dateinamen besitzt. Da kein Escaping implementiert wurde, dürfen Namen von Dateien, die von CloudFS genutzt werden sollen, kein „#“ enthalten. Andernfalls ist die Speicherung des Pfades nicht möglich. Analog zu der Speicherung einer neuen Datei werden Delta Updates gespeichert.

Metadaten-Dateien, die einem Verzeichnis zugeordnet sind, liegen im zugehörigen Metadaten-Verzeichnis. Die Dateien `Lock_A` und `Lock_B` stellen die beiden Einzelsperren dar, die für den Sperralgorithmus benötigt werden. In der Datei `Owner` wird der aktuelle Besitzer einer Objektsperre gespeichert. Die Datei `Version` enthält den Versionszähler eines Objekts und in der Datei `JournalFlag` wird angegeben, ob unausgeschriebene Journal-Einträge zum zugehörigen Objekt vorliegen. Die eindeutige ID jeder Datei im Datenbereich wird in der zugehörigen Metadaten-Datei `FileID` gespeichert. Im Metadaten-Ordner des Wurzelverzeichnisses ist zusätzlich die Datei `Root_Unlock` vorhanden, die zur Deadlock-Beseitigung verwendet wird. Um Metadaten-Dateien Inhaltsdateien zuordnen zu können, wird jeweils der Dateiname dem Namen einer Metadaten-Datei vorangestellt. Bei der Ansicht eines Ordners wird das jeweilige Metadaten-Verzeichnis nicht mit angezeigt und bleibt versteckt.

5.2. Datenstrukturen und globale Parameter

In der Komponente `cloudfs_global_params.h` werden Parameter und globale Variablen gespeichert. Die beim Start aus der Konfigurationsdatei ausgelesenen Werte werden ebenfalls hier abgelegt. Die einzelnen gespeicherten Parameter und Variablen, die global zur Verfügung gestellt werden, sind in Tabelle 5.2 zu sehen. Zudem werden drei Datentypen definiert, die für die Verwaltung von geöffneten Dateien, von Delta Updates und für das Anlegen von Journal-Einträgen notwendig sind. Die drei Datentypen sind in Tabelle 5.1 aufgelistet. Der Typ `entry_data_struct` repräsentiert einen Journal-Eintrag, wobei nicht bei jeder Operation alle enthaltenen Felder genutzt werden. Der Datentyp `openedfile_struct` repräsentiert eine geöffnete Datei und `delta_update_struct` wird genutzt, um einen Schreibvorgang in einer Delta Update-Datei zu repräsentieren.

5.3. Protokollfunktionen

In der Komponente `cloudfs_protocol.h` werden Funktionen implementiert, die direkt am Ablauf der Dateioperationen beteiligt sind. Sie werden von den Funktionen aufgerufen, die das Verhalten der Dateisystemoperationen in CloudFS definieren. So können hier Dateien und Verzeichnisse samt Metadaten im Datenbereich erstellt und gelöscht werden, Journal-Einträge angelegt, ausgeführt und gelöscht werden und das Journal nach Einträgen durchsucht werden.

Datentyp	Variable	Funktion
entry_data_struct	version	Versionsnummer des Eintrags
	operation	Name der Operation
	path	Pfad zur Objekt
	second_path	Pfad zum zweiten Objekt bei RENAME
	second_version	Versionsnummer des zweiten Objekts
	journal_section	Journal-Abschnitt, in dem der Eintrag steht
	seconds_atime	Zeit des letzten Zugriffs
	seconds_mtime	Modifikationszeit
	size	Größe der Datei
	mode	Zugriffsrechte
	uid	User-ID bei CHOWN
	gid	Group-ID bei CHOWN
openedfile_struct	filehandle	Filehandle der geöffneten Datei
	path	Pfad zur geöffneten Datei
	changed	Flag, ob die Datei verändert wurde
	delta_update	Flag, ob Delta Updates geschrieben werden
	new_size	Größe der Datei nach Schreibänderungen
	delta_updates	Liste von delta_update_structs
	content_fh	Filehandle für Leseoperationen
	delta_fh	Filehandle für Delta Updates
delta_update_struct	offset_in_DU_file	Offset innerhalb der Delta Update-Datei
	offset	Offset in Datendatei
	size	Länge der zu schreibenden Änderung

Tabelle 5.1.: Globale Datentypen

cloudfs_rootdir	Pfad, unter dem der Online-Speicher eingebunden ist
cloudfs_cachedir	Pfad zum Cache-Verzeichnis
maximum_packet_lifetime	maximale Lebenszeit eines Pakets zwischen Server und Client
maximum_clock_skew	maximale Differenz aller Client-Systemuhren
maximum_backoff_time	zusätzliche Wartezeit eines Clients bei Deadlock-Beseitigung
reserved_journal_section	Nummer des reservierten Journal-Abschnitts des Clients
writeback_to_content_section	Flag, ob Journal-Einträge ausgeschrieben werden
number_of_journal_sections	Anzahl der Journal-Abschnitte auf dem Online-Speicher
my_ClientID	eindeutige 128-Bit-ID des ausführenden Clients
my_ClientName	optionaler, eindeutiger, vom Nutzer wählbarer Name des Clients
last_filehandle_used	letztes vergebenes Filehandle für offene Dateien
last_timestamp_used	letzter Zeitstempel, der für eine Objekt-ID genutzt wurde
journal_semaphore	Semaphore zur sequentiellen Ausführung der Operationen
openedfiles	Liste von openfile_structs, die offene Dateien enthält

Tabelle 5.2.: Globale Parameter und Variablen

5.3.1. Anlegen und Löschen von Journal-Einträgen

Journal-Einträge werden in einer Änderungsliste des jeweiligen Journal-Abschnitts gespeichert. Dafür werden Einträge in der Änderungsliste in Blöcken unterteilt abgelegt, wobei in der ersten Zeile eines Blocks die eindeutige ID einer Datei oder eines Verzeichnisses steht, gefolgt von je einer Zeile pro Journal-Eintrag. Vor dem Erstellen eines Eintrags wird von der aktuellen Änderungsliste eine Sicherungskopie angelegt. Dadurch kann beim Auftreten von Fehlern während des Schreibens der neuen Änderungsliste im Bedarfsfall die alte Liste wiederhergestellt werden. Außerdem wird das Journal-Flag gesetzt, das auf einen unausgeschriebenen Eintrag hinweist. Anschließend werden die Einträge der alten Liste in die neue Liste kopiert, bis der Block der Datei oder des Verzeichnisses des neuen Eintrags erreicht ist. Dort wird nun zu Beginn eines Blocks der neue Eintrag geschrieben und die restlichen Einträge der Liste wieder kopiert. Ist noch kein Block für die betreffende Datei oder für das betreffende Verzeichnis vorhanden, so wird er am Ende der Liste angelegt. Wenn das Schreiben der neuen Änderungsliste erfolgreich war, wird die Sicherungskopie gelöscht und der Vorgang ist abgeschlossen. Andere Operationen überprüfen immer zuerst, ob eine Sicherungskopie vorhanden ist und nutzen gegebenenfalls diese. Dadurch werden unvollständig geschriebene Journal-Einträge ignoriert. Beim nächsten erfolgreichen Schreiben eines Eintrags wird die Sicherungskopie wieder gelöscht und der Zugriff auf die Änderungsliste erfolgt wieder auf die normale Version.

Zum Löschen von Journal-Einträgen muss zuerst die Version der Datei oder des Verzeichnisses im Datenbereich gelesen werden. Anschließend werden aus der Änderungsliste des reservierten Journal-Abschnitts des Clients alle Einträgen gelöscht, die eine ältere Version als die gelesene Version des Objekts im Datenbereichs aufweisen. Dabei ist das Vorgehen mit Anlegen einer Sicherungskopie, Kopieren nicht veränderter Einträge und Schreiben von Änderungen analog zum Erstellen eines Eintrags. Ältere Eintrage in anderen Journal-Abschnitten können nicht gelöscht werden, da ein Client nur Zugriff auf die Änderungsliste seines eigenen Abschnitts hat. Diese Einträge werden erst gelöscht, wenn ein Client des betreffenden Abschnitts eine Änderungsoperation auf der entsprechenden Datei oder dem entsprechenden Verzeichnis ausführt.

5.3.2. Weitere Journal-Funktionen

In der Komponente `cloudfs_protocol.h` sind weitere Funktionen enthalten, die den Zugriff auf das Journal und die darin enthaltenen Einträge ermöglichen. So gibt die Funktion `get_journal_entries` eine Liste aller Journal-Einträge zurück, die im Journal für eine bestimmte Datei oder ein bestimmtes Verzeichnis vorhanden sind. Dazu werden die Änderungslisten aller Abschnitte durchlaufen und die entsprechenden Einträge der Ergebnisliste hinzugefügt.

Attribute einer Datei werden, sofern keine unausgeschriebenen Journal-Einträge für sie vorliegen, direkt aus der auf dem Online-Speicher liegenden Datei selbst gelesen. Sind allerdings Einträge vorhanden, werden je nach Art der Operation die Werte verschiedener

Algorithmus 5.1 Algorithmus zur Rekonstruktion der aktuellen Attributwerte

```
 1: procedure GET_ATTRIBUTES(int target_version, char *path)
 2:     read attributes from content file
 3:
 4:     if (no journal entries exist) then
 5:         return
 6:     else
 7:         get list of journal entries in descending order
 8:         delete all entries from list with version > target_version
 9:         delete all entries from list with version ≤ content_version
10:     end if
11:
12:     while (entry list ≠ ∅ AND not all attributes are updated) do
13:         if (first entry of list deletes file) then
14:             return ERROR
15:         end if
16:
17:         update attributes that are changed by first entry if not yet done
18:         delete first entry from list
19:     end while
20: end procedure
```

Dateiattribute aktualisiert. So kann es zum Beispiel vorkommen, dass bei einem Schreibvorgang die Größe einer Datei verändert wird, die dann im entsprechenden Journal-Eintrag vermerkt ist. Um bei der Abfrage der Dateiattribute die aktuellen Werte zu erhalten, muss also das Journal nach Einträgen durchsucht werden, die eventuell den Wert eines Attributs aktualisieren. Diese Aufgabe übernimmt die Funktion `get_attributes`, die mit den Parametern eines Dateipfades und der Zielversion, bis zu der die Attributwerte rekonstruiert werden sollen, aufgerufen wird. Der Ablauf der Funktion ist in Algorithmus 5.1 zu sehen. Zu Beginn werden die Attribute der Datei im Datenbereich gelesen. Wenn keine unausgeschriebenen Journal-Einträge existieren, werden die gelesenen Attribute zurückgegeben. Andernfalls wird eine Liste der zur Datei gehörenden Journal-Einträge angelegt und nach Versionsnummer absteigend geordnet. Ist die als Funktionsparameter übergebene Zielversion gleich Null, werden alle Journal-Einträge betrachtet. Andernfalls werden die Einträge der Liste, die eine größere Version als die Zielversion besitzen, aus der Liste gelöscht. Nun wird jeweils der erste Eintrag der Liste gelesen und die bereits aus dem Datenbereich gelesenen Attribute, die durch den Eintrag verändert werden, aktualisiert. Da jeweils der neueste Journal-Eintrag am Anfang der Liste steht, muss jedes Attribut nur ein Mal aktualisiert werden. Einträge, die ein bereits aktualisiertes Attribut ebenfalls verändern, werden ignoriert, da diese Änderung durch einen anderen Eintrag bereits überschrieben wurde. Der Vorgang bricht ab, wenn alle Elemente der Liste durchlaufen wurden oder jedes Attribut aktualisiert wurde. Eine Ausnahme stellen Einträge dar, die eine RENAME-Operation beinhalten. Um die aktuellen Attribute dieser Version zu erhalten, muss die `get_attributes`-Funktion rekursiv für die

Quelldatei aufgerufen werden. Aus diesem Grund wird die Funktion auch mit der Angabe einer Zielversion aufgerufen, da bei diesem Aufruf nur die Attribute bis zur Dateiversion rekonstruiert werden müssen, die die Datei vor der Umbenennung widerspiegelt.

Eine weitere wichtige Funktion ist `is_file_deleted`. Sie wird aufgerufen, wenn eine Operation auf einer Datei ausgeführt werden soll, aber dafür noch unausgeschriebene Journal-Einträge bestehen. Mit Hilfe dieser Funktion kann dann überprüft werden, ob einer dieser Einträge die Datei löscht, was eventuell die Ausführung der Dateioperation unmöglich macht. Die Realisierung dieser Funktion ähnelt der von `get_attributes`: Es wird zuerst eine Liste der Journal-Einträge erstellt und entsprechend der Versionsnummer absteigend geordnet. Anschließend wird die Liste durchlaufen, bis entweder ein Eintrag gefunden wird, der die Datei löscht, oder das Ende erreicht wird. Wenn ein solcher Eintrag gefunden wurde, gibt die Funktion die Meldung zurück, dass die Datei gelöscht wurde, andernfalls, dass sie noch existiert.

5.3.3. Ausschreiben von Journal-Einträgen und Rekonstruktion der aktuellen Version

Das Ausschreiben der Journal-Einträge geschieht durch zwei Funktionen: `execute_journal_entry` schreibt genau einen Eintrag aus und wird direkt im Anschluss an das Erstellen des Journal-Eintrags aufgerufen, sofern der Client dies unterstützt. Die Funktion `writeback_journal_entries` wird genutzt, um zum einen vor Beginn einer Dateioperation bisher unausgeschriebene Journal-Einträge in den Datenbereich zu übertragen. Zum anderen wird sie verwendet, wenn ein Client das Ausschreiben von Journal-Einträgen nicht unterstützt, jedoch unausgeschriebene Einträge für eine Datei vorliegen. In diesem Fall muss die aktuelle Version rekonstruiert werden, beispielsweise um eine Leseoperation durchzuführen.

Der Ablauf der Funktion `writeback_journal_entries` ist in Algorithmus 5.2 zu sehen. Zu Beginn wird eine Liste erstellt, die alle Journal-Einträge der betreffenden Datei oder des betreffenden Verzeichnisses enthält. Anschließend werden alle Einträge der Liste gelöscht, die entweder eine ältere Version als die der Datei oder des Verzeichnisses im Datenbereich besitzen oder die eine höhere Version als die als Parameter übergebene Zielversion aufweisen. Da die `writeback_journal_entries`-Funktion sowohl für das Ausschreiben von Journal-Einträgen als auch zur lokalen Rekonstruktion der aktuellen Dateiversion genutzt wird, wird entsprechend des übergebenen Filehandles entweder die Datei im Datenbereich aktualisiert oder eine Kopie der Datei im lokalen Cache-Verzeichnis erstellt. Die Verwendung der Filehandles wird in Abschnitt 5.4.2 beschrieben.

Nun wird die Liste sequentiell durchlaufen, wobei immer der erste Eintrag bearbeitet und anschließend aus der Liste entfernt wird. Je nach Art der Operation sind zur Ausführung eines Eintrags unterschiedliche Maßnahmen notwendig. Beim Erstellen einer Datei wird die erstellte Datei aus dem Journal in den Datenbereich kopiert. Bei einer Schreiboperation dagegen werden die Delta Updates, die ebenfalls in einer Datei im Journal gespeichert

Algorithmus 5.2 Algorithmus zur Zurückschreiben von Journal-Einträgen

```
1:  procedure WRITEBACK_JOURNAL_ENTRIES(int target_version, int filehandle, char *path)
2:      get list of journal entries in ascending order
3:      delete all entries from list with version > target_version
4:      delete all entries from list with version ≤ content_version
5:
6:      if (filehandle = 0) then
7:          target file = content file
8:      else
9:          download content file into cache directory
10:         target file = cache file
11:     end if
12:
13:     while (entry list ≠ ∅) do
14:         current_entry = first element of list
15:         if (current_entry.operation = „CREATE“) then
16:             copy change file to target file
17:         else if (current_entry.operation = „WRITE“) then
18:             apply delta update to target file
19:         else if (current_entry.operation = „DELETE“) then
20:             delete target file
21:         else if (current_entry.operation = „DELETE_DIR“) then
22:             delete directory
23:         else if (current_entry.operation = „TRUNCATE“) then
24:             apply changes to target file
25:         else if (current_entry.operation = „CHMOD“) then
26:             apply changes to target file
27:         else if (current_entry.operation = „CHOWN“) then
28:             apply changes to target file
29:         else if (current_entry.operation = „UTIMENS“) then
30:             apply changes to target file
31:         else if (current_entry.operation = „DELETED_BY_RENAME“) then
32:             rename target file to temporary file
33:         else if (current_entry.operation = „RENAME“) then
34:             writeback source file if not yet done
35:             rename temporary file to target file
36:         end if
37:
38:         delete current_entry from list
39:         update content version if target file is content file
40:     end while
41:
42:     delete journal entries if target file is content file
43: end procedure
```

vorliegen, auf die Datei im Datenbereich angewandt. Die Ausführung einer Löschoperation hängt davon ab, ob nach dem entsprechenden Journal-Eintrag noch weitere Einträge vorliegen. Ist dies der Fall, wird der komplette Inhalt der Datei gelöscht, aber die Datei selbst bleibt bestehen. Sind keine weiteren Einträge vorhanden, wird die Datei gelöscht. Falls die Ausführung der Einträge im Datenbereich geschieht, werden ebenfalls die zugehörigen Metadaten gelöscht. Analog dazu ist das Vorgehen beim Löschen eines Verzeichnisses.

Die Ausführung der Operationen „CHMOD“, „CHOWN“, „TRUNCATE“ und „UTIMENS“ ist identisch. Es wird jeweils die im Journal stehende Änderung auf der Zieldatei angewandt. Die Ausführung einer Umbenennung dagegen ist zweigeteilt. Der Ablauf einer Umbenennung ist in Abschnitt 5.4.2 beschrieben. Die Quelldatei einer Umbenennungsoperation wird durch den entsprechenden Journal-Eintrag in eine temporäre Datei umbenannt. Diese befindet sich im gleichen Verzeichnis wie die Zieldatei und enthält in ihrem Namen die Namen von Quell- und Zieldatei sowie die jeweilige Version der Umbenennungsoperation, was eine Zuordnung dieser Datei zu der entsprechenden Operation ermöglicht. Bei der Ausführung eines Umbenennungseintrags der Zieldatei wird zuerst die eben beschriebene temporäre Datei erstellt, indem die `writeback_journal_entries`-Funktion für die Quelldatei aufgerufen wird, sofern die Datei nicht bereits besteht. Anschließend wird die temporäre Datei in die Zieldatei umbenannt und die Operation ist abgeschlossen.

Nach der Ausführung eines Eintrags wird jeweils die Version der Datei oder des Verzeichnisses im Datenbereich angepasst, sofern die Einträge in den Datenbereich ausgeschrieben werden. Zudem werden in diesem Fall vor dem Beenden der Funktion auch die eben ausgeschriebenen Journal-Einträge gelöscht, wobei dies nur diejenigen Einträge betrifft, die sich im Journal-Abschnitt des ausführenden Clients befinden. Die restlichen Einträge werden gelöscht, sobald ein Client des betreffenden Journal-Abschnitts auf die Datei zugreift.

Der Ablauf von `execute_journal_entry` gleicht dem von `writeback_journal_entries`. Es muss dabei lediglich keine Liste von Journal-Einträgen erstellt werden, da die Funktion direkt mit einem Eintrag als Parameter aufgerufen wird. Außerdem werden die gemachten Änderungen immer direkt in den Datenbereich geschrieben.

5.4. Schnittstelle von CloudFS zu FUSE

Die zentrale Komponente des Systems ist `main.c`. Hier befindet sich die `main`-Funktion, die bei Programmstart aufgerufen wird und die sämtliche Initialisierungsaufgaben übernimmt. Daneben existiert für jede Datei- und Verzeichnisoperation, die CloudFS dem Nutzer zur Verfügung stellt, eine Funktion, die den Ablauf der jeweiligen Operation definiert. Dabei wird dann zur Realisierung auf Funktionen anderer Komponenten zugegriffen.

FUSE ist darauf ausgelegt, dass mehrere Dateioperationen zur gleichen Zeit ausgeführt werden können. Da in CloudFS alle Operationen eines Clients auf die selbe Änderungsliste zugreifen und diese Datei lesen oder editieren müssen, würde bei gleichzeitiger Ausführung mehrerer Operationen ein Konflikt entstehen. Deshalb wird eine globale Semaphore eingeführt, die alle Operationen mit Zugriff auf die Änderungsliste vor Beginn ihrer eigentlichen

Ausführung erlangen müssen. Dies stellt sicher, dass immer nur höchstens eine Operation zur gleichen Zeit Zugriff auf die Änderungsliste hat, womit Konflikte vermieden werden. Das bedeutet allerdings, dass die Parallelität verloren geht, was wiederum Leistungseinbußen zur Folge hat. Nach Beenden einer Operation wird die Semaphore wieder freigegeben und eine andere Operation kann ausgeführt werden.

5.4.1. Initialisierung und Beenden des Programms

Beim Programmstart wird zuerst die Konfigurationsdatei gelesen, die beim Aufruf als Parameter übergeben werden muss. Dazu wird die zugehörige Funktion `parse_config_file` der Komponente `cloudfs_config_parser.h` aufgerufen. Diese liest die vorhandenen Parameter ein und speichert sie in den global zur Verfügung gestellten Variablen der Komponente `cloudfs_global_params.h`. Anschließend wird durch Lesen der Ordneranzahl des Journal-Verzeichnisses überprüft, wie viele Journal-Abschnitte zur Verfügung stehen. Sofern in der Konfigurationsdatei keine Client-ID angegeben ist, muss eine eindeutige 128-Bit-ID generiert werden, die dann in der Konfigurationsdatei gespeichert wird. Da das Erstellen von eindeutigen IDs nicht Hauptaugenmerk der Arbeit war, ist die Generierung vereinfacht. Es werden vier zufällige, vierstellige Zahlen generiert, in Zeichenketten umgewandelt und anschließend konkateniert. Da sich die erwartete Anzahl an Clients im niedrigen zweistelligen Bereich bewegt, ist die Wahrscheinlichkeit für zwei identische IDs bei einer 16-stelligen Zahl ausreichend klein, um von einer quasi-eindeutigen ID sprechen zu können. Nach der eventuell notwendigen Generierung einer Client-ID ist die Initialisierung abgeschlossen und der CloudFS-Client ist bereit, Operationen auszuführen.

Beim Beenden des Programms wird die Funktion `cloudfs_destroy` aufgerufen. Diese gibt alle gehaltenen Datei- und Verzeichnissperren auf. Zudem muss der reservierte Journal-Abschnitt freigegeben werden. Danach kann das Programm erfolgreich beendet werden.

5.4.2. Implementierung der Dateisystemoperationen

Im folgenden Abschnitt werden die Funktionen vorgestellt, die die Dateisystemoperationen von CloudFS implementieren. Der Entwurf der einzelnen Operationen wurde bereits in Kapitel 4.3.1. beschrieben.

cloudfs_getattr

Die Funktion `cloudfs_getattr` liest Attribute von Dateien und Verzeichnissen und gibt sie dem aufrufendem Programm zurück. Der Ablauf der Operation ist identisch mit dem der `get_attributes`-Funktion, die in Abschnitt 5.3.2 vorgestellt wurde. Der Grund für die doppelte Implementierung einer Funktion zum Lesen von Attributen liegt zum einen an der Verwendung der Semaphore, die vor Ausführungsbeginn einer Operation erlangt werden muss. Bei einem unausgeschriebenen Journal-Eintrag einer Umbenennungsoperation muss die Funktion `cloudfs_getattr` rekursiv für die Quelldatei aufgerufen werden. Diese könnte

allerdings nicht ausgeführt werden, da die Semaphore bereits durch die aufrufende Instanz der Funktion belegt ist. Stattdessen wird die interne Funktion `get_attributes` aufgerufen, die keine Semaphore zur Ausführung benötigt. Ein Konflikt mit der aufrufenden Instanz kann nicht auftreten, da diese während der Ausführung der internen Funktion blockiert, bis diese beendet ist. Des Weiteren ist mit der internen Funktion eine klare Trennung vollzogen zwischen Funktionen, die nach außen sichtbar sind und von anderen Programmen aufgerufen werden und internen Funktionen, die für die Ausführung einer Dateioperation benötigt werden.

cloudfs_create

Die Funktion `cloudfs_create` erstellt eine neue Datei, wobei dies auch der Fall ist, wenn die Datei zwar im Datenbereich noch existiert, aber durch einen Journal-Eintrag gelöscht wird. Zu Beginn werden unausgeschriebene Journal-Einträge in den Datenbereich übertragen, falls der Client das Ausschreiben von Journal-Einträgen unterstützt. Sollte dies der Fall sein, so muss der neueste Eintrag entweder eine Löschoperation gewesen sein oder die Datei wurde umbenannt und dadurch ebenfalls gelöscht. Andernfalls wäre nämlich die Funktion `cloudfs_open` aufgerufen worden, die eine bereits vorhandene Datei öffnet.

Falls die Datei noch nicht existiert, wird anschließend versucht, das Elternverzeichnis der zu erstellenden Datei zu sperren. Gelingt das nicht, wird die Operation abgebrochen. Andernfalls wird eine leere Datei samt Metadaten im Datenbereich erstellt. Existiert die Datei noch im Datenbereich und ist nur durch noch nicht ausgeschriebene Journal-Einträge gelöscht, wird ein neuer Eintrag namens „new_prepare“ erstellt. Dieser Eintrag markiert die Datei als nicht mehr gelöscht.

Nachdem die Datei erstellt oder als nicht mehr gelöscht markiert wurde, muss sie zuerst gesperrt, anschließend geöffnet und für Lese- und Schreibvorgänge vorbereitet werden. Dazu wird der Datei ein Filehandle zugewiesen, mit dem sie von anderen Operationen eindeutig identifiziert werden kann. Die Identifikation über den Dateinamen reicht nicht aus, da eine Datei vom Nutzer mehrmals geöffnet werden kann und trotzdem die Zuordnung möglich sein muss. Nun wird eine lokale Datei im Cache-Verzeichnis des Clients angelegt und geöffnet, in der Schreibvorgänge gespeichert werden. Diese Datei wird dabei entsprechend des zuvor zugewiesenen Filehandles benannt. Da es sich bei `cloudfs_create` um das Erstellen einer neuen Datei handelt, wird im Cache die komplette Datei gespeichert und nicht nur Delta Updates, wie es der Fall ist, wenn eine bereits existierende Datei geöffnet wird. Anschließend wird ein neues `openedfiles_struct` erstellt und der Liste der geöffneten Dateien hinzugefügt. Das Filehandle für Lese- und Schreibzugriffe des Elements verweist dabei auf die erstellte Datei im Cache. Nun ist die Datei für Lese- und Schreibvorgänge vorbereitet und die Funktion `cloudfs_create` ist abgeschlossen.

Algorithmus 5.3 Implementierung von cloudfs_open

```
 1: procedure CLOUDFS_OPEN(int mode, char *path)
 2:     if (journal flag is set AND writeback is enabled) then
 3:         call writeback_journal_entries on content file
 4:     end if
 5:
 6:     get file handle
 7:     if (read access is demanded) then
 8:         if (journal flag is set) then
 9:             download content file to cache
10:             call writeback_journal_entries on cache file
11:             open cache file for read operations
12:         else
13:             open content file for read operations
14:         end if
15:     end if
16:     if (write access is demanded) then
17:         create delta update file in cache
18:         open delta update file for write operations
19:     end if
20:
21:     add file to opened file list
22: end procedure
```

cloudfs_open

Der Ablauf der `cloudfs_open`-Funktion ähnelt dem von `cloudfs_create` und ist in Algorithmus 5.3 zu sehen. Auch hier werden zuerst alte Journal-Einträge ausgeschrieben, sofern der Client dies unterstützt. Das Anlegen einer neuen Datei fällt dagegen weg, da `cloudfs_open` nur bei bereits existierenden Dateien aufgerufen wird.

Nach dem eventuellen Ausschreiben wird der zu öffnenden Datei ein Filehandle zugewiesen, das in einem `openedfile_struct` in der Liste der geöffneten Dateien gespeichert wird. Das weitere Vorgehen ist vom angeforderten Modus abhängig: Wird ein Lesezugriff auf die Datei angefordert und existieren keine unausgeschriebenen Journal-Einträge, wird die Datei im Datenbereich geöffnet und Lesevorgänge direkt darauf ausgeführt. Wenn dagegen noch Einträge im Journal bestehen, muss die Datei zuerst aus dem Datenbereich heruntergeladen werden, um anschließend die unausgeschriebenen Journal-Einträge auf der Version im Cache anzuwenden. Lesevorgänge werden anschließend auf dieser Datei im Cache ausgeführt. Wenn Schreibzugriff auf die Datei angefordert wird, wird im Cache-Verzeichnis eine Delta Update-Datei erstellt, in der sämtliche Schreibvorgänge bis zum Übertrag in das Journal gespeichert werden.

cloudfs_write

Beim Aufruf von `cloudfs_write` wird das Filehandle übergeben, das der Datei beim Öffnen zugewiesen wurde. Anhand dessen kann in der Liste der geöffneten Dateien das passende Element wieder gefunden werden, in dem festgehalten ist, ob eine neue Datei oder ein Delta Update geschrieben werden soll. Beim Schreibvorgang in eine neue Datei wird der übergebene Schreibpuffer an den spezifizierten Offset in die Datei geschrieben. Bei Delta Updates wird in die zugehörige Datei im Cache-Verzeichnis ein Eintrag im Format #<Offset>#<Länge>#<Daten># geschrieben, wobei Offset für die Stelle in der Datei steht, Länge die Anzahl an zu schreibenden Bytes angibt und Daten die eigentlich zu schreibenden Daten darstellen. Alle diese Variablen werden beim Aufruf von `cloudfs_write` als Parameter übergeben. Zusätzlich zum Eintrag in der Delta Update-Datei wird im `openedfile_struct` der Datei eine Liste mit Elementen vom Typ `delta_update_struct` angelegt. Jedes Element stellt einen Schreibvorgang dar und beinhaltet den Offset innerhalb der Datei, die Anzahl der geschriebenen Bytes und die Position innerhalb der Delta Update-Datei. Diese Informationen sind notwendig, falls später ein Lesevorgang Daten aus dem Bereich des Delta Update lesen will.

cloudfs_read

Beim Aufruf von `cloudfs_read` wird ebenso wie bei `cloudfs_write` das Filehandle übergeben, das der Datei beim Öffnen zugewiesen wurde, das die Identifizierung der geöffneten Datei ermöglicht. Wenn der Lesevorgang auf einer neu erstellten Datei erfolgt, wird von der zugehörigen Cache-Datei gelesen, ebenso wenn eine Rekonstruktion der Datei im Cache notwendig war. Andernfalls wird direkt von der Datei im Datenbereich gelesen.

Eine Ausnahme stellt die Situation dar, wenn bereits ein Delta Update dieser Datei im Cache vorliegt und ein Teil dieser Daten wieder gelesen werden soll. Um dies zu überprüfen, wird die Liste der Delta Updates durchlaufen und nach Überschneidungen des zu lesenden Bereichs mit den bereits geschriebenen Bereichen gesucht. Wenn eine Überlappung gefunden wurde, wird der betreffende Part aus der Delta Update-Datei gelesen. Sollen noch weitere Daten gelesen werden als die, die von Delta Updates abgedeckt werden, wird der restliche Teil entweder aus der rekonstruierten Datei im Cache oder aus der Datei im Datenbereich gelesen.

cloudfs_fsync und cloudfs_release

Die Funktion `cloudfs_fsync` überträgt Änderungen, die seit dem Öffnen der Datei gemacht wurden, ins Journal auf den Online-Speicher. Die Datei wird aber nicht geschlossen, was bei `cloudfs_release` der Fall ist. Bei beiden Operationen wird, wenn eine neue Datei angelegt wurde, die zugehörige Datei im Cache ins Journal übertragen, ansonsten die Delta Update-Datei. Anschließend wird entweder ein Journal-Eintrag für die neu erstellte Datei oder für einen normalen Schreibvorgang erstellt, der danach sofort ausgeschrieben wird, sofern

der Client das Ausschreiben von Journal-Einträgen unterstützt. Falls die Datei durch den Aufruf von `cloudfs_release` geschlossen wird, jedoch keine Änderungen vorliegen, da die Datei entweder nur zum Lesen geöffnet war oder weil keine Schreibvorgänge vorgenommen wurden, wird die Datei direkt geschlossen.

Da die Datei bei `cloudfs_fsync` nach dem Übertragen der Änderungen weiter geöffnet bleibt, müssen auch weiterhin Schreibzugriffe möglich sein. Wurde bisher in eine neue Datei geschrieben, so wird nun auf Delta Updates umgestellt. Dazu muss eine entsprechende Datei im Cache angelegt werden und das Filehandle im `openedfile_struct` in der Liste der geöffneten Dateien angepasst werden. Wenn bisher schon Delta Updates geschrieben wurden, wird lediglich der Inhalt der bisherigen Delta Update-Datei gelöscht.

cloudfs_rename

Bei der Umbenennung einer Datei wird die selbe Funktion aufgerufen wie bei der Umbenennung eines Verzeichnisses. Deshalb realisiert `cloudfs_rename` sowohl die Umbenennung von Dateien als auch von Verzeichnissen. Da anhand der übergebenen Quell- und Zielpfade nicht unterschieden werden kann, ob es sich um Dateien oder Verzeichnisse handelt, werden zu Beginn der Operation per Konsolenbefehl `stat` die Attribute abgefragt, welche dann eine Unterscheidung ermöglichen. Die Funktion `cloudfs_rename` überschreibt ohne Nachfrage den eventuell schon vorhandenen Zielpfad. Eine Rückfrage an den Nutzer muss vom aufrufenden Programm realisiert werden.

Vor der Ausführung der Umbenennung werden alte unausgeschriebene Journal-Einträge in den Datenbereich übernommen, sofern der Client das Ausschreiben von Einträgen unterstützt. Wenn eine Datei umbenannt werden soll und die Zieldatei noch nicht existiert, so sind die gleichen Schritte wie beim Anlegen einer Datei notwendig: Sperren des Elternverzeichnisses und anschließendes Erstellen der Datei samt Metadaten. Nachdem beide Dateien existieren, müssen beide zur Durchführung der Umbenennung gesperrt werden. Nun werden für beide Dateien Journal-Einträge erstellt. Wichtig hierbei ist, dass das Schreiben atomar geschieht, also entweder beide Einträge geschrieben werden oder keiner der beiden. Denn falls die Operation durch einen Verlust der Verbindung zum Server abgebrochen wird und nur einer der beiden Einträge geschrieben wurde, könnten von anderen Clients weitere Umbenennungsoperationen auf den Dateien ausgeführt werden. Dies hätte zur Folge, dass entweder die Quelldatei für die Umbenennung nicht mehr zur Verfügung steht oder dass die Quelldatei zwar durch die Umbenennung in eine temporäre Datei gelöscht wird, aber nicht weiter zur Zieldatei hin überführt wird. Um beide Einträge atomar in die Änderungsliste zu schreiben, werden der Funktion `create_journal_entry` beide Einträge im selben Aufruf übergeben. Durch das Anlegen einer Sicherungskopie der Änderungsliste ist gewährleistet, dass entweder beide Einträge in die neue Liste geschrieben werden oder bei einem Fehler während des Schreibens der Einträge weiterhin mit der Sicherungskopie eine konsistente Version der Liste existiert. Diese werden wie bei allen anderen Operationen auch nach dem Anlegen der Einträge sofort ausgeschrieben, sofern der Client dies unterstützt.

Das Umbenennen von Verzeichnissen erfolgt ähnlich zu dem von Dateien. Nachdem das Quellverzeichnis erfolgreich gesperrt wurde, muss gegebenenfalls das Zielverzeichnis erstellt werden, sofern es noch nicht existiert. Hier ist das Vorgehen identisch wie bei `cloudfs_mkdir`, der Funktion zum Erstellen eines Verzeichnisses. Anschließend muss sowohl der komplette Unterbaum des Quellverzeichnisses als auch der des Zielverzeichnisses gesperrt werden. Sollte auch nur für eine Datei oder ein Verzeichnis keine Sperre erlangt werden können, werden die bereits reservierten Sperren wieder freigegeben und die Operation wird abgebrochen. Nachdem nun beide Unterbäume gesperrt sind, müssen für alle enthaltenen Dateien und Verzeichnisse Journal-Einträge angelegt werden. Dies geschieht durch den Aufruf der Funktion `mark_dir_for_rename`. Dort werden per Tiefensuche alle Dateien und Verzeichnisse abgearbeitet und für Unterverzeichnisse die Funktion rekursiv aufgerufen. Auch hier ist wieder zu beachten, dass die Einträge von Quelle und Ziel atomar geschrieben werden, um Inkonsistenzen zu vermeiden. Nachdem alle Einträge angelegt wurden, werden sie, sofern vom Client unterstützt, sofort in den Datenbereich übernommen. Hierfür kommt die Funktion `writeback_subtree` zum Einsatz, die jeweils für das Quell- und Zielverzeichnis aufgerufen wird und die Einträge des jeweiligen Unterbaums ausschreibt. Auch hier erfolgt die Abarbeitung per Tiefensuche und rekursivem Aufruf der Funktion bei Unterverzeichnissen.

cloudfs_unlink, cloudfs_chown, cloudfs_chmod, cloudfs_truncate und cloudfs_utimens

Die Abläufe der Funktionen `cloudfs_unlink`, `cloudfs_chown`, `cloudfs_chmod`, `cloudfs_truncate` und `cloudfs_utimens` sind nahezu identisch: Nachdem die Datei zu Beginn gesperrt werden muss, wird entsprechend der Operation ein Journal-Eintrag angelegt. Anschließend wird dieser Eintrag sofort ausgeschrieben, sofern der Client dies unterstützt. Danach kann die Dateisperre wieder freigegeben werden und die Operation ist beendet.

cloudfs_opendir und cloudfs_releasedir

Die einzige Aufgabe von `cloudfs_opendir` besteht darin, ähnlich zum Öffnen von Dateien, ein Filehandle des entsprechenden Verzeichnisses zu erstellen, indem das Verzeichnis geöffnet wird. Dieses wird dann von `cloudfs_readdir` zum Lesen des Verzeichnisses genutzt. Dementsprechend ist die einzige Aufgabe von `cloudfs_releasedir`, das Verzeichnis wieder zu schließen.

cloudfs_readdir

Die Funktion `cloudfs_readdir` liefert alle im spezifiziertem Verzeichnis enthaltenen Dateien und Verzeichnisse zurück. Allerdings werden diejenigen, die mit einem „#“ beginnen, ausgeblendet, was zum Beispiel auf die jeweiligen Metadaten-Verzeichnisse zutrifft. Bei der Abarbeitung der enthaltenen Dateien und Verzeichnisse wird zudem überprüft, ob sie durch

einen noch nicht ausgeschriebenen Journal-Eintrag bereits gelöscht sind. Sollte dies der Fall sein, werden die Einträge der betreffenden Datei oder des betreffenden Verzeichnisses in den Datenbereich übernommen. Der Grund für das Ausschreiben von Lösch-Einträgen beim Auflisten eines Verzeichnisses ist, dass das aufrufende Programm standardmäßig vor Beginn einer Dateisystemoperation die Attribute der entsprechenden Datei oder des entsprechenden Verzeichnisses liest. Durch den unausgeschriebenen Lösch-Eintrag wird dann zurückgemeldet, dass die Datei oder das Verzeichnis nicht mehr existiert, was einen Abbruch der Operation zur Folge hat. Somit würden Lösch-Operationen, die nicht sofort ausgeschrieben werden, nie in den Datenbereich übernommen. Aus diesem Grund erfolgt die Ausführung der Lösch-Einträge beim Auflisten des Verzeichnis-Inhalts.

cloudfs_mkdir

Das Anlegen eines neuen Verzeichnisses durch die Funktion `cloudfs_mkdir` erfolgt analog zum Erstellen einer neuen Datei: Nachdem das Elternverzeichnis gesperrt wurde, wird das Verzeichnis samt Metadaten-Verzeichnis erstellt. Wenn das Verzeichnis durch einen Journal-Eintrag, der noch nicht in den Datenbereich übernommen ist, bereits gelöscht wurde, muss es nicht neu erstellt werden. In diesem Fall wird lediglich die Version des Verzeichnisses inkrementiert. Dadurch veraltet der unausgeschriebene Lösch-Eintrag und das Verzeichnis existiert wieder.

cloudfs_rmdir

Damit das Löschen eines Verzeichnisses durch `cloudfs_rmdir` erfolgen kann, muss wie beim Umbenennen eines Verzeichnisses der komplette Unterbaum mit den enthaltenen Dateien und Unterverzeichnissen gesperrt werden. Sollten nicht alle dafür benötigten Sperren erlangt werden können, werden bis dahin gesetzte Sperren wieder freigegeben und die Operation wird abgebrochen. Anschließend muss für jede Datei und jedes Verzeichnis des Unterbaums ein Journal-Eintrag angelegt werden. Dies geschieht durch den Aufruf der Funktion `mark_dir_for_deletion`, die per Tiefensuche den Unterbaum durchläuft und die Einträge anlegt. Sofern der Client dies unterstützt, werden die Einträge nach dem Anlegen sofort ausgeschrieben, was wie beim Umbenennen eines Verzeichnisses durch die Funktion `writeback_subtree` geschieht.

5.5. Metadatenverarbeitung

Die Komponente `cloudfs_metadata_handling.h` regelt den Zugriff auf die Metadaten von Dateien und Verzeichnissen. So werden hier die Datei- und Verzeichnissperren gesetzt und wieder freigegeben und gegebenenfalls die Deadlock-Behandlung durchgeführt. Die Algorithmen hierfür wurden bereits in Kapitel 4 beschrieben. Außerdem können die Datei-

oder Verzeichnisversion und das Journal-Flag mit Funktionen dieser Komponente gelesen und aktualisiert werden.

Die Reservierung und die Freigabe eines Journal-Abschnitts ist ebenfalls hier implementiert. Der zugehörige Sperralgorithmus wurde ebenfalls schon in Kapitel 4 beschrieben. Die Auswahl, welcher Abschnitt reserviert werden soll, wird zufällig getroffen. Zu Beginn enthält die Liste der möglichen Abschnitte alle im Online-Speicher vorhandenen Abschnitte. Anschließend wird versucht, einen daraus zufällig ausgewählten Abschnitt zu reservieren. Ist der Vorgang erfolgreich, wird die Funktion beendet. Andernfalls wird der betreffende Abschnitt aus der Liste entfernt und ein anderer, zufällig ausgewählter Abschnitt wird ausgewählt. Dieser Vorgang wiederholt sich, bis die Liste leer ist. Anschließend wird sie wieder mit allen Abschnitten aufgefüllt und der Reservierungsversuch beginnt von vorne. Nach dem dritten Erreichen einer leeren Liste wird der Vorgang abgebrochen, was das Beenden des Client-Prozesses zur Folge hat. In diesem Fall konnte kein Journal-Abschnitt reserviert werden und der Client ist nicht in der Lage das System zu nutzen.

5.5.1. Journal-Flag

Das Journal-Flag gibt an, ob unausgeschriebene Einträge für eine Datei oder ein Verzeichnis im Journal existieren. Dadurch muss vor einer Operation nur dieses Flag gelesen werden, um die Aktualität der Version im Datenbereich zu prüfen und nicht das gesamte Journal durchsucht werden. Allerdings muss dann das Flag immer zuverlässig angeben, ob Einträge existieren oder nicht.

Wenn keine Einträge im Journal vorliegen, besitzt das Flag den Wert 0, andernfalls den Wert 1. Vor dem Anlegen eines neuen Eintrags wird das Flag allerdings zwischenzeitlich auf den Wert 2 gesetzt. Dieser Zustand zeigt an, dass eventuell Einträge vorliegen. Wenn man das Flag erst nach dem Anlegen eines Eintrags setzen würde, so bestünde die Möglichkeit, dass zwischen Anlegen des Eintrags und Setzen des Flags die Verbindung zum Server abbricht und das Flag nicht gesetzt werden kann. Dann wäre zwar ein neuer Journal-Eintrag vorhanden, allerdings würde das Flag noch anzeigen, dass keine unausgeschriebenen Einträge existieren. Somit würde ein Client nicht die korrekte Version der Datei oder des Verzeichnisses erkennen können. Durch das Setzen des Flags vor dem Anlegen eines Journal-Eintrags auf den Wert 2 wird dieses Problem umgangen. Wird beim nächsten Lesevorgang des Flags der Wert 2 gelesen, so wird das Journal auf Einträge mit einer höheren Versionsnummer als der Datei oder des Verzeichnisses im Datenbereich durchsucht. Existiert ein solcher Eintrag, erhält das Flag den Wert 1. Ansonsten wird ihm der Wert 0 zugewiesen.

5.6. Hilfsfunktionen

Die Komponente `cloudfs_helper.h` stellt Hilfsfunktionen zur Verfügung, die von Funktionen anderer Komponenten aufgerufen werden und nicht direkt am Ablauf der Da-

teioperationen beteiligt sind. Am häufigsten werden Funktionen zur Pfadumwandlung genutzt. Alle Pfade, die an die Funktionen der nach außen sichtbaren Dateisystemoperationen übergeben werden, beziehen sich auf das Verzeichnis, in dem CloudFS eingebunden ist. Dieses Verzeichnis stellt das Wurzelverzeichnis von CloudFS dar. So würde beispielsweise die Datei „`Beispiel.txt`“ im Verzeichnis „`/home/Nutzer/CloudFSMount/`“ mit dem Pfad „`/home/Nutzer/CloudFSMount/Beispiel.txt`“ übergeben werden. Da sich die Datei aber in Wirklichkeit im Datenbereich auf dem Online-Speicher befindet, der etwa im Verzeichnis „`/home/Nutzer/OnlineSpeicher/`“ gemountet ist, muss jeder Pfad so übersetzt werden, dass er auf die Datei im Onlinespeicher verweist, was dann schlussendlich zum Pfad „`/home/Nutzer/OnlineSpeicher/Content/Beispiel.txt`“ führt. Dazu werden die Pfadangaben aus der Konfigurationsdatei genutzt. Es existieren neben der Funktion zur Pfadumwandlung noch weitere Funktionen, die einen übergebenen Pfad manipulieren oder Pfade zu Metadaten-Verzeichnissen und Journal-Abschnitten zurückgeben.

In der Komponente `cloudfs_helper.h` sind weiterhin Hilfsfunktionen zur Stringverarbeitung implementiert. Außerdem stellt sie eine Funktion zur Verfügung, die eine beliebige Datei unter Angabe der Quell- und Zieldatei kopiert, die zum Beispiel zum Upload von Änderungen auf den Online-Speicher genutzt wird.

Die eindeutige ID jeder Datei wird ebenfalls in dieser Komponente generiert. Dazu wird die Client-ID mit der aktuellen Systemzeit des Clients konkateniert. Die Systemzeit wird durch die vergangenen Sekunden seit dem 1.1.1970 repräsentiert. Damit ist ausgeschlossen, dass zwei verschiedene Clients die gleiche ID generieren. Allerdings muss noch sichergestellt werden, dass diese ebenfalls eindeutig sind, falls in einer Sekunde mehrere IDs generiert werden müssen. Hierfür wird der zuletzt genutzte Zeitstempel in der globalen Variable `last_timestamp_used` gespeichert. Entspricht die aktuelle Systemzeit bei der Generierung einer neuen ID der des Zeitstempels der letzten ID, wird bei der neuen ID eine Sekunde addiert, wodurch die Eindeutigkeit wieder gegeben ist.

5.7. Konfigurationsdatei

In der Konfigurationsdatei kann der Nutzer verschiedene Parameter einstellen, die für den Betrieb des CloudFS-Clients notwendig sind. Im Einzelnen sind das:

rootdir Verzeichnis, in dem der Online-Speicher gemountet ist

cachedir Verzeichnis, das als Cache benutzt werden kann

writeback_to_content_section gibt an, ob Änderungen direkt in den Datenbereich übernommen werden sollen (Wert = 1) oder nicht (Wert = 0)

maximum_packet_lifetime maximale Lebenszeit eines Pakets auf dem Weg zwischen dem Client und dem Server; Angabe in Sekunden

maximum_clock_skew maximale Differenz der Systemuhren aller das Dateisystem nutzenden Clients; Angabe in Sekunden

maximum_backoff_time maximale Zeitspanne, die ein Client zusätzlich wartet, bis er eine Sperre nochmals auf einen Deadlock überprüft; Angabe in Sekunden

client_id 16-stellige ID des ausführenden Clients

client_name beliebiger vom Nutzer wählbarer Name, um den Client leichter identifizieren zu können

Die Datei ist so aufgebaut, dass auf eine Zeile mit dem Namen eines Parameters in eckigen Klammern eine Zeile mit dem zugehörigen Parameterwert folgt. Leere Zeilen oder Zeilen, die mit einem „#“ beginnen, werden ignoriert. Im Anhang ist eine Beispielkonfiguration zu finden, die Standardwerte enthält und auch für die Evaluation genutzt wurde.

Algorithmus 5.4 Implementierung des Administrationstools

```
1: procedure ADMINTOOL( )
2:     search for journal section that is reserved by given client
3:
4:     free journal section
5:
6:     traverse content section and search for files and directories locked by given client
7:     if such a file or directory is found, free it
8: end procedure
```

5.8. Administrationstool

Das Administrationstool ist ein separates Programm, das zum Einsatz kommt, wenn ein Client ungeregelt das System verlassen und gehaltene Sperren noch nicht wieder freigegeben hat. In diesem Fall können aufgrund des Sperralgorithmus und des zugrunde liegenden Systemmodells die Sperren nicht von den anderen Clients freigegeben oder überschrieben werden. Das Administrationstool gibt die Sperren des Clients, der das System ungeregelt verlassen hat, wieder frei.

Der Ablauf des Programms ist in Algorithmus 5.4 zu sehen. Beim Aufruf des Tools muss die Client-ID oder der vom Nutzer gewählte Name des betreffenden Clients übergeben werden. Anschließend wird anhand dieser Information nach dem Journal-Abschnitt gesucht, den der Client reserviert hatte. Nachdem der entsprechende Abschnitt gefunden wurde, wird dieser freigegeben und es werden Dateien und Verzeichnisse gesucht, die noch den betreffenden Client als Besitzer der Sperre ausweisen. Dabei wird der komplette Datenbereich per Tiefensuche abgearbeitet. Falls ein Objekt gefunden wird, das noch vom betreffenden Client gesperrt ist, wird es freigegeben.

6. Evaluation

CloudFS wurde in Hinblick auf verschiedene Leistungsparameter untersucht. In diesem Kapitel wird zuerst der Versuchsaufbau beschrieben, mit dem danach verschiedene Geschwindigkeitstests durchgeführt werden. Anschließend werden die Auswirkungen des Nicht-Ausschreibens von Journal-Einträgen untersucht und es wird auf die Häufigkeit von Deadlocks des Sperrverfahrens eingegangen.

6.1. Versuchsaufbau

In Abbildung 6.1 ist der Versuchsaufbau zu sehen, mit dem die verschiedenen Tests durchgeführt wurden. Zum einen kam ein Client-PC mit Ubuntu-Betriebssystem zum Einsatz (Client-PC 1), der über eine Internetverbindung mit 2 MBit/s Übertragungskapazität im Downlink und 256 KBit/s im Uplink verfügt. Zwei weitere Computer befanden sich im LAN der Universität Stuttgart. Einer dieser PCs wurde als Server verwendet, der den Clients per NFS Zugriff auf einen freigegebenen Ordner ermöglichte, wobei NFS v4 basierend auf TCP zum Einsatz kam. Der zweite Computer im Universitätsnetz wurde ebenfalls als Client genutzt (Client-PC 2). Diese beiden PCs waren untereinander per Gigabit-LAN verbunden und die Internetverbindung hatte eine Übertragungskapazität von mindestens 1 GBit/s. Um eine Kommunikation zwischen den beiden Universitätscomputern und Client-PC 1

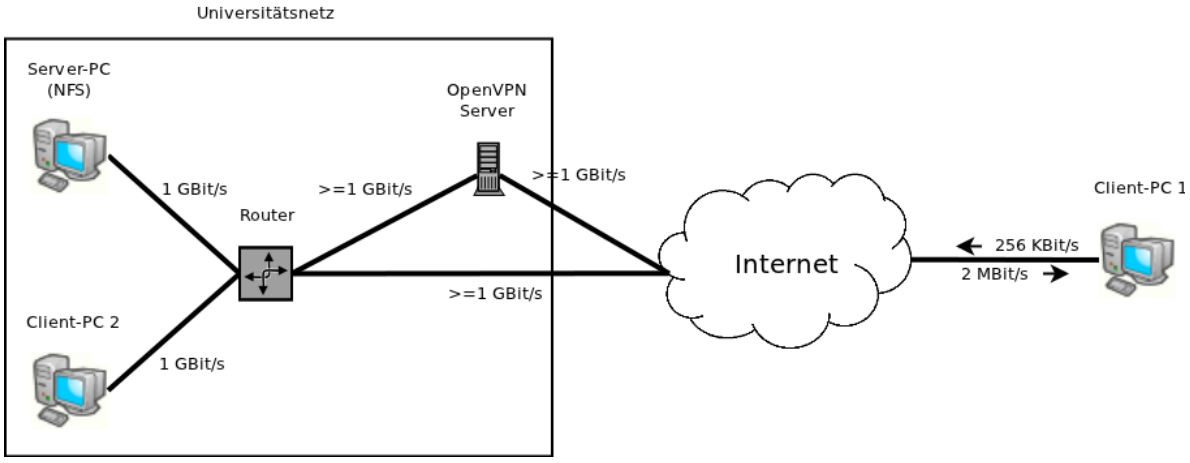


Abbildung 6.1.: Versuchsaufbau der Evaluation

zu ermöglichen, war eine VPN-Verbindung notwendig, die Client-PC 1 Zugang zum Universitätsnetz ermöglichte. Auf den Clients war schließlich jeweils die NFS-Freigabe des Servers eingebunden. Als Mount-Optionen von NFS wurden „rw“ und „sync“ verwendet, um Lese- und Schreibzugriff sowie synchrones Schreiben zu aktivieren. Auf den Client-PCs wurden dann je nach Evaluationsszenario ein oder mehrere CloudFS-Prozesse ausgeführt. Die Round Trip Time (RTT) von Client-PC 1 zum Server betrug etwa 60 ms, die von Client-PC 2 etwa 0,1 ms. Die durchgeführten Tests wurden drei Mal wiederholt. Die in diesem Kapitel angegebenen Messzeiten spiegeln das arithmetische Mittel der erzielten Zeiten der einzelnen Testläufe wider.

6.2. Untersuchung der Geschwindigkeit von Schreib- und Löschoperationen

Durch den Overhead für das für viele Dateisystemoperationen notwendige Sperren von Dateien und Verzeichnissen und das Journaling ist eine Verlangsamung der Ausführungsdauer der einzelnen Operationen in CloudFS gegenüber der direkten Speicherung der Daten auf einem Online-Speicher zu erwarten. Deshalb wurden Tests durchgeführt, die die Ausführungsdauern verschiedener Dateisystemoperationen untersuchen.

Es wurden verschiedene Testszenarios erstellt, die in Tabelle 6.1 zu sehen sind. Dabei wird jeweils die Geschwindigkeit beim Erstellen und Löschen von Dateien in CloudFS untersucht. In den ersten beiden Szenarios werden 50 leere Dateien erstellt beziehungsweise gelöscht. Die Operationen arbeiten also nur auf Metadaten und nicht auf Nutzdaten. In Szenario 3 bis 6 werden dagegen Dateien mit insgesamt 50 KByte Nutzdaten erstellt und gelöscht. Dabei verteilen sich diese Daten in den Szenarien 3 und 4 auf 50 Dateien mit jeweils 1 KByte Größe, wogegen Szenario 5 und 6 auf jeweils einer Datei mit 50 KByte Größe arbeiten. Das Erstellen und Löschen der Dateien wird von einem kleinen Programm ausgeführt, das in der Programmiersprache C implementiert ist. Dort werden die Dateien durch Aufruf der Funktionen `fopen` und `fclose` erstellt beziehungsweise geschlossen. Nutzdaten werden mit der Funktion `fputs` in die Dateien geschrieben. Das Löschen erfolgt per Aufruf der Funktion `remove`. Diese Funktionen resultieren in den Aufrufen der CloudFS-Dateisystemoperationen CREATE, WRITE, RELEASE und UNLINK.

Szenario 7 und 8 untersuchen dagegen ein Anwendungsszenario: Im Dateimanager Nautilus [nau] werden 50 Dateien à 1 KByte erstellt und anschließend in den Papierkorb verschoben. Dieser führt beim Erstellen einer Datei nach dem Erstellen zusätzlich jeweils noch die Befehle `CHMOD` und `UTIMENS` aus, die in den Aufrufen der gleichnamigen Dateisystemoperationen in CloudFS resultieren. Beim Verschieben einer Datei in den Papierkorb müssen verschiedene Ordner und Dateien angelegt und kopiert werden, wodurch eine aufwändigere Operation entsteht, als wenn die Dateien direkt gelöscht würden.

Die 8 Szenarien wurden zuerst auf Client-PC 1 und anschließend auf Client-PC 2 ausgeführt, womit der Einfluss der Verbindung zum Server ebenfalls untersucht werden konnte. Als Referenz zu den Testläufen mit CloudFS wurden die Szenarien jeweils auch noch direkt

Szenario	Beschreibung
1	50 Dateien (0 Byte pro Datei) per C-Programm erstellen
2	50 Dateien (0 Byte pro Datei) per C-Programm löschen
3	50 Dateien (1 kByte pro Datei) per C-Programm erstellen
4	50 Dateien (1 KByte pro Datei) per C-Programm löschen
5	1 Datei (50 KByte pro Datei) per C-Programm erstellen
6	1 Datei (50 KByte pro Datei) per C-Programm löschen
7	50 Dateien (1 KByte pro Datei) per Dateimanager erstellen
8	50 Dateien (1 KByte pro Datei) per Dateimanager löschen

Tabelle 6.1.: Untersuchte Szenarios der Geschwindigkeitstests

auf NFS ausgeführt, um den Overhead von CloudFS ermitteln zu können. Somit wurde jedes Szenario vier Mal durchgeführt, wobei die Verbindung und das genutzte Dateisystem variierten.

Die Ergebnisse der einzelnen Szenarien sind in Abbildung 6.2 dargestellt. Zum einen ist zu sehen, dass die Operationen auf CloudFS signifikant von der Anbindung des Clients abhängen. So kann man beobachten, dass Operationen über die LAN-Verbindung etwa 100 bis 200 Mal schneller ablaufen als die über die vergleichsweise langsame Internetverbindung. So benötigt Client-PC 1 zum Beispiel 330 Sekunden für Szenario 1, wogegen die Operationen auf Client-PC 2 bereits nach 2,5 Sekunden beendet sind. Auf CloudFS ausgeführte Szenarien weisen gegenüber NFS etwa um den Faktor 50 langsamere Ausführungszeiten auf. Das schlechtere Abschneiden von CloudFS war hier zu erwarten, da jede Dateioperation in jedem Fall auf NFS ausgeführt werden muss und bei CloudFS noch der Overhead, wie beispielsweise für das Sperren der Dateien oder das Anlegen von Journal-Einträgen, hinzukommt.

Das Erstellen einer einzelnen, 50 KByte großen Datei in Szenario 5 ist im Vergleich zu Szenario 3, bei dem 50 Dateien mit insgesamt ebenfalls 50 KByte erstellt wurden, deutlich weniger zeitintensiv. Dies ist bei beiden Verbindungstypen und Dateisystemen zu beobachten und ist durch den Overhead, der durch das Erstellen der einzelnen Dateien samt Metadaten entsteht, zu erklären. In Szenario 6, in dem die erstellte Datei wieder gelöscht wird, ist das analoge Verhalten zu beobachten. Die erzielten Zeiten für das Erstellen von 50 Dateien mit einer Größe von 1 KByte in Szenario 3 weisen eine etwa 50% längere Ausführungszeit auf als die in Szenario 1, in dem 50 leere Dateien erstellt wurden. Die zusätzlich benötigte Zeit resultiert aus dem Upload der Nutzdaten. Bei den Szenarien 2 und 4, in denen die Dateien wieder gelöscht werden, ist dagegen kein Unterschied zu beobachten, da hier die Größe der Dateien keine Rolle spielt.

In Szenario 7 werden 50 Dateien mit jeweils 1 KByte Größe per Dateimanager erstellt. Hier ist etwa eine Verdopplung der Ausführungszeiten bei beiden Verbindungstypen und Dateisystemen im Gegensatz zu Szenario 3 zu sehen. Dies kann mit der Ausführung der Befehle `CHMOD` und `UTIMENS`, die der Dateimanager im Anschluss an das Erstellen der Dateien ausführt, erklärt werden. In Szenario 8, in dem das Verschieben von 50 Dateien in den Papierkorb untersucht wird, ist im Vergleich zu Szenario 4, in dem die Dateien direkt

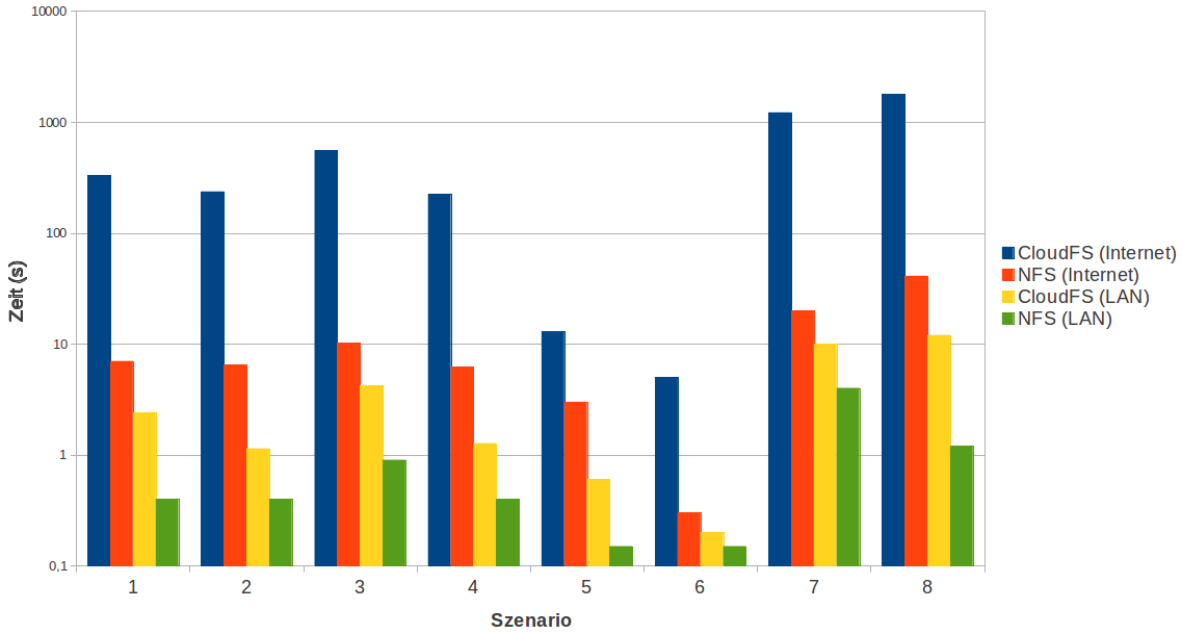


Abbildung 6.2.: Ergebnisse der Testszenarios

gelöscht werden, bei CloudFS eine Verlängerung der Ausführungszeit um den Faktor 10 zu beobachten. Die direkte Ausführung auf NFS weist etwa eine um den Faktor 3 verlängerte Ausführungszeit auf.

Die Geschwindigkeitstests zeigen, dass die Ausführungszeiten der Dateioperationen auf CloudFS deutlich länger sind als die auf NFS. Besonders deutlich wird der Unterschied, wenn eine komplexe Operation wie das Verschieben von Dateien in den Papierkorb in einem Dateimanager ausgeführt wird. Außerdem ist zu sehen, dass die Clientanbindung ebenfalls deutlichen Einfluss auf die Ausführungsdauer hat: Operationen des Clients mit LAN-Verbindung zum Server sind 100 bis 200 Mal schneller als die des Clients mit Internetverbindung.

6.3. Profiling der häufigsten Dateioperationen

Neben der Messung der Ausführungsdauern von dem Erstellen und Löschen von Dateien wurde für die häufigsten Dateioperationen OPEN, READ/WRITE, RELEASE, RENAME und DELETE ein Profiling, also das Untersuchen auf ihre Ausführungsdauern und deren Aufteilung auf ihre einzelnen Komponenten, durchgeführt. Eine Dateioperation gliedert sich in das Erwerben von benötigten Sperren, das Anlegen und Ausschreiben von Journal-Einträgen und den Upload von Änderungen, wobei nicht jede Operation alle der genannten Schritte ausführt. Die Operationen wurden auf beiden Client-PCs ausgeführt. Die gemessenen Ausführungszeiten für Client-PC 2, der über eine LAN-Verbindung zum Server verfügt, betragen allerdings alle höchstens 0,2 Sekunden. Aus diesem Grund werden in der weiteren

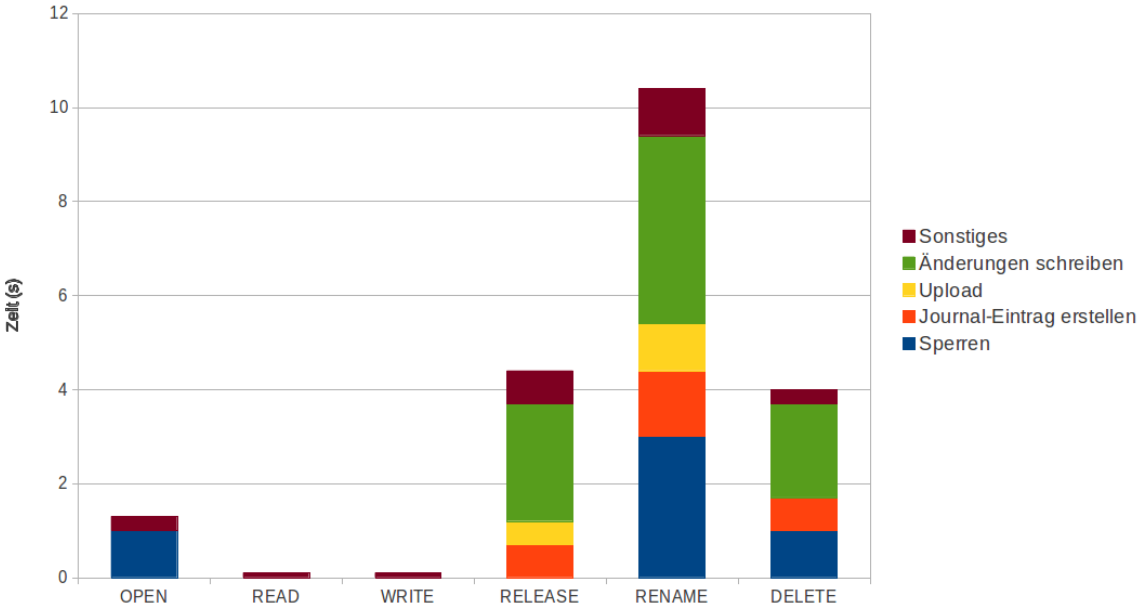


Abbildung 6.3.: Ausführungsdauern der häufigsten Dateioperationen und deren Komponenten

Betrachtung und Analyse nur die Ergebnisse von Client-PC 1, der per Internet mit dem Server verbunden ist, berücksichtigt.

Die Ergebnisse der Messungen sind in Abbildung 6.3 zu sehen. Das Öffnen einer Datei dauert etwa 1,3 Sekunden, wobei der Hauptteil auf das Sperren der Datei entfällt. Die Dauer eines WRITE-Aufrufs wurde beim Schreiben von 1 KByte gemessen, wobei die Ausführungszeit weniger als 0,1 Sekunden beträgt. Dies ist dadurch zu erklären, dass das Schreiben in CloudFS auf einer lokalen Datei geschieht, die erst beim Schließen der Datei auf den Server übertragen wird. Für die Messung eines READ-Aufrufes wurde ebenfalls 1 KByte einer Datei gelesen. Die Ausführungsdauer beträgt auch hier weniger als 0,1 Sekunden. Allerdings sind hier bei größeren zu lesenden Datenmengen längere Zeitspannen zu beobachten. Zum Vergleich wurde eine Datei von 1 MByte gelesen, wobei für die Ausführung von READ 55 Sekunden benötigt werden. Das Schließen der Datei, in die 1 KByte an Daten geschrieben wurde, dauert etwa 4,3 Sekunden. Dabei entfallen 0,7 Sekunden auf das Erstellen des zugehörigen Journal-Eintrags und 0,5 Sekunden auf den Upload der gemachten Änderungen. Auch hier wurde zum Vergleich eine Datei mit 1 MByte geschrieben, was die benötigte Zeit für den Upload auf 58 Sekunden verlängert. Der Hauptanteil der Ausführungsdauer von RELEASE nimmt allerdings das direkte Übernehmen der Änderung aus dem Journal in den Datenbereich ein, das mit etwa 2,2 Sekunden zu Buche schlägt.

Das Verschieben einer Datei nimmt mit 10,5 Sekunden die längste Zeit in Anspruch. Ein Grund hierfür ist, dass neben der Sperre der Quelldatei, die bei den meisten Operationen

erlangt werden muss, die Sperre der Zieldatei und die des zugehörigen Elternverzeichnisses gehalten werden muss, um die Operation durchführen zu können. Da auch hier pro Sperre zirka 1 Sekunde benötigt wird, summiert sich der Anteil der Zeit zur Erlangung der Sperren auf 3 Sekunden. Auch die Zeit zum Erstellen der Journal-Einträge verdoppelt sich im Gegensatz zum Schließen einer Datei, da dies hier für zwei Dateien geschehen muss. Im vorliegenden Fall wurde die Datei in einen anderen Ordner verschoben, wobei keine existierende Datei überschrieben wurde, sodass eine neue Datei samt Metadaten angelegt werden musste. Hierfür wurde etwa 1 Sekunde benötigt. Das Ausschreiben der Änderungen verdoppelt sich fast im Gegensatz zum Schließen einer Datei auf 4 Sekunden, da auch hier die Journal-Einträge zweier Dateien ausgeschrieben werden mussten.

Das Löschen einer Datei dauert zirka 4 Sekunden. Die einzelnen Bestandteile des Löschvorgangs weisen die gleiche Ausführungsdauern auf, die schon bei den anderen Operationen beobachtet werden konnten. So wird für das Sperren der Datei 1 Sekunde benötigt, das Erstellen des Journal-Eintrags dauert 0,7 Sekunden und auch hier nimmt das Ausschreiben der Änderung in den Datenbereich den Hauptteil mit etwa 2 Sekunden ein.

Die Untersuchung der Ausführungsdauern hat gezeigt, dass die RENAME-Operation mehr als doppelt so viel Zeit in Anspruch nimmt als die anderen getesteten Operationen. Außerdem ist zu beobachten, dass die einzelnen Komponenten in allen Dateioperationen stets die gleiche Ausführungsdauer besitzen. So benötigt das Erwerben einer Sperre etwa 1 Sekunde, das Anlegen eines Journal-Eintrags 0,7 Sekunden und das Ausschreiben des Eintrags etwa 2 Sekunden. Lediglich bei dem Upload von Änderungen und der Operation READ konnten keine konstanten Zeiten festgestellt werden, da diese direkt von der Größe der zu übertragenden Datei abhängen.

6.4. Zurückschreiben / Nicht-Zurückschreiben von Änderungen

In einer weiteren Testreihe wurden die Auswirkungen auf die Ausführungsdauer verschiedener Dateioperationen untersucht, falls ein Client gemachte Änderungen aus dem Journal nicht direkt in den Datenbereich ausschreibt, sondern sie dort belässt. Diese Funktion ist vor allem für mobile Clients gedacht, die dadurch weniger Daten über ihre verhältnismäßig langsame Internetverbindung übertragen müssen. Für die Evaluation führte ein Client ein Programm aus, das ununterbrochen die gleiche Operationsreihenfolge durchführte. Dabei wurden zwei verschiedene Abläufe getestet, die in Abbildung 6.4 zu sehen sind. Beide Abläufe Schreiben und Lesen die gleiche Datei zu Beginn und fragen anschließend die Dateiattribute ab. Bei Ablauf 1 wird die Datei danach zudem in eine temporäre Datei umbenannt, was in der nächsten Operation durch eine weitere Umbenennung rückgängig gemacht wird. Beide Abläufe wurden sowohl auf Client-PC 1 als auch auf Client-PC 2 ausgeführt, um somit auch den Einfluss der Verbindung zum Server untersuchen zu können.

Die gemessenen Ausführungsdauern bei sofortigem Zurückschreiben sind in Tabelle 6.2 zu sehen. Es ist zu beobachten, dass die Ausführungszeiten der einzelnen Operationen konstant sind. Dies war zu erwarten, da bei jedem Durchlauf der gleiche Zustand der Datei angetroffen wird und keine alten, unausgeschriebenen Journal-Einträge vorliegen. Zudem

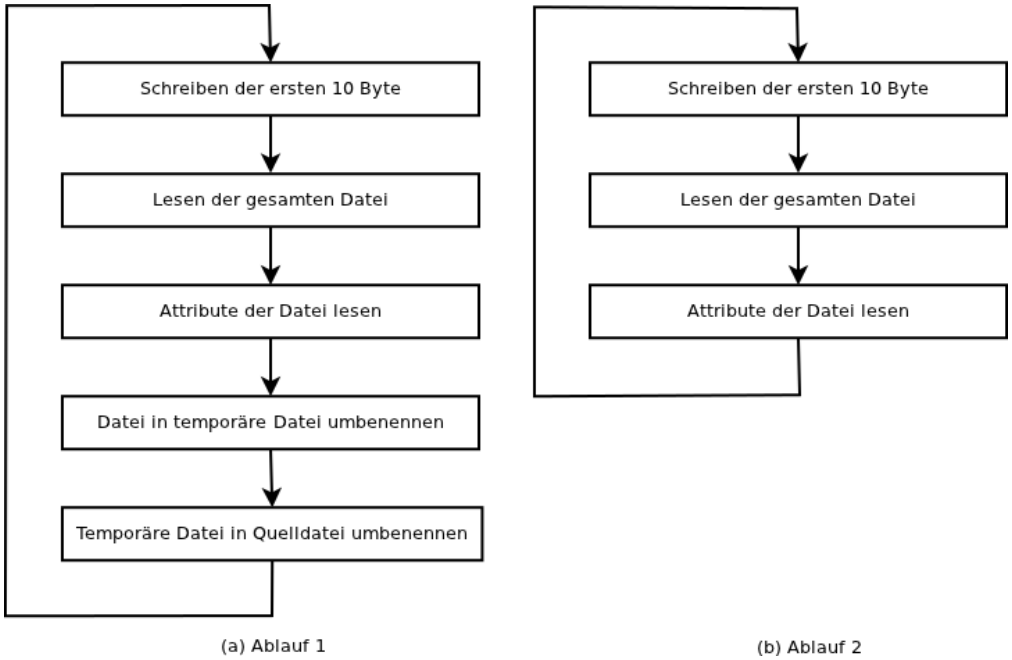


Abbildung 6.4.: Programmabläufe zum Testen der Auswirkungen des Nicht-Zurückschreibens von Journal-Einträgen

	Client-PC 1		Client-PC 2	
	Ablauf 1	Ablauf 2	Ablauf 1	Ablauf 2
Schreiben	5,6 s	5,6 s	<0,1 s	<0,1 s
Lesen	0,5 s	0,5 s	<0,1 s	<0,1 s
Attribute lesen	0,2s	0,2 s	<0,1 s	<0,1 s
Umbenennen 1	10,5 s	-	<0,1 s	-
Umbenennen 2	10,5 s	-	<0,1 s	-

Tabelle 6.2.: Ausführungsdauern bei direktem Zurückschreiben

ist wie schon bei den Geschwindigkeitstests zu sehen, dass die Ausführungsdauern im LAN nur einen Bruchteil der Ausführungsdauern über das Internet benötigen.

Nachdem die Referenzwerte bei sofortigem Ausschreiben der Änderungen gemessen wurden, wurden die Abläufe nochmals ohne Ausschreiben der Journal-Einträge wiederholt. Das Ergebnis der Messung des ersten Ablaufs ist in Abbildung 6.5 zu sehen, das des zweiten Ablaufs in Abbildung 6.6. Die Ausführungsdauern der Operationen Schreiben, Lesen und Attribute lesen sind nun nicht mehr konstant, da jeweils zuerst die aktuelle Version der Datei rekonstruiert werden muss, bevor die eigentliche Operation ausgeführt werden kann. Bei Ablauf 1 ist ein exponentieller Anstieg der Ausführungsdauer zu beobachten, wogegen bei Ablauf 2 der Anstieg in etwa linear verläuft. Die Ursache der leichten Schwankungen der Ausführungszeiten bei Ablauf 2 ist unbekannt und wurde in dieser Evaluation nicht weiter untersucht. Die Ausführungsdauern steigen ohne Umbenennungsoperationen pro-

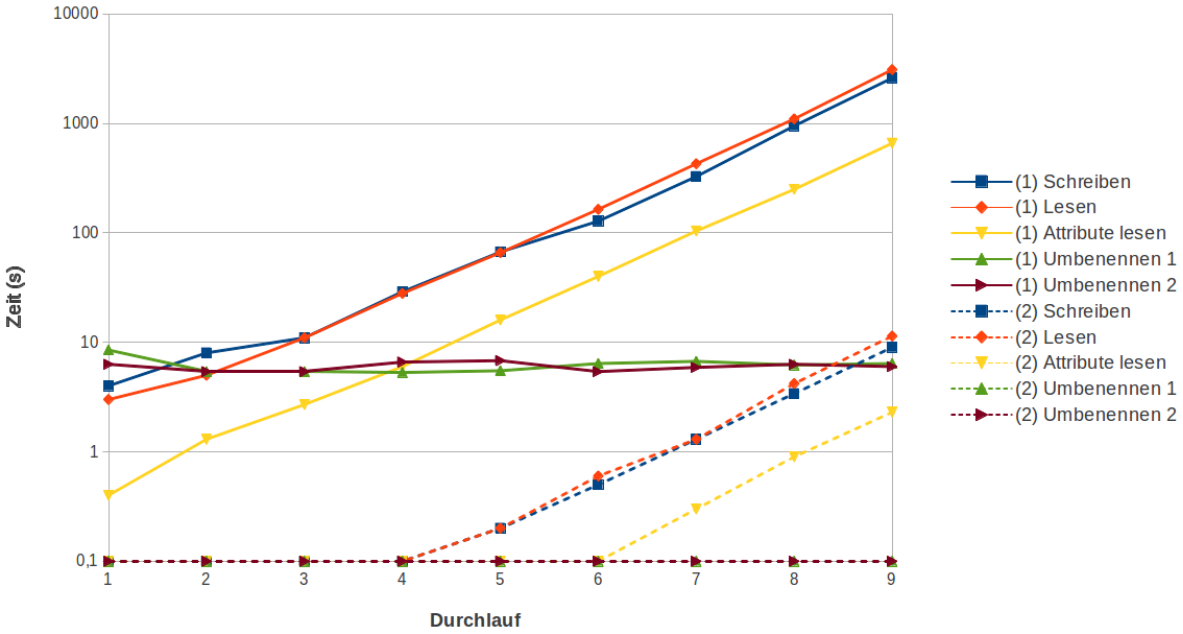


Abbildung 6.5.: Ablauf 1 bei Nicht-Zurückschreiben der Änderungen (ausführender Client in Klammern)

portional zu der Anzahl der Journal-Einträge, die für die betreffende Datei vorliegen. Im Vergleich zu den gemessenen Ausführungszeiten bei sofortigem Zurückschreiben fällt auf, dass die Umbenennungsoperationen schneller ablaufen. Der Grund dafür ist das fehlende Ausschreiben, wodurch etwa 4 Sekunden gespart werden können. Die Ausführungszeiten von den Operationen Lesen, Schreiben und Attribute lesen steigen dagegen deutlich bei Nicht-Zurückschreiben der Journal-Einträge. Lediglich bei der ersten Ausführung des Schreibens ist eine kürzere Laufzeit zu beobachten, da hier noch keine unausgeschriebenen Einträge vorliegen und auch die Zeit für das Zurückschreiben wegfällt.

Man sieht, dass die Rekonstruktion der Datei einen wesentlich höheren Aufwand erfordert, wenn unausgeschriebene Umbenennungsoperationen vorliegen. Der Grund hierfür ist, dass bei einem neuen Durchlauf immer alle früheren Dateiversionen rekonstruiert werden müssen, da die Rekonstruktion immer bei der ältesten, nicht ausgeschriebenen Version beginnt. Der gewählte Ablauf ist ein Extrembeispiel, da die selben Dateien immer wieder ineinander umbenannt werden, wodurch die einzelnen Dateiversionen mehrfach rekonstruiert werden müssen, um die aktuelle Version zu erhalten. Allerdings ist gut zu erkennen, dass mehrere unausgeschriebene Journal-Einträge einer Umbenennung oder Verschiebung einer Datei den Rekonstruktionsaufwand erheblich erhöhen.

Das Verhalten der Ausführungsdauern ist bei beiden Clients das selbe, wobei die Zeiten auf Client 1, der über die langsamere Internetverbindung verfügt, etwa um den Faktor 100 bis 200 größer sind. Bei Ablauf 2 sind die gemessenen Zeiten für Client-PC 2 stets unter 0,1 Sekunden, auch bei einem weiteren Test, der 50 Durchläufe umfasste. Generell lässt sich sagen, dass ohne Umbenennungsoperationen die Ausführungsdauern der Operationen proportional

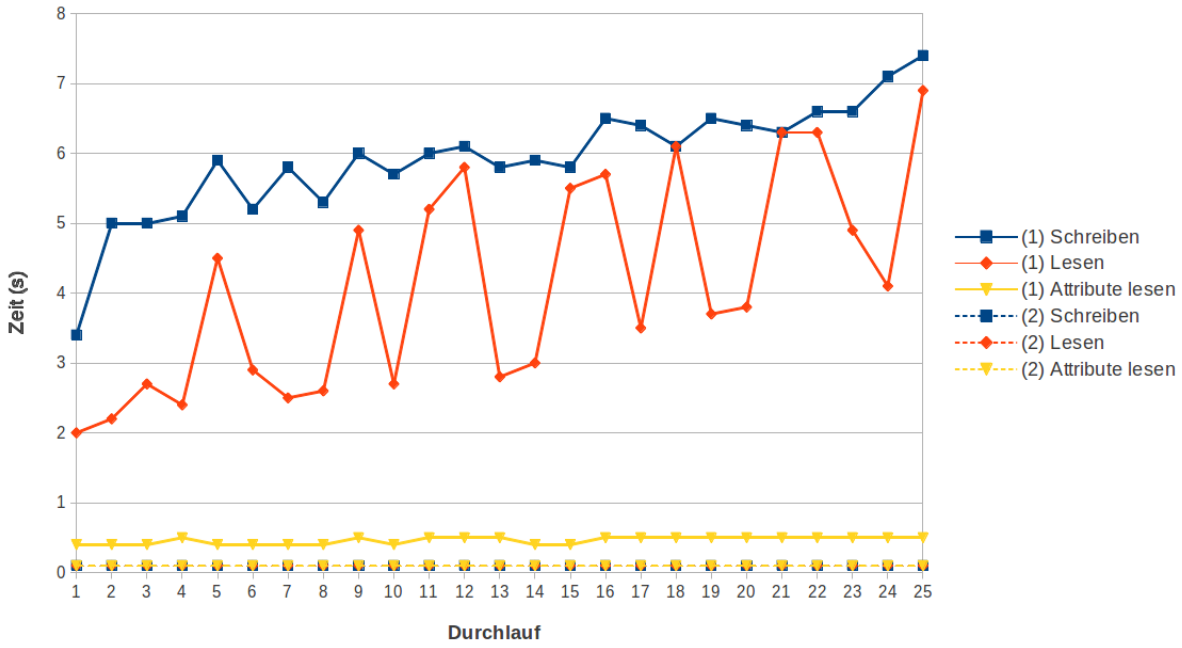


Abbildung 6.6.: Ablauf 2 bei Nicht-Zurückschreiben der Änderungen (ausführender Client in Klammern)

zu den unausgeschriebenen Journal-Einträgen steigen. Nur bei der Rekonstruktion von Umbenennungen sind deutlich längere Ausführungsdauern zu beobachten.

6.5. Häufigkeit von Deadlocks

Neben der Geschwindigkeit von Dateioperationen auf CloudFS wurde die Häufigkeit eines Deadlocks, der beim gleichzeitig ausgeführten Versuch eine Datei oder ein Verzeichnis zu sperren auftreten kann, untersucht. Dazu wurden zwei CloudFS-Prozesse auf Client-PC 1, was zwei aktiven CloudFS-Clients auf diesem PC entspricht, und ein Prozess auf Client-PC 2 ausgeführt. Somit nutzten drei Clients parallel das gleiche Dateisystem. Anschließend wurde auf allen Clients ein Programm ausgeführt, das die selbe Datei öffnet, 10 Bytes in die Datei schreibt und danach schließt. Diesen Vorgang wiederholte das Programm ununterbrochen. Es wurde dann beobachtet, wie oft es zum Deadlock der betroffenen Datei kam. Selbst nach einer Stunde Laufzeit war kein Deadlock zu beobachten. Es trat oft der Fall ein, dass die Sperre von einem Client gehalten wurde und somit die beiden anderen die Datei nicht erfolgreich öffnen konnten. Auch starteten die Clients immer wieder die Prozedur zur Deadlock-Erkennung, jedoch war das Ergebnis immer negativ und es lag kein Deadlock vor.

Die Verbindungen der Clients zum Server waren stabil und wiesen kaum Schwankungen in den Paketlaufzeiten auf. Es besteht die Möglichkeit, dass eventuell Deadlocks auftreten

könnten, wenn die Paketlaufzeiten größeren Schwankungen unterliegen und somit die Reihenfolge der eintreffenden Nachrichten der einzelnen Clients am Server untereinander verändert wird. Dieser Fall wäre zum Beispiel bei mobilen Clients denkbar, die über eine eher instabile Internetverbindung mit schwankender Verbindungsqualität verfügen. Diese Konstellation wurde jedoch im Rahmen dieser Evaluation nicht untersucht.

6.6. Fazit

Die Evaluation hat gezeigt, dass CloudFS im Gegensatz zu NFS längere Ausführungszeiten der einzelnen Dateisystemoperationen aufweist. Dies war zu erwarten, da CloudFS auf NFS aufsetzt und zusätzliche Funktionen wie das Sperrverfahren oder ein Journaling implementiert. Außerdem war zu beobachten, dass die Ausführungsdauern sehr stark von der Client-Anbindung zum Server abhängen. Die gemessenen Zeiten des Clients mit Internetzugang waren etwa 100 bis 200 Mal langsamer als die des Clients, der über eine LAN-Verbindung verfügt. Ebenfalls große Auswirkungen hat das Nicht-Zurückschreiben von Journal-Einträgen. Die Steigerung der Ausführungszeiten ist proportional zu der Anzahl an Journal-Einträgen, sofern nur Schreibeinträge vorliegen. Umbenennungen dagegen können wie im getesteten Fall sogar zu exponentiellem Wachstum der Zeiten führen. Die Untersuchung auf die Häufigkeit von Deadlocks ergab, dass selbst drei Clients ununterbrochen versuchen können eine Datei zu sperren, ohne dass ein Deadlock auftritt.

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde das Dateisystem CloudFS entwickelt. Im Gegensatz zu Cluster- und Netzwerk-Dateisystemen setzt CloudFS nicht einen zentralen Kontrollprozess oder direkte Kommunikation zwischen den Clients voraus. Die Koordination der Zugriffe auf die gespeicherten Daten erfolgt ausschließlich über indirekte Kommunikation über die verwendeten Online-Speicher. Der Nutzer kann somit parallel mit mehreren Geräten auf bei Cloud-Diensten zur Datenspeicherung abgelegte Daten zugreifen, auch wenn kein zentraler Kontrollprozess auf einem Server ausgeführt werden kann.

Zu Beginn wurden die Grundlagen von Dateisystemen beschrieben. Dabei wurden Dateisysteme für lokale Speicher, Netzwerk- und Cluster-Dateisysteme betrachtet. Das FUSE-Framework wurde ebenfalls vorgestellt, auf dessen Basis die Implementierung eines Linux-Teibers von CloudFS realisiert wurde.

Nachdem das Systemmodell beschrieben wurde, auf dem CloudFS basiert, wurde der Entwurf des Dateisystems vorgestellt. Hier wurde im Speziellen der Entwurf der Cluster-Schicht dargestellt, die für die Synchronisation der Zugriffe auf die gespeicherten Daten sowie die Behandlung von Inkonsistenzen bei Verbindungsabbrüchen zuständig ist. Ebenfalls wurde das verwendete Sperrverfahren sowie die Verwendung eines Journals beschrieben. Es wurde auf Details des Dateisystems eingegangen, wobei ausschließlich dateibasierte Speicher betrachtet wurden.

Mit Hilfe des FUSE-Frameworks wurde anschließend für das entworfene Dateisystem ein Linux-Treiber implementiert. Es wurden die verwendeten Komponenten beschrieben sowie auf Details und Probleme bei der Implementierung eingegangen. Im Speziellen wurden die Implementierungen der von CloudFS zur Verfügung gestellten Dateisystemoperationen beschrieben.

Um verschiedene Leistungsparameter von CloudFS zu untersuchen, wurde eine Evaluation durchgeführt. Dabei wurde die Ausführungsgeschwindigkeit verschiedener Dateisystemoperationen untersucht und ein Profiling der am häufigsten verwendeten Operationen erstellt. Dabei zeigte sich, dass zum einen CloudFS gegenüber den zugrunde liegenden Online-Speichern deutlich längere Ausführungszeiten aufweist. Außerdem war der Einfluss der verwendeten Verbindung des Clients zum Server signifikant. Des Weiteren wurden die Auswirkungen von unausgeschriebenen Journal-Einträgen untersucht. Es stellte sich heraus, dass abgesehen von RENAME-Operationen die Ausführungszeiten proportional zu der Anzahl der Journal-Einträge steigen. Die Häufigkeit von Deadlocks wurde ebenfalls untersucht. Im verwendeten Versuchsaufbau waren keine Deadlocks zu beobachten, wenn drei Clients ununterbrochen die selbe Datei zu sperren versuchten.

Das entwickelte Dateisystem kann durch verschiedene Maßnahmen erweitert und optimiert werden. Der vorgestellte Entwurf von CloudFS beschränkte sich auf dateibasierte Online-Speicher sowie im Speziellen auf den Entwurf der Cluster-Schicht. Um blockbasierte Speicher nutzen zu können, müsste der Entwurf für dateibasierte Speicher angepasst werden. Außerdem sind in der vorgestellten Gesamtarchitektur weitere Komponenten vorhanden, die in dieser Arbeit nicht berücksichtigt wurden. Um die Funktionalität des entwickelten Treibers von CloudFS zu erweitern, müssten diese Komponenten ebenfalls implementiert werden.

Dateisystemoperationen müssen vor ihrer eigentlichen Ausführung eine Semaphore reservieren. Damit wird die parallele Ausführung der Operationen verhindert, um Inkonsistenzen beim Zugriff auf die gemeinsam genutzte Änderungsliste zu vermeiden. Eine mögliche Lösung wäre, die Nutzung der Semaphore auf das Lesen und Schreiben von Einträgen der Änderungsliste zu begrenzen. Somit könnten andere Arbeitsschritte der Operationen dennoch parallel ausgeführt werden, was gerade im Fall des evaluierten Szenarios bei Verwendung eines Dateimanagers deutliche Leistungssteigerungen nach sich zieht. Eine weitere Möglichkeit wäre, das Schreiben der Einträge in einen Thread auszulagern, um volle Parallelität der Operationen zu erreichen.

Aus der Evaluation lassen sich Optimierungsmöglichkeiten des entwickelten Systems ableiten. Das Profiling der häufigsten Operationen hat gezeigt, dass für das Setzen einer Objektsperre etwa 1 Sekunde benötigt wird. Wenn mehrere Sperren zur Ausführung der Operation benötigt werden, werden diese sequentiell gesetzt. Eine mögliche Optimierung wäre, das Setzen von mehreren Sperren zu parallelisieren. Zum Beispiel beim Verschieben eines Ordners mit vielen enthaltenen Dateien und Verzeichnissen, bei dem für jedes Objekt im Unterbaum des Ordners eine Sperre benötigt wird, würde diese Veränderung die Ausführungszeit deutlich verkürzen.

Zwischengespeicherte Änderungsdateien eines Schreibvorgangs im Journal enthalten im Dateinamen den Pfad zur betreffenden Datei im Datenbereich. Die „/“-Zeichen im Dateipfad wurden dabei durch „#“ ersetzt, um einen gültigen Dateinamen zu erhalten. Es wurde allerdings kein Escaping implementiert, sodass Dateien, die von CloudFS gelesen werden sollen, kein „#“ enthalten dürfen. Die Implementierung von Escaping würde diesen Umstand beseitigen.

Das CloudFS zugrunde liegende System sieht ausschließlich indirekte Kommunikation der Clients über den Online-Speicher vor. Wenn in einem gegebenem Fall in jedem LAN mit NAT ein Client existiert, der durch Konfiguration direkt mit anderen Clients außerhalb des LAN kommunizieren kann, ist eine Optimierung des Sperrverfahrens möglich. Dann ist es nicht mehr nötig, die Koordination der Zugriffe über den Online-Speicher abzuwickeln. Die Clients mit direkter Kommunikationsmöglichkeit könnten ein Peer-to-Peer-Netzwerk aufbauen und ein verteiltes Verfahren zum Setzen von Objektsperren verwenden. Die anderen Clients im LAN könnten zum Sperren von Objekten ihre Anfragen über den Client mit direkter Kommunikationsmöglichkeit weiterleiten und somit alle Clients des Systems erreichen. Mit dieser Erweiterung sind schnellere Ausführungszeiten der Dateisystemoperationen zu erwarten.

Eine andere Möglichkeit, das Sperren von Dateien und Verzeichnissen zu optimieren, bieten die zugrunde liegenden Speicher. Wenn ein Speicherprotokoll zum Einsatz kommt, das selbst ein sicheres Sperrverfahren zur Verfügung stellt, kann auf das entwickelte Verfahren von CloudFS verzichtet werden. Dadurch würde der Overhead des Sperrverfahrens von CloudFS wegfallen und die Dateisystemoperationen würden beschleunigt.

A. Beispiel einer Konfigurationsdatei

```
#Root directory where online storage is mounted and all data is stored
[rootdir]
/home/testuser/OnlineStorage

#local directory that is used for cached and temporary data
[cachedir]
/home/testuser/Cache

#instantly write journal data to content section after creating journal entry
#0 means deferred writeback, 1 means instant writeback
[writeback_to_content_section]
1

#Maximum packet life time of a packet from / to the server
#WARNING: Setting this value too low can cause file inconsistencies!
#Usually maximum packet lifetime is 2 mins (TCP)
[maximum_packet_lifetime]
120

#Maximum clock skew of all client system clocks
[maximum_clock_skew]
20

#After a deadlock occured, delay each client for a different amount of time before
#trying to clear the deadlock, so that the chance of a new deadlock is minimized
[maximum_backoff_time]
10

#ID of client that is using this config file
[client_id]
6804515347366062

#User-given name for this client
[client_name]
Home Client

#endofconfig
```

Literaturverzeichnis

[AEH75] E. A. Akkoyunlu, K. Ekanadham, R. V. Huber. Some constraints and tradeoffs in the design of network communications. 9(5):67–74, 1975. (Zitiert auf Seite 30)

[Bau] W. Baumann. davfs2 Filesystem. http://savannah.nongnu.org/projects/davfs2/. (Zitiert auf Seite 19)

[ext] Ext3 Filesystem. www.kernel.org/doc/Documentation/filesystems/ext3.txt. (Zitiert auf Seite 12)

[FUS] Filesystem in Userspace FUSE. http://fuse.sourceforge.net. (Zitiert auf den Seiten 7 und 18)

[gfs] Global File System GFS. http://sources.redhat.com/cluster/gfs/. (Zitiert auf Seite 15)

[GMa] GMail Filesystem over FUSE. http://sr71.net/projects/gmailfs/. (Zitiert auf Seite 18)

[Gou] V. Gough. EncFS Encrypted Filesystem. www.arg0.net/encfs. (Zitiert auf den Seiten 16 und 18)

[Han] T. D. Hanson. ut hash, a hash table for C. http://uthash.sourceforge.net/. (Zitiert auf Seite 59)

[HR01] T. Härder, E. Rahm. *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage*. Springer, 2001. (Zitiert auf Seite 17)

[lus] Lustre Filesystem. www.lustre.org. (Zitiert auf Seite 15)

[Mica] Microsoft Corp. BitLocker Drive Encryption. http://windows.microsoft.com/en-US/windows7/products/features/bitlocker. (Zitiert auf Seite 15)

[Micb] Microsoft Corp. File Allocation Table (FAT) Filesystem. www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx. (Zitiert auf Seite 12)

[nau] Nautilus File Manager. http://live.gnome.org/Nautilus. (Zitiert auf Seite 80)

[Now89] B. Nowicki. NFS: Network File System Protocol specification. RFC 1094 (Informational), 1989. URL http://www.ietf.org/rfc/rfc1094.txt. (Zitiert auf Seite 14)

[Sil] Silicon Graphics International Corp. CXFS Filesystem. http://www.sgi.com/products/storage/software/cxfs.html. (Zitiert auf Seite 15)

[SMS+04] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), 2004. URL http://www.ietf.org/rfc/rfc3720.txt. Updated by RFCs 3980, 4850, 5048. (Zitiert auf Seite 14)

[Tec] Technical Committee T10. Small Computer Systems Interface (SCSI). www.t10.org. (Zitiert auf Seite 14)

[tru] TrueCrypt File Encryption. www.truecrypt.org. (Zitiert auf Seite 16)

[web] Web-based Distributed Authoring and Versioning WebDAV. www.webdav.org/specs/. (Zitiert auf Seite 14)

Alle URLs wurden zuletzt am 13.04.2012 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Thorsten Frosch)