Vorarlberg University of Applied Sciences

FHV
Vorarlberg University
of Applied Sciences

Masterarbeit
Fachhochschul-Studiengang
Master Informatik

symCloud - Entwicklung eines verteilten Speicherkonzeptes als Grundlage für eine Filehostingplattform

ausgeführt von

Johannes Wachter, BSc
1310249016

zur Erlangung des akademischen Grades
Master of Science in Engineering, MSc

Dornbirn, im Juli 2015

Betreuer: Prof. (FH) Dipl. Thomas Feilhauer

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 16 Juli 2015 Johannes Wachter, BSc

Zusammenfassung

Filehostingplattformen sind in der heutigen Zeit allgegenwärtig. Ohne einen Zugang zu einem der allgemein verfügbaren Dienste ist heutzutage eine Zusammenarbeit in einer Gruppe von Menschen, kaum möglich. Einige Menschen jedoch haben Bedenken, ihre Daten einem Betreiber anzuvertrauen, den sie nicht kontrollieren können. Die Angst vor dem Kontrollverlust ermöglicht quelloffenen Lösungen den Einstieg in diesen Markt. Die vorliegende Arbeit beschäftigt sich mit der Konzeption einer Speicherlösung für eine derartige Software.

Das entstehende Konzept soll Anwendung in einer neuen Filehostingplattform finden. Diese Plattform nennt sich symCloud und ist eine neue Software, die Ideen aus verschiedenen Applikationen und Technologien kombiniert, um eine optimale Lösung für den Anwender zu schaffen. In dieser Arbeit werden Technologien und Anwendungen im Bereich Filehosting vorgestellt, jedoch stellen diese meist nur Insellösungen dar. Das bedeutet, dass diese Anwendungen eine Zusammenarbeit zwischen BenutzerInnen, die nicht bei dem selben Anbieter / Dienst registriert sind, nicht bzw. nur über Umwege ermöglichen. Genau diesen Anwendungsfall versucht symCloud mit einer Idee aus Diaspora zu implementieren. Dieses verteilte soziale Netzwerk schafft ihren BenutzerInnen die Möglichkeit, mit anderen BenutzerInnen in Kontakt zu treten, die auf verschiedenen Servern registriert sind. Viele der in dieser Arbeit aufgegriffenen Ideen, wie zum Beispiel Versionierung oder Verteilung, wurden schon im Projekt Xanadu (Theodor Holm Nelson 1960) behandelt. Dieses Projekt dient als Ausgangslage für die Definition der Anforderungen an symCloud.

Das Ergebnis dieser Arbeit ist ein Konzept für eine verteilte Datenhaltung. Diese Speicherlösung ist unabhängig von der Anwendung, in der sie integriert ist. Als Beweis für die Funktionstüchtigkeit dieses Konzeptes, wurde ein einfacher Prototyp entwickelt. Er implementiert neben den wichtigsten Komponenten des Konzeptes, auch eine Plattform und einen Client, um Dateien aus einem lokalen Ordner mit der Plattform zu synchronisieren. Über definierte Schnittstellen werden diese Daten auf den vorher konfigurierten

Servern verteilt.

Abstract

File hosting platforms are ubiquitous nowadays. Co-operation in a team without one of the commonly available services is almost impossible. But some people worry about the security and privacy of their data, because they have no control over the location of their data, which standards and rules apply there, and who has acces to it. The fear of losing control allows open source solutions to enter the file hosting market. This thesis deals with the conception of a storage for such an open source solution.

The resulting concept will be applied in a new file hosting platform. The platform is called symCloud and implements the concept which combines ideas from different software projects and technologies in order to create an optimal solution for the user. Furthermore, the thesis presents technologies and applications in the field of file hosting, but these are usually only isolated applications. This means that these applications do not support the direct collaboration between users that do not use the same file hoster or service. But this use case is exaclty what symCloud tries to accomplish by using the central idea of the distributed social network Diaspora. This social network enables users to contact other users which are registered on different servers.

The result of this thesis is a concept for a distributed file storage which is independent of the application in which it is integrated. The implemented prototype contains the main parts of the described concepts. As an example of usage, the storage was integrated in a platform. Additionally, a client allows the user to synchronize files with this platform. The synchronized data will be distributed over a list of configured servers.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Seit dem Aufkommen von Cloud-Diensten, befinden sich immer mehr AnwenderInnen in einem Konflikt zwischen Datensicherheit und Datenschutz. Cloud-Dienste ermöglichen es, Daten sicher zu speichern und mit anderen zu teilen. Jedoch gibt es große Bedenken der BenutzerInnen im Bezug auf den Datenschutz, wenn sie Ihre Daten aus der Hand geben. Mit dieser Thematik beschäftigen sich auch verschiedene Studien. Sie beweisen, dass es immer mehr NutzerInnen in die Cloud zieht (siehe Abbildung 1.1 - Beispiel Dropbox[1]), allerdings gibt es sehr viele Benutzer die Bedenken gegen diese Anwendungen haben (siehe Abbildung 1.2).

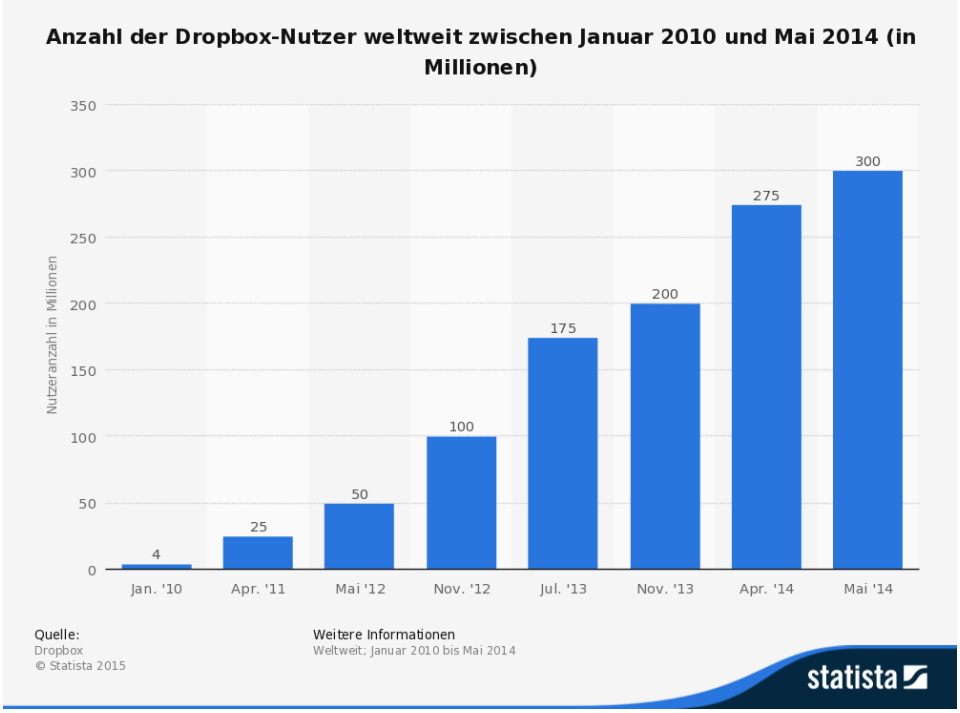

Abbildung 1.1.: Anzahl der Dropbox NutzerInnen weltweit zwischen Januar 2010 und Mai 2014 (in Millionen) [Dropbox 2014]

[1] https://www.dropbox.com/

Die Statistik aus der Abbildung 1.1 zeigt, wie die Benutzungszahlen des kommerziellen Cloud-Dienstes Dropbox in den Jahren 2010 bis 2014 von anfänglich 4 Millionen auf 300 Millionen NutzerInnen im Jahre 2014 angestiegen sind.

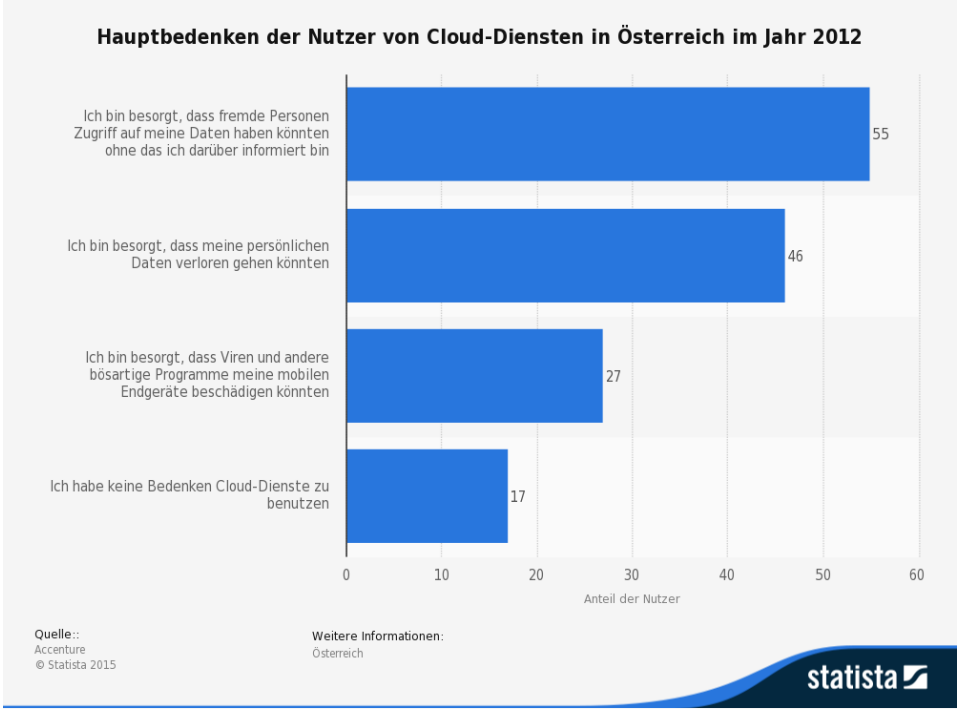

Abbildung 1.2.: Hauptbedenken der NutzerInnen von Cloud-Diensten in Österreich im Jahr 2012 [Accenture 2012]

Entgegen dieses Trends wurde im Jahre 2012 in Österreich erhoben (siehe 1.2), dass nur etwa 17% der AnwenderInnen diese Dienste ohne Bedenken verwenden. Das in dieser Studie am häufigsten genannte Bedenken ist: **Fremdzugriff auf die Daten, ohne informiert zu werden**.

Dieses Bedenken ist seit den Abhörskandalen durch verschiedenste Geheimdienste, wie zum Beispiel die NSA, noch verstärkt worden. Dies zeigt eine Umfrage aus dem Jahre 2014, die in Deutschland durchgeführt wurde (Abbildung 1.3) deutlich. Dabei gaben 71% an, dass das Vertrauen zu Cloud-Diensten durch diese Skandale beschädigt worden ist.

1.1. Alternativen zu kommerziellen Cloud-Diensten

Diese Statistiken zeigen, dass immer mehr Menschen das Bedürfnis verspüren, die Kontrolle über ihre Daten zu behalten, aber trotzdem die Vorzüge solcher Dienste

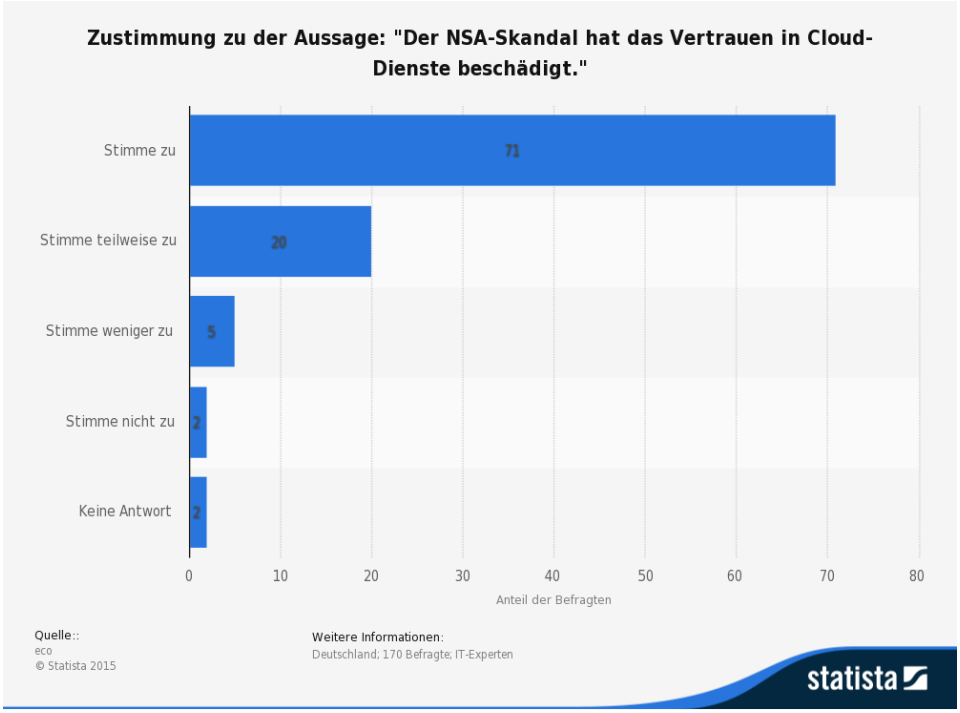

Abbildung 1.3.: Zustimmung zu der Aussage: "Der NSA-Skandal hat das Vertrauen in Cloud-Dienste beschädigt." [eco 2014]

nutzen wollen. Aufgrund dessen erregen Projekte wie Diaspora[2], ownCloud[3] und ähnliche Softwarelösungen immer mehr Aufmerksamkeit.

Diaspora Diaspora ist ein dezentrales soziales Netzwerk. Dieses soziale Netzwerk besteht aus dezentralen unabhängigen Servern bzw. Knoten (Pods genannt). Die BenutzerInnen sind durch die verteilte Infrastruktur nicht von einem Dienstleister gebunden. Das Netzwerk ermöglicht, Freunden bzw. der Familie, eine private "social-media" Plattform anzubieten und diese nach seinen Wünschen zu gestalten. Das Interessante an der Pods-Architektur ist es, dass sich die Knoten beliebig vernetzen lassen. Dies ermöglicht es BenutzerInnen, die nicht auf demselben Server registriert sind, miteinander zu kommunizieren. Pods können von jedem installiert und betrieben werden; dabei kann die BetreiberIn bestimmen, wer in sein Netzwerk eintreten darf und welche Server mit dem eigenen Server Kontakt aufnehmen dürfen. Die verbundenen Pods tauschen die Daten ohne einen zentralen Knoten aus. Dies garantiert die volle Kontrolle über seine Daten im Netzwerk [Diasporafoundation 2015]. Entwickelt wurde dieses Projekt in der Programmiersprache Ruby.

[2] https://diasporafoundation.org/
[3] https://owncloud.org/

ownCloud Das Projekt ownCloud ist eine Software, die es ermöglicht, Dateien in einer privaten Cloud zu verwalten. Mittels Endgeräte-Clients können die Dateien synchronisiert und über die Plattform auch geteilt werden. Insgesamt bietet die Software einen ähnlichen Funktionsumfang wie gängige kommerzielle Lösungen an [ownCloud 2015b]. Zusätzlich bietet es eine Kollaborationsplattform, mit der zum Beispiel Dokumente über einen Online Editor, von mehreren BenutzerInnen gleichzeitig, bearbeitet werden können. Implementiert ist dieses Projekt hauptsächlich in den Programmiersprachen PHP und JavaScript.

Beide Software-Pakete ermöglichen es den NutzerInnen, Ihre Daten in einer vertrauenswürdigen Umgebung zu verwalten. Diese Umgebung wird nur ungern verlassen, um seine Daten anderen zur Verfügung zu stellen. In dieser Arbeit wird speziell auf die Anforderungen von Anwendungen eingegangen, die es ermöglichen sollen, Dateien zu verwalten, mit anderen zu teilen und in einem definierbaren Netzwerk zu verteilen. Speziell wird der Fall betrachtet, wenn zwei BenutzerInnen, die auf verschiedenen Servern registriert sind, Dateien zusammen verwenden wollen. Dabei sollen die Vorgänge, die nötig sind, um die Dateien zwischen den Servern zu übertragen, transparent für die NutzerInnen gehandhabt werden.

1.2. Projektidee

SymCloud ist eine private Cloud-Software, die es ermöglicht, über dezentrale Knoten (ähnlich wie Diaspora) Dateien über die Grenzen des eigenen Servers hinweg zu teilen. Verbundene Knoten tauschen über sichere Kanäle Daten aus, die anschließend über einen Client mit dem Endgerät synchronisiert werden können. Dabei ist es für die BenutzerIn irrelevant, woher die Daten stammen.

Diese Arbeit beschäftigt sich weniger mit der Plattform, als mit die Konzepten, die es ermöglichen eine solche Plattform umzusetzen. Dabei wird im speziellen die Datenhaltung für solche Systeme betrachtet. Um diese Konzepte so unabhängig wie möglich von der Plattform zu gestalten, werden diese in einer eigenständigen Bibliothek entwickelt. Dieser Umstand ermöglicht eine Weiterverwendung in anderen Plattformen und Anwendungen, die ihren BenutzerInnen ermöglichen wollen, Dateien zu erstellen, zu verwalten, zu bearbeiten oder zu teilen. Damit kann das erstellte Konzept als Grundlage für eine "Spezifikation" von derartigen Prozessen ausgebaut werden.

In der ersten Phase, in der diese Arbeit entsteht, werden grundlegende Konzepte aufgestellt. Diese beginnen mit der Festlegung eines Datenmodells und der Implementierung einer Datenbank, die in der Lage ist die Daten mit anderen Servern zu teilen. Dieses Teilen von Daten soll voll konfigurierbar sein, was bedeutet, dass die AdministratorInnen die Freiheit haben zu entscheiden, welche Server welche Daten zur Verfügung gestellt bekommen. Dabei gibt es zwei Stufen der Konfiguration, zum einen über eine Liste von vertrauenswürdigen Servern, welche sozusagen eine "Whitelist" darstellt, mit denen die BenutzerInnen kommunizieren dürfen. Die zweite Stufe sind die Rechte auf ein einzelnes Objekt. Diese Rechte regeln zusätzlich, welche BenutzerInnen (und damit die Server, auf denen die BenutzerInnen registriert sind) das Objekt tatsächlich verwenden dürfen.

Kurz gesagt, symCloud ist eine Kombination der beiden Applikationen ownCloud und Diaspora. Dabei sollte es die Dateiverwaltungsfunkionen von ownCloud und die Pods-Architektur von Diaspora kombinieren, um eine optimale Alternative zu kommerziellen Lösungen, wie Dropbox anzubieten.

1.3. Inspiration

Als Inspirationsquelle für das Konzept von symCloud dienten neben den schon erwähnten Applikationen auch das Projekt Xanadu[4]. Dieses Projekt wurde im Jahre 1960 von Ted Nelson initiiert, allerdings nie finalisiert. Er arbeitet seit der Gründung an einer Implementierung an der Software [Atwood 2009]. Ted Nelson prägte den Begriff des Hypertext mit der Veröffentlichung eines wissenschaftlichen Artikels "The Hypertext. Proceedings of the World Documentation Federation" im Jahre 1965. Darin beschrieb er Hypertext als Lösung für die Probleme, die normales Papier mit sich bringt [Nelson; Smith; Mallicoat 2007].

Die darin beschriebenen Probleme sind unter Anderem:

Verbindungen Text besteht oft aus einer Menge von anderen Texten, wie zum Beispiel Zitaten oder Querverweisen. Diese Verbindungen lassen sich mithilfe von normalem Papier nur schwer abbilden bzw. visualisieren.

Form Ein Blatt Papier ist begrenzt in seiner Größe und Form. Es zwingt den Text daher in eine bestimmte Form, welche später weder verändert noch erweitert werden kann.

[4] http://hyperland.com/TBLpage

Hypertext sollte nicht das Medium, sondern die BenutzerInnen in den Vordergrund stellen. Durch verschiedenste Mechanismen sollte Xanadu die Möglichkeit schaffen, dass BenutzerInnen Dokumente verlinken und zusammensetzen können. Jedes Dokument wäre im Netzwerk eindeutig auffindbar und versioniert (also in verschiedenen Versionen abrufbar). Damit ist Xanadu ein nie zu Ende gebrachtes Konzept einer digitalen Bibliothek [Nelson; Smith; Mallicoat 2007].

1981 veröffentlichte Ted Nelson in seinem Buch "Literary Machines" 17 Thesen, die die Grundsätze des Projekts Xanadu beschreiben sollten [Nelson 1981]. Einige davon wurden durch Tim Berners-Lee in der Erfindung des "World Wide Webs" umgesetzt, andere jedoch vernachlässigt [Atwood 2009]. Einige dieser Thesen, die vernachlässigt wurden, sind konkrete Denkanstöße für ein Projekt wie symCloud.

1. Every Xanadu server can be operated independently or in a network.
2. Every user is uniquely and securely identified.
3. Every user can search, retrieve, create and store documents.
4. Every document can have secure access controls.
5. Every document can be rapidly searched, stored and retrieved without user knowledge of where it is physically stored.
6. Every document is automatically stored redundantly to maintain availability even in case of a disaster.

Diese Thesen werden in den folgenden Anforderungen an ein System wie symCloud zusammengefasst.

1.4. Anforderungen

Aufgrund der beschriebenen Projekte, die als Inspirationsquelle dienen, werden in diesem Abschnitt die Anforderungen an symCloud beschrieben. Diese Anforderungen sind unterteilt in:

Datensicherheit In diesen Abschnitt der Anforderungen fallen Gebiete wie Datenschutz und der Schutz vor Fremdzugriff.

Funktionalitäten Ein System wie symCloud sollte Funktionen mit sich bringen, die es erlauben, sich gegen andere Cloud-Lösungen behaupten zu können.

Architektur Aufgrund der Inspiration durch Diaspora und Xanadu ist diese Anforderung von der verteilten Architektur von Diaspora inspiriert.

Neben diesen funktionalen Anforderungen gelten allgemeine Anforderungen an die Software wie zum Beispiel:

Stand der Technik Die Entwicklung der Software entspricht dem Stand der Technik.

Wartbarkeit und Erweiterbarkeit Wartbarkeit bezeichnet den Grad der Effektivität mit der eine Software, durch einen Entwickler angepasst werden kann [ISO 2011]. Gleiches gilt für die Erweiterbarkeit.

1.4.1. Datensicherheit

Sinngemäß versteht man nach DIN44300, Teil 1,

- unter **Datensicherheit** die Bewahrung von Daten vor Beeinträchtigung, insbesondere durch Verlust, Zerstörung oder Verfälschung und vor Missbrauch.
- unter **Datenschutz** die Bewahrung schutzwürdiger Belange von Betroffenen oder Beeinträchtigung durch die Verarbeitung ihrer Daten, wobei es sich bei den Betroffenen um natürliche oder juristische Personen handeln kann.

Ausgedrückt in einer weniger formalen Sprache bedeutet Datenschutz den Schutz von Daten und Programmen vor unzulässiger Benutzung [Stahlknecht; Hasenkamp 2013].

Die internationalen Kriterien für die Bewertung der Sicherheit von Systemen gehen von drei grundsätzlichen Gefahren aus [Stahlknecht; Hasenkamp 2013]:

Verlust der Verfügbarkeit Benötigte Daten sind, durch einen Ausfall oder Zerstörung (zum Beispiel durch einen Benutzerfehler), nicht mehr verfügbar.

Verlust der Integrität Die Daten sind unabsichtlich oder bewusst verfälscht worden.

Verlust der Vertraulichkeit Unbefugte erhalten Zugriff auf Daten, die nicht für sie bestimmt sind.

Konkrete Bedrohungen, die eine dieser drei Gefahren auslösen können, sind Katastrophen, technische Defekte oder menschliche Handlungen (ob unbewusst oder bewusst spielt hierbei keine Rolle) [Stahlknecht; Hasenkamp 2013].

Drei der im vorherigen Abschnitt genannten Thesen des Projekts Xanadu bieten Ansätze, wie diese Anforderungen umgesetzt werden können. Durch die Redundanz (These 6) der Daten kann sowohl der Verlust der Verfügbarkeit oder Integrität in vielen Fällen verhindert werden. Wenn das System sich vergewissern will, ob die Daten valide sind,

fordert es alle Kopien der Daten an und vergleicht sie. Sind alle Versionen identisch, kann eine Verfälschung ausgeschlossen werden. Nicht mehr verfügbar sind Daten erst dann, wenn alle Kopien der Daten verloren gegangen sind. Aus diesem Grund können Replikationen helfen die Datensicherheit des Systems zu erhöhen. Die Thesen 2 und 4 bieten einen Schutz vor dem Verlust der Vertraulichkeit, indem die BenutzerIn eindeutig identifiziert werden kann und ein Zugriffsberechtigungssystem die Berechtigung überprüft. Dadurch kann ausgeschlossen werden, dass sich Dritte über die Schnittstellen des Systems, Zugriff auf Daten verschaffen, die sie nicht sehen dürften.

1.4.2. Funktionalitäten

Um ein System wie symCloud konkurrenzfähig zu vergleichbaren Systemen wie Dropbox oder ownCloud [ownCloud 2015b] zu machen, sind drei Kernfunktionalitäten unerlässlich:

Versionierung von Dateien Die Versionierung ist ein wesentlicher Bestandteil von vielen Filehosting-Plattformen. Es ermöglicht nicht nur das Wiederherstellen von alten Dateiversionen, sondern auch das Wiederherstellen von gelöschten Dateien.

Zusammenarbeit zwischen BenutzerInnen Um eine grundlegende Zusammenarbeit zwischen BenutzerInnen zu ermöglichen, ist es unerlässlich die Dateien bzw. Ordner teilen zu können.

Zugriffsberechtigungen vergeben Um die Transparenz des Systems zu steigern, sollten die BenutzerInnen entscheiden können, welche Dateien bzw. Ordner von wem bzw. wie verwendet werden können.

Diese drei Anforderungen sind auch Bestandteil des Xanadu Projektes. Durch die Versionierung kann sichergestellt werden, dass Dokumente, wenn sie einmal veröffentlicht wurden über dieselbe URL erreichbar sind. Eine neue Version der Datei besitzt eine neue URL. Dies ist speziell für Zitierungen wichtig. Die Möglichkeit Dateien zusammen zu verwenden und die Zugriffsberechtigungen sind ebenfalls zentrale Bestandteile von Xanadu.

1.4.3. Architektur

Inspiriert von der Pods-Architektur von Diaspora sollte es möglich sein, verschiedene Installationen von symCloud zu einem Netzwerk zusammenschließen zu können. Dabei

liegt der Fokus auf der Datenverteilung. Dadurch können Daten gezielt im Netzwerk verteilt werden. Damit sie dort vorhanden sind, wo sie gebraucht werden. Aufgrund der Datensicherheitsanforderungen sollten die Daten nicht wahllos im Netzwerk verteilt, sondern Konzepte ausgearbeitet werden, um Daten aufgrund der Zugriffsberechtigungen auf das Netzwerk zu verteilen.

Eine Architektur, wie die von Diaspora, erfüllt die These eins von Xanadu, indem ein Server sowohl für sich alleine arbeiten, als auch in einem Netzwerk mit anderen Servern interagieren kann.

1.4.4. Abgrenzung

Wichtige, aber in dieser Arbeit nicht betrachtete Anforderungen, sind:

Effizienz und Performance Die Effizienz und die Performance eines Systems ist meist nicht der Grund für einen Erfolg, allerdings meist eines der wichtigsten Gründe bei einem Misserfolg.

Verschlüsselung Um die Datensicherheit zu gewährleisten, sollten die Daten auf dem Speichermedium und bei der Übertragung zwischen den einzelnen Stationen, verschlüsselt werden, um den Schutz vor Fremdzugriff auch außerhalb des Systems zu gewährleisten.

Diese Anforderungen sind, wie schon erwähnt, außerhalb des Fokuses dieser Arbeit und des Konzeptes, das in dieser Arbeit beschrieben wird. Sie sind allerdings wichtige Anforderungen an ein produktiv eingesetztes System und sollten daher zumindest eine Erwähnung in dieser Arbeit finden. Sie sind vor allem als Anregung für weiterführende Entwicklungen oder Untersuchungen gedacht.

1.5. Kapitelübersicht

Im Kapitel 2 wird ein Überblick über den aktuellen Stand der Technik gegeben. Dabei werden zuerst einige Begriffe für die weitere Arbeit definiert. Danach werden Anwendungen und Technologien durchleuchtet, die die Bereiche “Cloud-Datenhaltung”, “verteilte Daten” und “verteilte Datenmodelle” umfassen.

Anschließend werden in einem Evaluierungskapitel (Kapitel 3) Technologien betrachtet, die es ermöglichen, Daten in einer verteilten Architektur zu speichern. Dazu wurden die

Bereiche “Objekt-Speicherdienste”, “verteilte Dateisysteme” und “Datenbankgestützte Dateisysteme” mit Hilfe von Beispielen analysiert und auf ihre Tauglichkeit als Basis für ein Speicherkonzept evaluiert.

Das Kapitel 4 befasst sich mit der Konzeption von symCloud. Dabei geht es zentral um das Datenmodell und die Datenbank, die diese Daten speichert und verteilt.

Dieses Konzept wurde in einem Prototypen implementiert. Die Details der Implementierung und die verwendeten Technologien werden in Kapitel 5 beschrieben.

Abschließend (Kapitel 6) werden die Ergebnisse der Arbeit zusammengefasst und analysiert. Zusätzlich wird ein Ausblick über die Zukunft des Projektes und mögliche Erweiterungen vorgestellt.

2. Stand der Technik

In diesem Kapitel werden moderne Anwendungen und ihre Architektur analysiert. Dazu werden zunächst die beiden Begriffe “verteilte Systeme” und “verteilte Dateisysteme” definiert. Anschließend werden vier Anwendungen beschrieben, die als Inspiration für das Projekt symCloud verwendet werden.

2.1. Verteilte Systeme

Andrew Tanenbaum definiert den Begriff der “verteilten Systeme” in seinem Buch folgendermaßen:

> “Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes kohärentes System erscheinen”

Diese Definition beinhaltet zwei Aspekte. Der erste Aspekt besagt, dass die einzelnen Maschinen in einem verteilten System autonom sind. Der zweite Aspekt bezieht sich auf die Software, die die Systeme miteinander verbinden. Durch die Software glaubt die BenutzerIn, dass er es mit einem einzigen System zu tun hat [Tanenbaum; Steen 2003, p. 18].

Eines der besten Beispiele für verteilte Systeme sind Cloud-Computing Dienste. Diese Dienste bieten verschiedenste Technologien. Diese umfassen Rechnerleistungen, Datenspeicher, Datenbanken und Netzwerke. Der Anwender kommuniziert hierbei immer nur mit einem System, allerdings verbirgt sich hinter der Oberfläche ein komplexes System aus vielen Hard- und Softwarekomponenten, welches sehr stark auf Virtualisierung setzt.

Gerade im Bereich der verteilten Dateisysteme, bietet sich die Möglichkeit, Dateien über mehrere Server zu verteilen an. Dies ermöglicht die Verbesserung von Datensicherheit, durch Replikation über verschiedene Server und steigert die Effizienz, durch paralleles Lesen der Daten. Diese Dateisysteme trennen meist die Nutzdaten von ihren Metadaten

und halten diese, als Daten zu den Daten, in einer effizienten Datenbank gespeichert. Um Informationen zu einer Datei zu erhalten, wird die Datenbank nach den Informationen durchsucht und direkt an den Benutzer weitergeleitet. Dies ermöglicht schnellere Antwortzeiten, da nicht auf die Nutzdaten zugegriffen werden muss und steigert dadurch die Effizienz des Systems [Seidel 2013]. Das Kapitel 3.3 befasst sich genauer mit verteilten Dateisystemen.

2.2. Cloud-Datenhaltung

Es gibt verschiedene Applikationen, die es erlauben, seine Dateien in einer Cloud-Umgebung zu verwalten. Viele dieser Applikationen sind kommerzielle Produkte, wie zum Beispiel Dropbox[1] oder Google Drive[2]. Andere jedoch sind frei verfügbar, wie zum Beispiel ownCloud[3], welches darüber hinaus sogar einen offenen Quellcode besitzt. Zwei dieser Applikationen sollen hier etwas genauer betrachtet und, soweit es möglich ist, die Speicherkonzepte analysiert werden.

2.2.1. Dropbox

Dropbox NutzerInnen können jederzeit von ihrem Desktop aus, über das Internet, mobile Geräte oder mit Dropbox verbundene Anwendungen auf ihre Dateien und Ordner zugreifen.

Alle diese Clients stellen Verbindungen mit sicheren Servern her, über die sie Zugriff auf Dateien haben und diese für andere Nutzer freigeben können. Wenn Dateien auf einem Client geändert werden, werden diese automatisch mit dem Server synchronisiert. Alle verknüpften Geräte aktualisieren sich daraufhin automatisch. Dadurch werden Dateien, die hinzugefügt, verändert oder gelöscht werden, auf allen Clients aktualisiert bzw. gelöscht.

Der Dropbox-Service betreibt verschiedenste Dienste, die sowohl für die Handhabung und Verarbeitung von Metadaten, als auch für die Verwaltung des Blockspeichers verantwortlich sind [Dropbox 2015c].

[1] https://www.dropbox.com
[2] https://www.google.com/intl/de_at/drive
[3] https://owncloud.org/

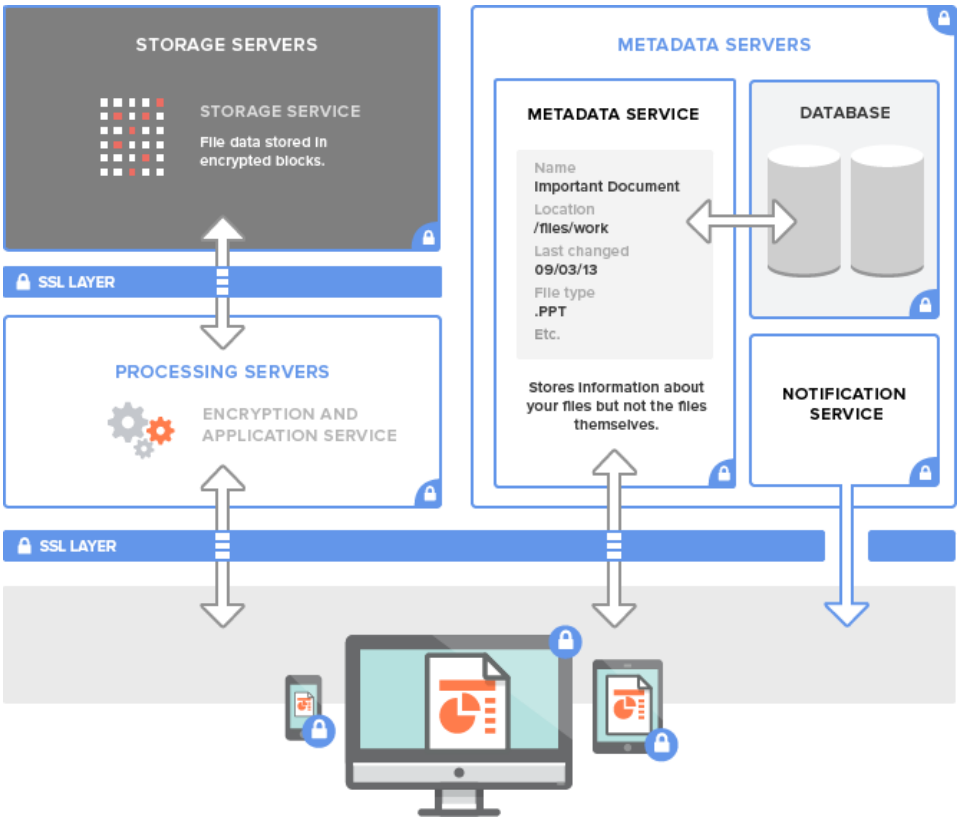

Abbildung 2.1.: Blockdiagramm der Dropbox Services [Dropbox 2015c]

In der Abbildung 2.1 werden die einzelnen Komponenten in einem Blockdiagramm dargestellt. Wie im Kapitel 2.1 beschrieben, trennt Dropbox intern die Dateien von ihren Metadaten. Der Metadata Service speichert die Metadaten und die Informationen zu ihrem Speicherort in einer Datenbank, der Inhalt der Dateien liegt jedoch in einem separaten Storage Service. Dieser Service verteilt die Daten wie ein "Load Balancer" über viele Server.

Der Storage Service ist wiederum von außen durch einen Application Service abgesichert. Die Authentifizierung erfolgt über das OAuth2 Protokoll [Dropbox 2015a]. Diese Authentifizierung wird für alle Services verwendet, auch für den Metadata Service, den Processing Servers und den Notification Service.

Der Processing oder Application Block dient als Zugriffspunkt zu den Daten. Eine Applikation, die auf Daten zugreifen möchte, muss sich an diesen Servern anmelden und bekommt dann Zugriff auf die angefragten Daten. Dies ermöglicht auch Drittherstellern Anwendungen zu entwickeln, die mit Daten aus der Dropbox arbeiten. Für diesen Zweck gibt es im Authentifizierungsprotokoll OAuth2 sogenannte "Scopes" (siehe Kapitel B). Eine weitere Aufgabe, die diese Schicht erledigt, ist die Verschlüsselung der Anwendungsdaten [Dropbox 2015c].

Die Nachteile von Dropbox im Bezug auf die im Kapitel 1.4 aufgezählten Anforderungen sind:

Closed Source Der Source-Code von Dropbox ist nicht verfügbar, daher sind eigene Erweiterungen auf die API des Herstellers angewiesen.

Datensicherheit Da Dropbox ausschließlich als "Software as a Service" angeboten wird und nicht auf eigenen Servern installiert werden kann, ist die Datensicherheit im Bezug auf den Schutz vor Fremdzugriff nicht gegeben.

Alles in allem ist Dropbox als Grundlage für symCloud aufgrund der fehlenden Erweiterbarkeit nicht geeignet. Dieser Umstand ist der Tatsache geschuldet, dass der Source-Code nicht frei zugänglich ist und es nicht gestattet wird die Software auf eigenen Servern zu installieren.

2.2.2. ownCloud

Nach den neuesten Entwicklungen arbeitet ownCloud (genauere Beschreibung in Kapitel 1) an einem ähnlichen Usecase wie symCloud. Unter dem Namen "Remote shares" wurde

in der Version sieben eine Erweiterung in den Core übernommen, mit dem es möglich sein sollte, sogenannte “Shares” mittels Link auch in einer anderen Installation einzubinden. Dies ermöglicht es, Dateien auch über die Grenzen des eigenen Servers hinweg miteinander zu teilen [Donauer 2014]. Jedoch ist diese Verteilung nicht in der Architektur verankert und nur über eine Systemerweiterung möglich.

Die kostenpflichtige Variante von ownCloud geht hier noch einen Schritt weiter. In Abbildung 2.2 ist dargestellt, wie ownCloud als eine Art Verbindungsschicht zwischen verschiedenen Lokalen- und Cloud-Speichersystemen dienen soll [ownCloud 2015a, S. 1].

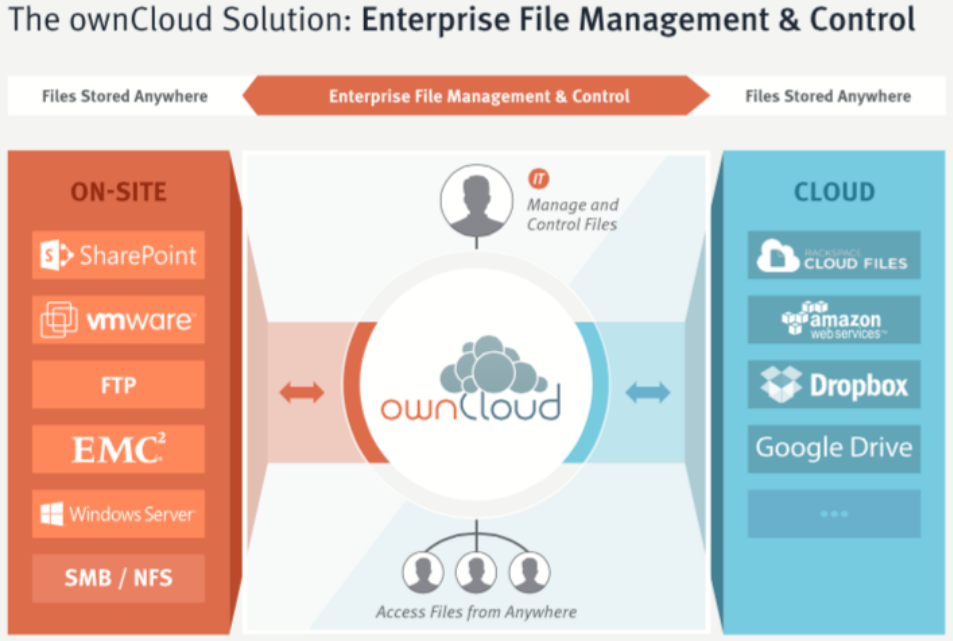

Abbildung 2.2.: ownCloud Enterprise Architektur Übersicht [ownCloud 2015a]

Um die Integration in ein Unternehmen zu erleichtern, bietet ownCloud verschiedenste Dienste an. Unter Anderem ist es möglich, Benutzerdaten über LDAP oder ActiveDirectory zu verwalten und damit ein Doppeltes führen der BenutzerInnen zu vermeiden [ownCloud 2015a, S. 2].

Für einen produktiven Einsatz wird eine skalierbare Architektur, wie in Abbildung 2.3, vorgeschlagen. An erster Stelle steht ein Load-Balancer, der die Last der Anfragen auf mindestens zwei Webserver verteilt. Diese Webserver sind mit einem MySQL-Cluster verbunden, in dem die User-Daten, Anwendungsdaten und Metadaten der Dateien gespeichert sind. Dieser Cluster besteht wiederum aus mindestens zwei redundanten Datenbankservern. Diese Architektur ermöglicht auch bei stark frequentierten Installationen eine horizontale Skalierbarkeit. Zusätzlich sind die Webserver mit dem File-Storage

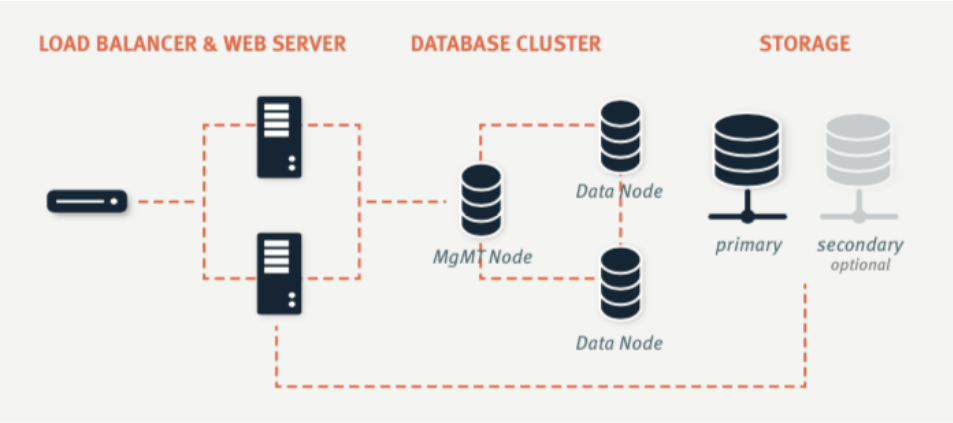

Abbildung 2.3.: Bereitstellungsszenario von ownCloud [ownCloud 2015a]

verbunden. Auch hier ist es möglich, diesen redundant bzw. skalierbar aufzubauen, um die Effizienz und Sicherheit zu gewährleisten [ownCloud 2015a, S. 3–4].

Die Nachteile von ownCloud im Bezug auf die im Kapitel 1.4 aufgezählten Anforderungen sind:

Architektur Die Software ist dafür ausgelegt, die Anforderungen, auf einem einzigen Server zu erfüllen. Es ermöglicht zwar eine verteilte Architektur, allerdings nur, um die Last auf verschiedene Server aufzuteilen. Im Gegensatz dazu versucht symCloud die Daten zwischen verschiedenen Instanzen zu verteilen, um die Zusammenarbeit zwischen BenutzerInnen zu ermöglichen, die auf verschiedenen Servern registriert sind.

Stand der Technik Aufgrund der Tatsache, dass die Entwicklung von ownCloud schon im Jahre 2010 begann und sich die Programmiersprache PHP bzw. dessen Community rasant weiterentwickelt, ist der Kern von ownCloud in einem Stil programmiert der nicht mehr dem heutigen Stand der Technik entspricht.

Obwohl ownCloud viele Anforderungen, wie zum Beispiel Versionierung oder Zugriffsberechtigungen erfüllen kann, ist das Datenmodell nicht dafür ausgelegt, die Daten zu verteilen. Ein weiterer großer Nachteil ist die bereits angesprochene veraltete Codebasis, die eine Erweiterung erschwert.

2.3. Verteilte Daten - Beispiel Diaspora

Diaspora (allgemeine Beschreibung in Kapitel 1) ist ein gutes Beispiel für Applikationen, die ihre Daten über die Grenzen eines Servers hinweg verteilen können. Diese Daten werden mit Hilfe von standardisierten Protokollen über einen sicheren Transportlayer versendet. Für diese Kommunikation zwischen den Diaspora Instanzen (Pods genannt) wird ein eigenes Protokoll namens "Federation protocol" verwendet. Es ist eine Kombination aus verschiedenen Standards, wie zum Beispiel Webfinger, HTTP und XML [Diasporafoundation 2014a]. In folgenden Situationen wird dieses Protokoll verwendet:

- Information zu BenutzerInnen finden, die auf anderen Servern gespeichert sind.
- Die erstellte Informationen an BenutzerInnen zu versenden, mit denen sie geteilt wurden.

Diaspora verwendet das Webfinger Protokoll, um zwischen den Servern zu kommunizieren. Dieses Protokoll wird verwendet, um Informationen über BenutzerInnen oder anderen Objekte abfragen zu können. Identifiziert werden diese Objekte über eine eindeutige URL. Es verwendet den HTTPS-Standard als Transportlayer für eine sichere Verbindung. Als Format für die Antworten wird JSON verwendet [Jones 2013, K. 1].

Beispiel [Diasporafoundation 2014a]:

Alice versucht mit Bob in Kontakt zu treten. Um die nötigen URLs für die weitere Kommunikation zwischen den Servern zu ermitteln führt der Pod von Alice einen Webfinger lookup auf den Pod von Bob aus. Der Response enthält einen ähnlichen Inhalt, wie in Listing 2.1 dargestellt. Dieser Response wird LRDD ("Link-based Resource Descriptor Document"[4]) genannt und enthält die URL um die Daten von dem Server abzufragen.

Listing 2.1: Host-Meta Inhalt von Bob

```
1 <Link rel="lrdd"
2       template="https://bob.diaspora.example.com/?q={uri}"
3       type="application/xrd+xml" />
```

Um Informationen über den Benutzer Bob zu erhalten, führt der Pod von Alice einen Request auf die angeführte URL mit dem Benutzernamen von Bob aus. Dieser Response enthält alle relevanten Informationen über Bob und weitere Links für nähere Details oder

[4] https://tools.ietf.org/html/rfc6415#section-6.3

Aktionen, die auf den gesuchten BenutzerInnen ausgeführt werden können. Der Response kann bei nicht eindeutigem Suchbegriff auch mehrere Entitäten enthalten.

Über dieses Protokoll lassen sich die Pods von Alice und Bob verbinden. Die Daten die dabei verteilt werden, werden auf jedem Pod in einer relationalen Datenbank abgelegt [Diasporafoundation 2014b].

Diaspora ist ein gutes Beispiel, wie Daten in einem dezentralen Netzwerk verteilt werden können. Da allerdings das gesamte Konzept dafür ausgelegt ist, BenutzerInnen miteinander kommunizieren zu lassen, ist die Erweiterung auf ein Dateimodell sehr schwierig. Jedoch könnte eine Kommunikation zwischen Diaspora und symCloud durch die Abstraktion der API durch das Webfinger Protokoll ermöglicht werden (siehe Kapitel 6.4).

2.4. Verteilte Datenmodelle - Beispiel GIT

GIT[5] ist eine verteilte Versionsverwaltung, welche ursprünglich entwickelt wurde, um den Source-Code des Linux Kernels zu verwalten.

Die Software ist im Grunde eine Key-Value Datenbank. Die Objekte werden in Form einer Datei abgespeichert, wobei der Name bzw. der Pfad aus dem Key des Objektes besteht. In der Datei wird der jeweilige Inhalt des Objektes abgelegt. Dieser Key wird ermittelt, indem ein sogenannter SHA berechnet wird. Der SHA ist ein mittels "Secure-Hash-Algorithm" berechneter Hashwert der Daten [Chacon 2009, K. 9.2]. Das Listing 2.2 zeigt, wie ein SHA in einem Unix-Terminal berechnet werden kann [Keepers 2012].

Listing 2.2: Berechnung des SHA eines Objektes

```
1 $ OBJECT='blob 46\0{"name": "Johannes Wachter", "job": "Web-Developer
      "}'
2 $ echo -e $OBJECT | shasum
3 6c01d1dec5cf5221e86600baf77f011ed469b8fe -
```

Im Listing 2.3 wird ein GIT-Objekt vom Typ BLOB erstellt und in den "Objects" Ordner geschrieben.

Listing 2.3: Erzeugung eines GIT-BLOB

```
1 $ git init
```

[5] http://git-scm.com/

```
2 $ OBJECT='blob 46\0{"name": "Johannes Wachter", "job": "Web-Developer
    "}'
3 $ echo -e $OBJECT | git hash-object -w --stdin
4 6c01d1dec5cf5221e86600baf77f011ed469b8fe
5 $ find .git/objects -type f
6     .git/objects/6c/01d1dec5cf5221e86600baf77f011ed469b8fe
```

Die Objekte in GIT sind "immutable", das bedeutet, dass sie nicht veränderbar sind. Ein einmal erstelltes Objekt wird nicht mehr aus der Datenbank gelöscht oder geändert. Bei der Änderung eines Objektes wird ein neues Objekt mit einem neuen Key erstellt [Keepers 2012].

Objekt Typen

GIT kennt folgende Typen:

BLOB Ein BLOB repräsentiert eine einzelne Datei in GIT. Der Inhalt der Datei wird in einem Objekt gespeichert. Bei Änderungen ist GIT auch in der Lage, inkrementelle DELTA-Dateien zu speichern. Beim Wiederherstellen werden diese DELTAs der Reihe nach aufgelöst. Ein BLOB besitzt für sich gesehen keinen Namen [Chacon 2009, K. 9.2].

TREE Der TREE beschreibt einen Ordner im Repository. Ein TREE enthält Referenzen auf andere TREE bzw. BLOB Objekte und definiert damit eine Ordnerstruktur. Wie auch der BLOB besitzt ein TREE für sich gesehen keinen Namen. Der Name des Objektes wird über die Referenz zu einem TREE oder einem BLOB festgelegt (siehe Listing 2.4) [Chacon 2009, K. 9.2].

Listing 2.4: Inhalt eines TREE Objektes

```
1 $ git cat-file -p 601a62b205bb497d75a231ec00787f5b2d42c5fc
2 040000 tree f4f5562f575ac208eac980a0cd1c46d874e37298  images
3 040000 tree 61e121cc69e523a68212227f5642fe9b692f5639  diagrams
4 100644 blob d4ada98ad3542643a3c6bb8d25ccce0bc85614fb  00_title.src.md
5 100644 blob 5c14fdfdebc8a52b74b529689714a1a6d7d2f4d1  01_introduction.
    src.md
6 ...
```

COMMIT Der COMMIT enthält eine Referenz auf den TREE des Stammverzeichnisses zu einem bestimmten Zeitpunkt.

Listing 2.5: Inhalt eines COMMIT Objektes

```
1 $ git show -s --pretty=raw 6031a1aa
2 commit 6031a1aa3ea39bbf92a858f47ba6bc87a76b07e8
3 tree 601a62b205bb497d75a231ec00787f5b2d42c5fc
4 parent 8982aa338637e5654f7f778eedf844c8be8e2aa3
5 author Johannes <johannes.wachter@example.at> 1429190646 +0200
6 committer Johannes <johannes.wachter@example.at> 1429190646 +0200
7
8     added short description gridfs and xtreemfs
```

Ein COMMIT Objekt enthält folgende Werte (siehe Listing 2.5):

Tabelle 2.1.: Eigenschaften eines COMMIT [Chacon 2009, K. 9.2]

Zeile	Name	Beschreibung
2	commit	SHA des Objektes
3	tree	TREE-SHA des Stammverzeichnisses
4	parent(s)	Ein oder mehrere Vorgänger
5	author	Verantwortlicher für die Änderungen
6	committer	ErstellerIn des COMMITs
8	comment	Beschreibung des COMMITs

Anmerkungen (zu der Tabelle 2.1):

- Ein COMMIT kann mehrere Vorgänger haben. Dieser Mechanismus würde zum Beispiel bei einem MERGE verwendet werden, um die beiden Vorgänger zu speichern, die zusammengeführt wurden.
- AutorIn und ErstellerIn des COMMITs können sich unterscheiden: Wenn zum Beispiel eine BenutzerIn einen PATCH erstellt, ist diese BenutzerIn die AutorIn und damit die Verantwortliche für die Änderungen. Die BenutzerIn, die den Patch nun auflöst und den `git commit` Befehl ausführt, ist die ErstellerIn bzw. der Committer.

REFERENCE ist ein Verweis auf ein bestimmtes COMMIT-Objekt. Diese Referenzen sind die Grundlage für das Branching-Modell von GIT [Chacon 2015].

In der Abbildung 2.4 wird ein einfaches Beispiel für ein Repository visualisiert. Die Ordnerstruktur, die dieses Beispiel enthält, ist im Listing 2.6 dargestellt.

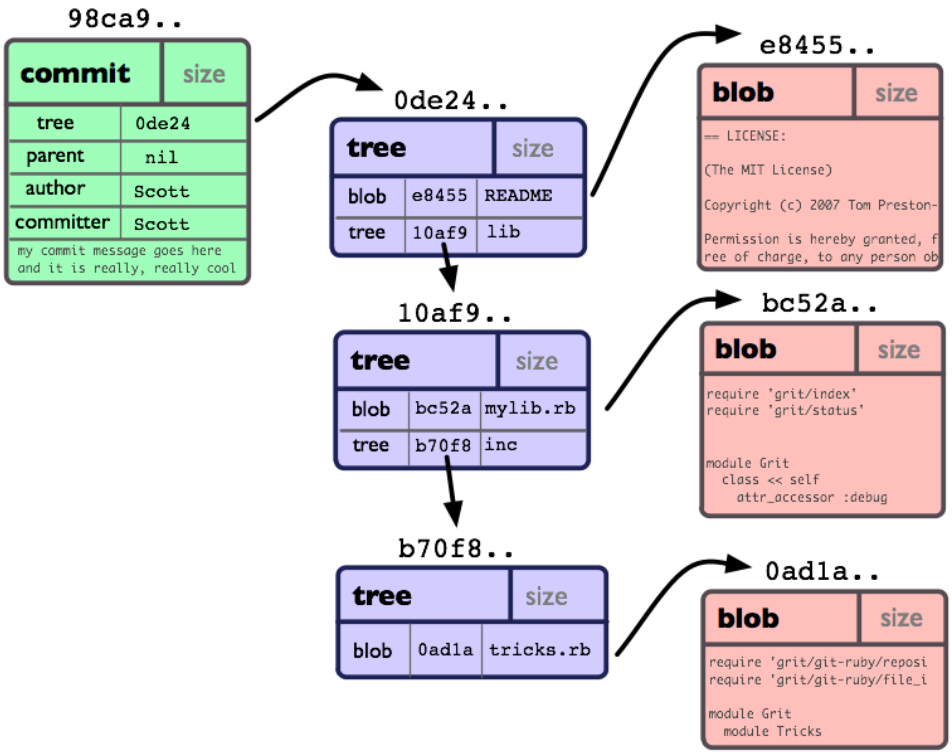

Abbildung 2.4.: Beispiel eines Repositories [Chacon 2015]

Listing 2.6: Ordernstruktur zum Repository Beispiel

```
1 |-- README (Datei)
2 |-- lib (Ordner)
3     |-- inc (Ordner)
4     |   |-- tricks.rb (Datei)
5     |-- mylib.rb (Datei)
```

Der COMMIT (98ca9..) enthält eine Referenz auf den ROOT-TREE (0de24..). Dieser TREE enthält weitere Referenzen zu einem TREE-Objekt (10af9..) mit dem Namen `lib` und dem BLOB-Objekt (e8455..) mit dem Namen README. Diese Struktur wird bis zum TREE-Objekt (b70f8..) fortgesetzt, welches eine Referenz auf den BLOB (0ad1a..) mit dem Namen tricks.rb enthält. Das Beispiel visualisiert, wie komplexe Ordnerstrukturen in GIT verwaltet und gespeichert werden können.

Die Nachteile von GIT im Bezug auf die im Kapitel 1.4 aufgezählten Anforderungen sind:

Architektur Die Architektur von GIT ist im Grunde ein ausgezeichnetes Beispiel für die Verteilung der Daten. Auch das Datenmodell ist optimal für die Verteilung ausgelegt.

Jedoch besitzt GIT keine Mechanismen um die Verteilung zu automatisieren. Die Verteilung erfolgt bei GIT immer als bewusste Aktion, also der Ausführung eines `push` oder `pull` Befehls, der BenutzerInnen. Ein weiteres Problem ist die fehlende Möglichkeit Zugriffsberechtigungen festzulegen.

Aufgrund der fehlenden Verteilungsmechanismen ist die Anwendung GIT für die Verwendung als Datenspeicher für das Projekt ungeeignet. Da es jedoch viele der Anforderungen erfüllt, wird dieses Datenmodell als Grundlage für symCloud herangezogen, siehe dazu Kapitel 4.2. Außerdem wird die Idee der Key-Value Datenbank bei der Konzeption der Datenbank im Kapitel 4.3 aufgegriffen.

2.5. Zusammenfassung

In diesem Kapitel wurden zuerst die Begriffe "verteilte Systeme" und "verteilte Dateisysteme" definiert. Diese Begriffe werden in den folgenden Kapiteln in dem hier beschriebenen Kontext verwendet. Anschließend wurden aktuelle Systeme anhand der Kriterien betrachtet, die für symCloud von Interesse sind. Jedes dieser Systeme bietet Ansätze, die bei der Konzeption von symCloud berücksichtigt werden.

Dropbox Kommerzielles Produkt mit gewünschtem Funktionsumfang hinsichtlich der Dateisynchronisierung und Benutzerinteraktion. Allerdings in Bezug auf das Projekt symCloud aufgrund des nicht verfügbaren Quellcodes nur als Inspirationsquelle verwendbar.

ownCloud Dieses quelloffene Projekt ist eine gute Alternative zu kommerziellen Lösungen wie zum Beispiel Dropbox. Es ist eine der Inspirationsquellen für symCloud, jedoch aufgrund des überholten Programmierstils und der fehlenden verteilten Architektur, ist es nicht als Grundlage für symCloud verwendbar.

Diaspora Das quelloffene Social-Media Projekt ist ein Pionier in Sachen verteilter Architektur. Das Protokoll, welches von Diaspora eingesetzt wird, dient als weitere Inspiration für die Entwicklung von symCloud.

GIT Aufgrund des Datenmodells von GIT ist diese Versionsverwaltung für die verteilte Anwendung optimal ausgerüstet. Daher wird es als Grundlage für symCloud dienen. Es ermöglicht den verbundenen Servern (Clients) eine schnelle und einfache Synchronisation der Daten.

Fazit: Jedes dieser vier Systeme bietet Ansätze, die für die Entwicklung von symCloud relevant sind. Jedoch ist keines dieser Systeme geeignet die Anforderungen aus Kapitel 1.4 vollständig zu erfüllen.

3. Evaluation bestehender Technologien für Speicherverwaltung

Ein wichtiger Aspekt von Cloud-Anwendungen ist die Speicherverwaltung. Es bieten sich verschiedenste Möglichkeiten der Datenhaltung in der Cloud an. Dieses Kapitel beschäftigt sich mit der Evaluierung von verschiedenen Diensten bzw. Lösungen, mit denen Speicher verwaltet und möglichst effizient zur Verfügung gestellt werden kann.

Aufgrund der Anforderungen - siehe Kapitel 1.4 des Projektes - werden folgende Kriterien an die Speicherlösung gestellt.

Ausfallsicherheit Die Speicherlösung ist das Fundament jeder Cloud-Anwendung. Ein Ausfall dieser Schicht bedeutet oft einen Ausfall der kompletten Anwendung.

Skalierbarkeit Die Datenmengen einer Cloud-Anwendung sind oft schwer abschätzbar und können sehr große Ausmaße annehmen. Daher ist die Skalierbarkeit eine wichtige Anforderung an eine Speicherlösung.

Datenschutz Der Datenschutz ist ein wichtiger Punkt beim Betreiben der eigenen Cloud-Anwendung. Meist gibt es eine kommerzielle Konkurrenz, die mit günstigen Preisen die AnwenderInnen anlockt, um dann ihre Daten zu verwenden. Damit SystemadministratorInnen nicht auf einen Provider angewiesen sind, sollte die Möglichkeit bestehen, Daten privat auf dem eigenen Server zu speichern.

Flexibilität Um Daten flexibel speichern zu können, sollte es möglich sein, Verlinkungen und Metadaten direkt in der Speicherlösung abzulegen. Dies erleichtert die Implementierung der eigentlichen Anwendung.

Versionierung Eine optionale Eigenschaft ist die integrierte Versionierung der Daten. Dies würde eine Vereinfachung der Anwendungslogik ermöglichen, da Versionen nicht in einem separaten Speicher abgelegt werden müssen.

Performance Die Performance ist ein wichtiger Aspekt einer Speicherverwaltung. Sie kann zwar durch Caching-Mechanismen verbessert werden, jedoch ist es ziemlich aufwändig diese Caches immer aktuell zu halten. Daher sollten diese Caches nur für "nicht veränderbare" Daten verwendet werden, um den Aufwand für eine Aktualisierung zu reduzieren.

3.1. Datenhaltung in Cloud-Infrastrukturen

Es gibt unzählige Möglichkeiten die Datenhaltung in Cloud-Infrastrukturen umzusetzen. In diesem Kapitel werden drei grundlegende Technologien mit Hilfe von Beispielen analysiert.

Objekt-Speicherdienste Speicherdienste wie zum Beispiel Amazon S3[1], ermöglichen das Speichern von Objekten (Dateien, Ordner und Metadaten). Sie sind optimiert für den parallelen Zugriff von mehreren Instanzen einer Anwendung, die auf verschiedenen Hosts installiert sind. Erreicht wird dies durch eine webbasierte HTTP-Schnittstelle, wie bei Amazon S3 [Amazon-Web-Services 2015b].

Verteilte Dateisysteme Diese Dateisysteme fungieren als einfache Laufwerke und abstrahieren dadurch den komplexen Ablauf der darunter liegenden Services. Der Zugriff auf diese Dateisysteme erfolgt meist über system-calls wie zum Beispiel `fopen` oder `fclose`. Dies ergibt sich aus der Transparenz-Anforderung [Coulouris; Dollimore; Kindberg 2003, S. 369], die im Kapitel 3.3.1 beschrieben werden.

Datenbank gestützte Dateisysteme Erweiterungen zu Datenbanken wie zum Beispiel GridFS[2] von MondoDB können verwendet werden, um große Dateien effizient und sicher in der Datenbank abzulegen [MongoDB 2015].

Aufgrund der vielfältigen Anwendungsmöglichkeiten werden zu jedem der drei Technologien ein oder zwei Beispiele als Referenz dargestellt.

3.2. Amazon Simple Storage Service (S3)

Amazon Simple Storage Service bietet Entwicklern einen sicheren, beständigen und sehr gut skalierbaren Objektspeicher. Es dient der einfachen und sicheren Speicherung großer

[1] http://aws.amazon.com/de/s3/
[2] http://docs.mongodb.org/manual/core/gridfs/

Datenmengen [Amazon-Web-Services 2015a]. Daten werden in sogenannte “Buckets” gegliedert. Jeder Bucket kann unbegrenzt Objekte enthalten. Die Gesamtgröße der Objekte ist jedoch auf 5TB beschränkt. Sie können nicht verschachtelt werden, allerdings können sie Ordner enthalten, um die Objekte zu gliedern.

Die Kernfunktionalität des Services besteht darin, Daten in sogenannten Objekten zu speichern. Diese Objekte können bis zu 5GB groß werden. Zusätzlich wird zu jedem Objekt ca. 2KB Metadaten abgelegt. Bei der Erstellung eines Objektes werden automatisch vom System Metadaten erstellt. Einige dieser Metadaten können vom Benutzer überschrieben werden, wie zum Beispiel `x-amz-storage-class`, andere werden vom System automatisch gesetzt, wie zum Beispiel `Content-Length`. Diese systemspezifischen Metadaten werden beim Speichervorgang auch automatisch aktualisiert [Amazon-Web-Services 2015c]. Für eine vollständige Liste dieser Metadaten siehe Anhang A.

Zusätzlich zu diesen systemdefinierten Metadaten ist es möglich, benutzerdefinierte Metadaten abzulegen. Das Format dieser Metadaten entspricht einer Key-Value Liste. Diese Liste ist auf 2KB limitiert.

3.2.1. Versionierung

Die Speicherlösung bietet eine Versionierung der Objekte an. Diese kann über eine Rest-API, mit folgendem Inhalt (siehe Listing 3.1), in jedem Bucket aktiviert werden.

Listing 3.1: Aktiviert die Versionierung für ein Objekt [Amazon-Web-Services 2015c]

```
1 <VersioningConfiguration
2     xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
3   <Status>Enabled</Status>
4 </VersioningConfiguration>
```

Ist die Versionierung aktiviert, gilt diese für alle Objekte, die dieser Bucket enthält. Wird anschließend ein Objekt überschrieben, resultiert dies in einer neuen Version, dabei wird die Version-ID im Metadaten Feld `x-amz-version-id` auf einen neuen Wert gesetzt [Amazon-Web-Services 2015d]. Dies veranschaulicht die Abbildung 3.1.

Abbildung 3.1.: Versionierungsschema von Amazon S3 [Amazon-Web-Services 2015d]

3.2.2. Skalierbarkeit

Die Skalierbarkeit ist aufgrund der von Amazon verwalteten Umgebung sehr einfach. Es wird soviel Speicherplatz zur Verfügung gestellt, wie benötigt wird. Der Umstand, dass mehr Speicherplatz benötigt wird, zeichnet sich nur auf der Rechnung des Betreibers ab.

3.2.3. Datenschutz

Amazon ist ein US-amerikanisches Unternehmen und ist daher an die Weisungen der Amerikanischen Geheimdienste gebunden. Aus diesem Grund wird es in den letzten Jahren oft kritisiert. Laut einem Bericht der ITWorld beteuerte Terry Wise[3], dass jede gerichtliche Anordnung zur Einsicht in die Daten mit dem Kunden abgesprochen wird [Gohring 2013]. Dies gilt aber vermutlich nicht für Anfragen der NSA, da sich diese in der Regel auf die Anti-Terror Gesetze berufen. Diese verpflichten den Anbieter zur absoluten Schweigepflicht. Um dieses Problem zu umgehen können Systemadministratoren sogenannte “Availability Zones” auswählen und damit steuern, wo die Daten gespeichert werden. Zum Beispiel werden Daten in einem Bucket mit der Zone Irland auch wirklich in Irland gespeichert. Zusätzlich ermöglicht Amazon die Verschlüsselung der Daten [Fuchs 2013].

Wer trotzdem Bedenken hat, seine Daten aus den Händen zu geben, kann auf verschiedene kompatible Lösungen zurückgreifen.

[3] Amazons Zuständiger für die Zusammenarbeit zwischen den Partner

3.2.4. Alternativen zu Amazon S3

Es gibt einige Amazon S3 kompatible Anbieter, die einen ähnlichen Dienst anbieten. Diese sind allerdings meist auch US-Amerikanische Firmen und daher an dieselben Gesetze wie Amazon gebunden. Wer daher auf Nummer sicher gehen will und seine Daten bzw. Rechner-Instanzen ganz bei sich behalten will, kommt nicht um die Installation einer privaten Cloud-Lösung herum.

Eucalyptus Eucalyptus ist eine Open-Source-Infrastruktur zur Nutzung von Cloud-Computing auf einem Rechner Cluster. Der Name ist ein Akronym für "Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems". Die hohe Kompatibilität macht diese Software-Lösung zu einer optimalen Alternative zu Amazon-Web-Services. Es bietet neben Objektspeicher auch andere AWS kompatible Dienste an, wie zum Beispiel EC2 (Rechnerleistung) oder EBS (Blockspeicher) [Hewlett-Packard 2013].

Riak Cloud Storage Riak Cloud Storage ist eine Software, mit der es möglich ist einen verteilten Objekt-Speicherdienst zu betreiben. Diese implementiert die Schnittstelle von Amazon S3 und ist damit kompatibel zu der aktuellen Version [Basho Technologies 2015]. Riak Cloud Storage unterstützt die meisten Funktionalitäten die Amazon bietet. Die Installation von Riak-CS ist im Gegensatz zu Eucalyptus sehr einfach und kann daher auf nahezu jedem System durchgeführt werden.

Beide vorgestellten Dienste bieten momentan keine Möglichkeit Objekte zu versionieren. Außerdem ist das Vergeben von Berechtigungen nicht so einfach möglich wie bei Amazon S3. Diese Aufgabe muss von der Applikation, die diese Dienste verwendet, übernommen werden.

3.2.5. Performance

HostedFTP veröffentlichte im Jahre 2009 in einem Report ihre Erfahrungen mit der Performance zwischen EC2 (Rechner Instanzen) und S3 [HostedFTP 2009]. Über ein Performance Modell wurde festgestellt, dass die Zeit für den Up- / Download einer Datei in zwei Bereiche aufgeteilt werden kann.

Feste Transaktionszeit Die feste Transaktionszeit ist eine fixe Zeitspanne, die für die Bereitstellung bzw. Erstellung der Datei benötigt wird. Die Dateigröße beein-

flusst diese Zeit kaum, allerdings kann es aufgrund schwankender Auslastung zu Verzögerungen kommen.

Downloadzeit Die Downloadzeit ist linear abhängig zu der Dateigröße und kann aufgrund der Bandbreite schwanken.

Ausgehend von diesen Überlegungen kann davon ausgegangen werden, dass die Upload- bzw. Downloadzeit einen linearen Verlauf proportional zur Dateigröße aufweist. Diese These wird von den Daten unterstützt. Aus dem Diagramm (Abbildung 3.2) kann die feste Transaktionszeit von ca. 140ms abgelesen werden.

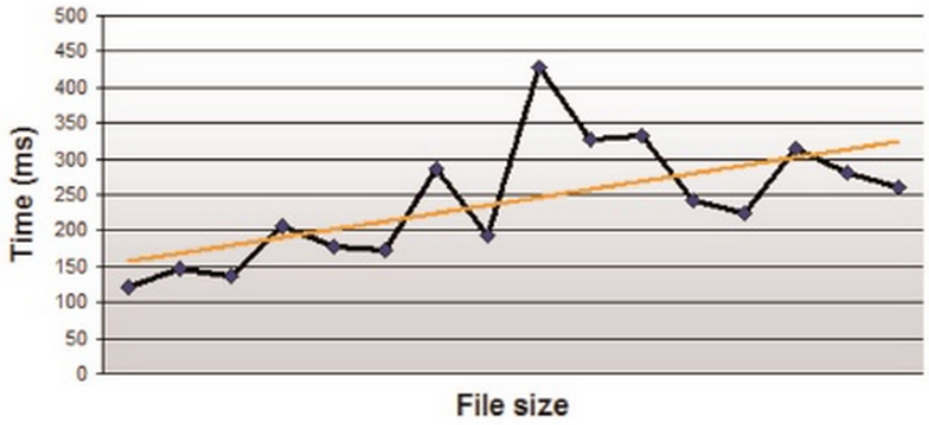

Abbildung 3.2.: Upload Analyse zwischen EC2 und S3 [HostedFTP 2009]

Für den Download von Dateien entsteht laut den Daten aus dem Auswertungen keine fixe Transaktionszeit. Die Zeit für den Download ist also nur von der Größe der Datei und der Bandbreite abhängig.

3.3. Verteilte Dateisysteme

Verteilte Dateisysteme unterstützen die gemeinsame Nutzung von Informationen in Form von Dateien. Sie bieten Zugriff auf Dateien, die auf einem entfernten Server abgelegt sind, wobei eine ähnliche Leistung und Zuverlässigkeit erzielt wird, wie für lokal gespeicherte Daten. Wohldurchdachte verteilte Dateisysteme erzielen oft sogar bessere Ergebnisse in Leistung und Zuverlässigkeit als lokale Systeme. Die entfernten Dateien werden genauso verwendet wie lokale Dateien, da verteilte Dateisysteme die Schnittstelle des Betriebssystems emulieren. Dadurch können die Vorteile von verteilten Systemen in einem Programm

genutzt werden, ohne dieses anpassen zu müssen. Die Schreibzugriffe bzw. Lesezugriffe erfolgen über ganz normale “system-calls” [Coulouris; Dollimore; Kindberg 2003, S. 363ff].

Diese Abstraktion ist ein großer Vorteil der verteilten Dateisysteme im Vergleich zu Speicherdiensten wie Amazon S3. Da die Schnittstelle zu den einzelnen Systemen abstrahiert wird, muss die Software nicht angepasst werden, wenn das Dateisystem gewechselt wird.

3.3.1. Anforderungen

Die Anforderungen an verteilte Dateisysteme lassen sich wie folgt zusammenfassen.

Zugriffstransparenz Client-Programme sollten, egal ob lokal oder verteilt, über dieselbe Operationsmenge verfügen. Es sollte irrelevant sein ob Daten aus einem verteilten oder lokalen Dateisystem stammen. Dadurch können Programme unverändert weiterverwendet werden, obwohl dessen Dateien verteilt werden [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

Ortstransparenz Es sollte keine Rolle spielen, wo die Daten physikalisch gespeichert sind [Schütte 2014, S. 5]. Das Programm sieht immer denselben Namensraum, egal von wo aus es ausgeführt wird [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

Nebenläufige Dateiaktualisierungen Dateiänderungen die von einem Client ausgeführt werden, sollten die Operationen anderer Clients, die dieselbe Datei verwenden, nicht stören. Um diese Anforderung zu erfüllen muss eine funktionierende Nebenläufigkeitskontrolle implementiert werden. Die meisten aktuellen Dateisysteme unterstützen freiwillige oder zwingende Sperren auf Datei- oder Datensatzebene.

Dateireplikationen Unterstützt ein Dateisystem Dateireplikationen, kann ein Datensatz durch mehrere Kopien des Inhalts an verschiedenen Positionen dargestellt werden. Das bietet zwei Vorteile: Lastverteilung durch mehrere Server und Erhöhung der Fehlertoleranz. Nur wenige Dateisysteme unterstützen vollständige Replikationen, die meisten unterstützen jedoch ein lokales Caching von Dateien, welches eine eingeschränkte Art der Dateireplikation darstellt [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

Fehlertoleranz Da der Dateidienst normalerweise der meist genutzte Dienst in einem Netzwerk ist, ist es unabdingbar, dass er auch dann weiter ausgeführt wird, wenn einzelne Server oder Clients ausfallen. Ein Fehlerfall sollte zumindest nicht zu Inkonsistenzen führen [Schütte 2014, S. 5].

Konsistenz In konventionellen Dateisystemen werden Zugriffe auf Dateien auf eine einzige Kopie der Daten geleitet. Wird nun diese Datei auf mehreren Servern verteilt, müssen die Operationen an alle Server weitergeleitet werden. Die Verzögerung die dabei auftritt, führt in dieser Zeit zu einem inkonsistenten Zustand des Systems [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

Sicherheit Fast alle Dateisysteme unterstützen eine Art Zugriffskontrolle auf die Dateien. Dies ist ungleich wichtiger, wenn viele Benutzer gleichzeitig auf Dateien zugreifen. In verteilten Dateisystemen besteht der Bedarf die Anforderungen des Clients auf korrekte Benutzer-IDs umzuleiten, die dem System bekannt sind [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

Effizienz Verteilte Dateisysteme sollten sowohl in Bezug auf die Funktionalitäten als auch auf die Leistung, mit konventionellen Dateisystemen vergleichbar sein [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

Andrew Birrell und Roger Needham setzten sich folgende Entwurfsziele für Ihr Universal File System [Birrell; Needham 1980]:

> "We would wish a simple, low-level, file server in order to share an expensive resource, namely a disk, whilst leaving us free to design the filing system most appropriate to a particular client, but we would wish also to have available a high-level system shared between clients."

Aufgrund der Tatsache, dass Festplatten heutzutage nicht mehr so teuer wie in den 1980ern sind, ist das erste Ziel nicht mehr von zentraler Bedeutung. Die Vorstellung von einem Dienst, der die Anforderung verschiedenster Clients mit unterschiedlichen Aufgabenstellungen erfüllt, ist ein zentraler Aspekt der Entwicklung von verteilten (Datei-) Systemen [Coulouris; Dollimore; Kindberg 2003, S. 369ff].

3.3.2. NFS

Das verteilte Dateisystem Network File System wurde von Sun Microsystems entwickelt. Das grundlegende Prinzip von NFS ist, dass jeder Dateiserver eine standardisierte Dateischnittstelle implementiert. Über diese Schnittstelle werden Dateien des lokalen Speichers den BenutzerInnen zur Verfügung gestellt. Das bedeutet, dass es keine Rolle spielt welches System dahinter steht. Ursprünglich wurde es für UNIX Systeme entwickelt. Mittlerweile gibt es aber Implementierungen für verschiedenste Betriebssysteme

[Tanenbaum; Steen 2003, S. 645ff].

NFS ist dennoch weniger ein Dateisystem als eine Menge von Protokollen, die in der Kombination mit den Clients ein verteiltes Dateisystem ergeben. Die Protokolle wurden so entwickelt, dass unterschiedliche Implementierungen einfach zusammenarbeiten können. Auf diese Weise kann durch NFS eine heterogene Menge von Computern verbunden werden. Dabei ist es sowohl für die BenutzerInnen als auch für den Server irrelevant mit welcher Art von System er verbunden ist [Tanenbaum; Steen 2003, S. 645ff].

Architektur

Das zugrundeliegende Modell von NFS ist das eines entfernten Dateidienstes. Dabei erhält ein Client den Zugriff auf ein transparentes Dateisystem, das von einem entfernten Server verwaltet wird. Dies ist vergleichbar mit RPC[4]. Der Client erhält den Zugriff auf eine Schnittstelle, um auf Dateien zuzugreifen, die ein entfernter Server implementiert [Tanenbaum; Steen 2003, S. 647ff].

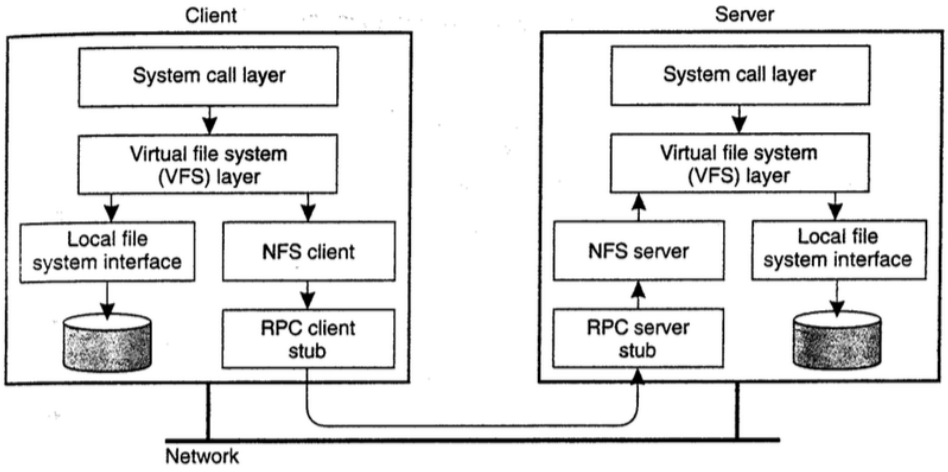

Abbildung 3.3.: NFS Architektur [Tanenbaum; Steen 2003, S. 647]

Der Client greift über die Schnittstelle des lokalen Betriebssystems auf das Dateisystem zu. Die lokale Dateisystemschnittstelle wird jedoch durch ein virtuelles Dateisystem ersetzt (VFS), die eine Schnittstelle zu den verschiedenen Dateisystemen darstellt. Das VFS entscheidet anhand der Position im Dateibaum, ob die Operation an das lokale Dateisystem oder an den NFS-Client weitergegeben wird (siehe Abbildung 3.3). Der NFS-Client ist eine separate Komponente, die sich um den Zugriff auf entfernte Dateien kümmert. Dabei fungiert der Client als eine Art Stub-Implementierung der Schnittstelle

[4]Remote Procedure Calls http://www.cs.cf.ac.uk/Dave/C/node33.html

und leitet alle Anfragen an den entfernten Server weiter (RPC). Diese Abläufe werden aufgrund des VFS-Konzeptes vollkommen transparent für die BenutzerIn durchgeführt [Tanenbaum; Steen 2003, S. 647ff].

3.3.3. XtreemFS

Als Alternative zu konventionellen verteilten Dateisystemen bietet XtreemFS eine unkomplizierte und moderne Variante eines verteilten Dateisystems an. Es wurde speziell für die Anwendung in einem Cluster mit dem Betriebssystem XtreemOS entwickelt. Mittlerweile gibt es aber Server- und Client-Anwendungen für fast alle Linux Distributionen. Außerdem Clients für Windows und MAC.

Die Hauptmerkmale von XtreemFS sind:

Distribution Eine XtreemFS Installation enthält eine beliebige Anzahl an Servern, die auf verschiedenen Maschinen betrieben werden können. Diese Server, sind entweder über einen lokalen Cluster oder über das Internet miteinander verbunden. Der Client kann sich mit einem beliebigen Server verbinden und mit ihm Daten austauschen. Es garantiert konsistente Daten, auch wenn verschiedene Clients mit verschiedenen Servern kommunizieren. Vorausgesetzt ist, dass alle Komponenten miteinander verbunden und erreichbar sind [Kolbeck 2014, K. 2.3].

Replikation Die drei Hauptkomponenten von XtreemFS, Directory Service, Metadata-Catalog und die Object-Storage-Devices (siehe Abbildung 3.4) können redundant verwendet werden, dies führt zu einem fehlertoleranten System. Die Replikationen zwischen diesen Systemen erfolgen mit einem Hot-Backup (siehe Kapitel 3.3.4). Dieses Backup springt automatisch ein, wenn ein anderer Server ausfällt [Kolbeck 2014, K. 2.3].

Striping XtreemFS splittet Dateien in sogenannte “Stripes” (oder “Chunks”). Diese Chunks werden auf verschiedenen Servern gespeichert und können dann parallel von mehreren Servern gelesen werden. Die gesamte Datei kann mit der zusammengefassten Netzwerk- und Festplatten-Bandbreite mehrerer Server heruntergeladen werden. Die Größe der Chunks und die Anzahl der Server, auf denen die Chunks repliziert werden, kann pro Datei bzw. pro Ordner festgelegt werden [Kolbeck 2014, K. 2.3].

Security Um die Sicherheit der Dateien zu gewährleisten, unterstützt XtreemFS sowohl Authentifizierung als auch ein Berechtigungssystem. Der Netzwerkverkehr zwischen

den Servern ist verschlüsselt. Die Standard Authentifizierung basiert auf lokalen Benutzernamen und ist auf die Vertrauenswürdigkeit der Clients bzw. des Netzwerkes angewiesen. Um mehr Sicherheit zu erreichen, unterstützt XtreemFS aber auch eine Authentifizierung mittels X.509 Zertifikaten[5] [Kolbeck 2014, K. 2.3].

Architektur

XtreemFS implementiert eine objektbasierte Datei-Systemarchitektur was bedeutet, dass die Dateien in Objekte mit einer bestimmten Größe aufgeteilt werden und die Objekte auf verschiedenen Servern gespeichert werden. Die Metadaten werden in separaten Datenbankservern gespeichert. Diese Server organisieren die Dateien in eine Menge von sogenannten "Volumes". Jedes Volume ist ein eigener Namensraum mit einem eigenen Dateibaum, vergleichbar mit den Buckets von Amazon S3. Die Metadaten speichern zusätzlich eine Liste von Chunk-IDs mit den jeweiligen Servern, auf denen dieser Chunk zu finden ist. Außerdem legt eine Richtlinie fest, wie diese Datei auf gesplittet und auf die Server verteilt werden soll. Daher kann die Größe der Metadaten von Datei zu Datei unterschiedlich sein [Kolbeck 2014, K. 2.4].

Eine XtreemFS Installation besteht aus drei Komponenten (siehe Abbildung 3.4):

DIR Das "Directory-Service" ist das zentrale Register indem alle anderen Services aufgelistet werden. Die Clients oder andere Services verwenden diesen, um zum Beispiel die "Object-Storage-Devices" oder "Metadata- and Replica-Catalogs" zu finden [Kolbeck 2014, K. 2.4].

MRC Der "Metadata- and Replica-Catalog" verwaltet die Metadaten der Datei, wie zum Beispiel Dateiname, Dateigröße oder Bearbeitungsdatum. Zu jeder Datei kann außerdem spezifiziert werden, auf welchen "Object-Storage-Devices" die Chunks abgelegt wurden. Zusätzlich authentifiziert und autorisiert er dem Benutzer den Zugriff auf die Dateien bzw. Ordner [Kolbeck 2014, K. 2.4].

OSD Das "Object-Storage-Device" speichert die Objekte ("Strip", "Chunks" oder "Blobs") der Dateien. Die Clients schreiben und lesen Daten direkt von diesen Servern [Kolbeck 2014, K. 2.4].

[5] http://tools.ietf.org/html/rfc5280

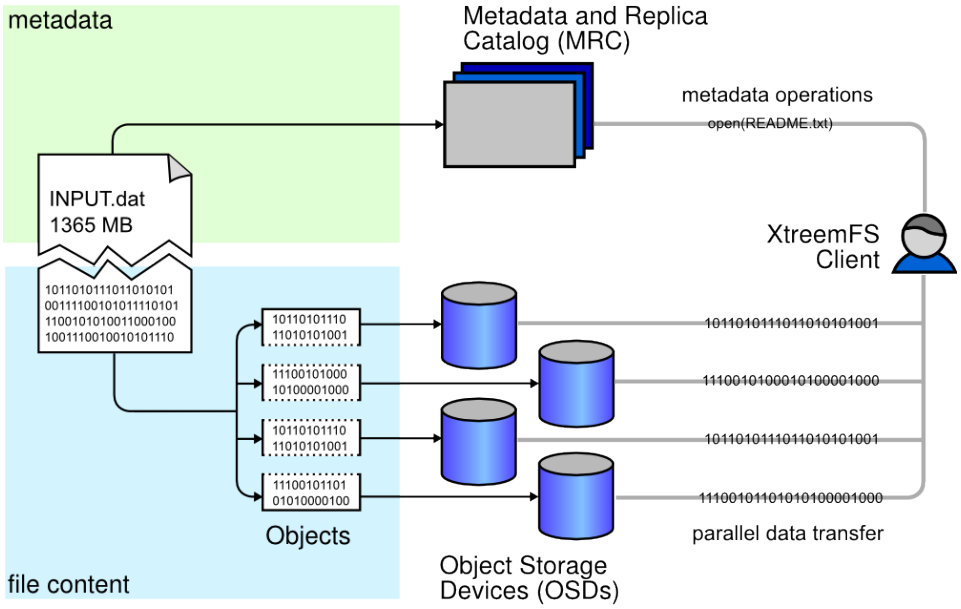

Abbildung 3.4.: XtreemFS Architektur [XtreemFS 2014b]

3.3.4. Exkurs: Datei Replikation

Ein wichtiger Aspekt von verteilten Dateisystemen ist die Replikation von Daten. Sie steigert sowohl die Zuverlässigkeit, als auch die Leistung der Lesezugriffe. Das größte Problem dabei ist allerdings, die Konsistenz der Repliken zu erhalten. Dabei muss bei jedem schreibenden Zugriff ein Update aller Repliken erfolgen, ansonsten ist die Konsistenz nicht mehr gegeben [Tanenbaum; Steen 2003, S. 333ff].

Die Hauptgründe für Replikationen von Daten sind Zuverlässigkeit und Leistung [Gray u. a. 1996]. Wenn Daten repliziert werden ist es unter Umständen möglich weiterzuarbeiten, wenn eine Replik ausfällt. Die BenutzerIn lädt sich die Daten von einem anderen Server herunter. Zusätzlich können durch Repliken fehlerhafte Dateien erkannt werden. Wenn eine Datei zum Beispiel auf drei Servern gespeichert wurde und auf allen drei Server Schreib- bzw. Lesezugriffe ausgeführt werden, kann durch den Vergleich der Antworten, erkannt werden ob eine Datei fehlerhaft ist. Dazu müssen nur zwei Antworten denselben Inhalt besitzen und es kann davon ausgegangen werden, dass es sich um die richtige Datei handelt [Tanenbaum; Steen 2003, S. 333ff].

Der andere wichtige Grund für Replikationen ist die Leistung des Systems. Hier gibt es zwei Aspekte: Netzwerklast und die geographische Lage. Wenn ein System nur aus einem Server besteht, ist dieser Server der vollen Last der Zugriffe ausgesetzt. Teilt man diese Last auf, kann die Leistung des Systems gesteigert werden. Zusätzlich kann

durch Repliken auch die Geschwindigkeit der Lesezugriffe gesteigert werden, indem dieser Zugriff über mehrere Server parallel erfolgt. Auch die geographische Lage der Daten spielt bei der Leistung des Systems eine entscheidende Rolle. Wenn Daten in der Nähe des Prozesses gespeichert werden, in dem sie erzeugt bzw. verwendet werden, ist sowohl der schreibende als auch der lesende Zugriff schneller möglich. Diese Leistungssteigerung ist allerdings nicht linear zu den verwendeten Servern. Es ist einiges an Aufwand zu betreiben, um diese Repliken synchron zu halten und dadurch die Konsistenz zu wahren [Tanenbaum; Steen 2003, S. 333ff].

Damit ein Verbund von Servern die Konsistenz ihrer Daten gewährleisten kann, werden Konsistenzprotokolle eingesetzt. In XtreemFS wird ein sogenanntes primärbasiertes Protokoll verwendet [Kolbeck 2014, K. 6]. In diesen Protokollen ist jedem Datenelement "x" ein primärer Server zugeordnet, der dafür verantwortlich ist Schreiboperationen für "x" zu koordinieren. Es gibt zwei Arten dieses Protokoll umzusetzen: Entferntes- und Lokales-Schreiben.

Entferntes-Schreiben

Es gibt zwei Arten zur Implementierung dieses Protokolls. Das eine ist ein nicht replizierendes Protokoll, bei dem alle Schreib- bzw. Lesezugriffe auf den primären Server des Objektes ausgeführt werden. Das andere ist das sogenannte "Primary-Backup" Protokoll, welches über einen festen primären Server für jedes Objekt verfügt. Dieser Server wird bei der Erstellung des Objektes festgelegt und nicht verändert. Zusätzlich wird festgelegt, auf welchen Servern Repliken für dieses Objekt angelegt werden. In XtreemFS werden diese Einstellungen "replication policy" genannt [Kolbeck 2014, K. 6.1.3].

Der Prozess, der eine Schreiboperation (siehe Abbildung 3.5) auf das Objekt ausführen will, gibt sie an den primären Server weiter. Dieser führt die Operation lokal an dem Objekt aus und gibt die Aktualisierungen an die Backup-Server weiter. Jeder dieser Server führt die Operation aus und gibt eine Bestätigung an den primären Server zurück. Nachdem alle Backups die Aktualisierung durchgeführt haben, sendet auch der primäre Server eine Bestätigung an den ausführenden Server. Dieser Server kann nun sicher sein, dass die Aktualisierung auf allen Servern ausgeführt und damit sicher im System gespeichert wurde. Durch diesen blockierenden Prozess, kann ein gravierendes Leistungsproblem entstehen. Für Programme, die lange Antwortzeiten nicht akzeptieren können, ist es eine Variante, das Protokoll nicht blockierend zu implementieren. Das bedeutet, dass der primäre Server die Bestätigung direkt nach dem lokalen Ausführen der Operation zurückgibt und erst danach die Aktualisierungen an die Backups weiter

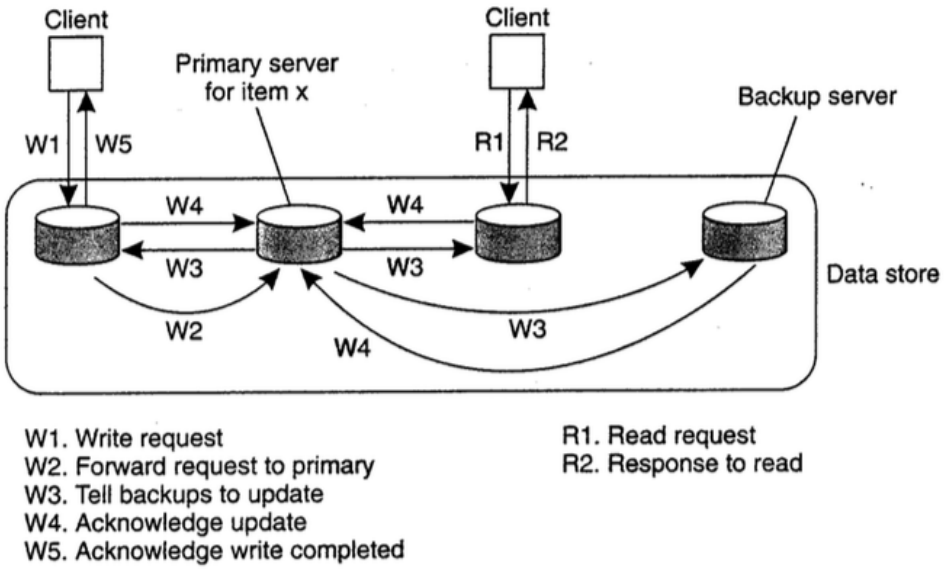

Abbildung 3.5.: Primary-Backup: Entferntes-Schreiben [Tanenbaum; Steen 2003, S. 385]

gibt [XtreemFS 2014a]. Aufgrund der Tatsache, dass alle Schreiboperationen auf einem Server ausgeführt werden, können diese einfach abgesichert und dadurch die Konsistenz gewahrt werden. Eventuelle Transaktionen oder Locks müssen nicht im Netzwerk verteilt werden [Tanenbaum; Steen 2003, S. 384ff].

Lokales-Schreiben

Auch dieses Protokoll kann in zwei verschiedenen Arten implementiert werden. Die erste Variante ist ein nicht replizierendes Protokoll, bei dem das Objekt vor einem Schreibzugriff auf den ausführenden Server verschoben und dadurch der primäre Server des Objekts geändert wird. Nachdem die Schreiboperation ausgeführt wurde, bleibt das Objekt solange auf diesem Server bis ein anderer Server schreibend auf das Objekt zugreifen will. Die andere Variante ist ein "Primäres-Backup Protokoll" (siehe Abbildung 3.6), bei dem der primäre Server des Objektes auf den ausführenden Server migriert wird [Tanenbaum; Steen 2003, S. 386ff].

Dieses Protokoll ist auch für mobile Computer geeignet, die in einem Offline-Modus betrieben werden können. Dazu wird dieser zum primären Server für die Objekte, die er vermutlich während seiner Offline-Phase bearbeiten wird. Während der Offline-Phase können nun Aktualisierungen lokal ausgeführt werden und die anderen Clients können lesend auf eine Replik zugreifen. Sie bekommen zwar keine Aktualisierungen, können aber sonst ohne Einschränkungen weiterarbeiten. Nachdem die Verbindung wiederhergestellt wurde, werden die Aktualisierungen an die Backup-Server weitergegeben, sodass der Datenspeicher

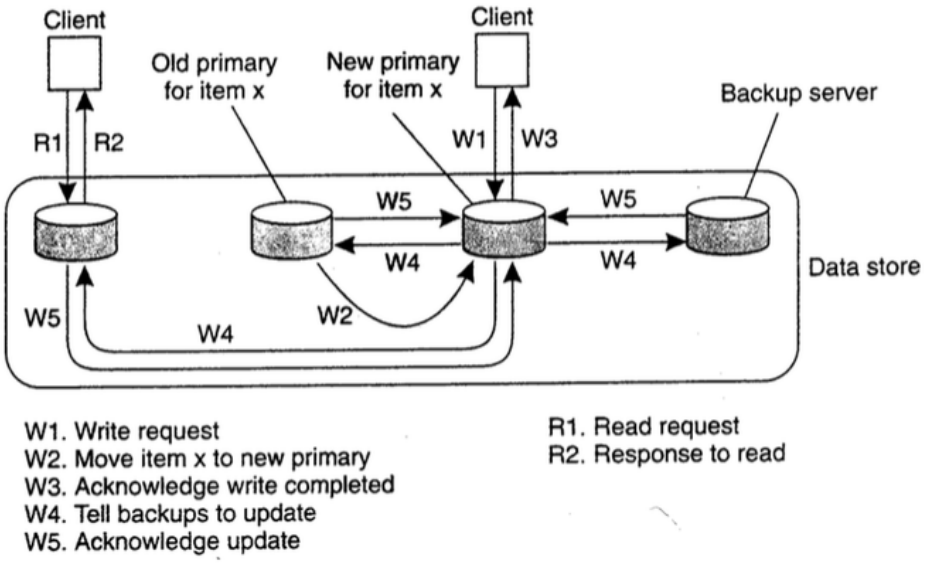

Abbildung 3.6.: Primary-Backup: Lokales-Schreiben [Tanenbaum; Steen 2003, S. 387]

wieder in einen konsistenten Zustand übergehen kann [Tanenbaum; Steen 2003, S. 386ff].

3.3.5. Zusammenfassung

Im Bezug auf die Anforderungen (siehe Kapitel 1.4) bieten die analysierten verteilten Dateisysteme von Haus aus keine Versionierung. Es gab Versuche der Linux-Community, mit Wizbit[6], ein auf GIT-basierendes Dateisystem zu entwerfen, dass die Versionierung mitliefern sollte [Paul 2008]. An diesem Projekt wurde allerdings seit Ende 2009 nicht mehr weiterentwickelt [OpenHUB 2009].

Die benötigten Zugriffsberechtigungen werden zwar auf der Systembenutzerebene durch ACL unterstützt, jedoch müsste die Anwendung für alle BenutzerInnen eine SystembenutzerIn anlegen [Kolbeck 2014, K. 7.2]. Dies wäre zwar mit einer einzelnen Installation machbar, jedoch macht es eine verteilte Anwendung komplizierter und eine Installation aufwändiger. Allerdings können gute Erkenntnisse aus der Analyse der Replikationsmechanismen bzw. der Konsistenzprotokolle von XtreemFS gezogen und in ein Gesamtkonzept mit eingebunden werden.

Die hier beschriebenen Protokolle und Konzepte werden im Kapitel 4.3 aufgegriffen, um die Daten effizient und sicher zwischen den Servern zu verteilen. Dabei werden

[6] https://www.openhub.net/p/wizbit

wesentliche Konzepte des Lokalen-Schreibens verwendet, um das Protokoll für symCloud zu definieren.

3.4. Datenbankgestützte Dateiverwaltungen

Einige Datenbanksysteme, wie zum Beispiel MongoDB[7], bieten eine Schnittstelle um Dateien abzuspeichern. Viele dieser Systeme sind meist nur begrenzt für große Datenmengen geeignet. MongoDB und GridFS sind jedoch genau für diese Anwendungsfälle ausgelegt, daher soll diese Technologie im folgenden Abschnitt genauer betrachtet werden.

3.4.1. MongoDB & GridFS

MongoDB bietet die Möglichkeit BSON-Dokumente in der Größe von bis zu 16MB zu speichern. Dies ermöglicht die Verwaltung kleinerer Dateien ohne zusätzliche Layer. Für größere Dateien und zusätzliche Features bietet MongoDB mit GridFS eine Schnittstelle, mit der es möglich ist größere Dateien und die dazugehörigen Metadaten zu speichern. Dazu teilt GridFS die Dateien in Chunks einer bestimmten Größe auf. Standardmäßig ist die Größe von Chunks auf 255Byte gesetzt. Die Daten werden in der Kollektion `chunks` und die Metadaten in der Kollektion `files` gespeichert. Durch die verteilte Architektur von MongoDB werden die Daten automatisch auf allen Systemen synchronisiert. Zusätzlich bietet das System die Möglichkeit, über Indexe schnell zu suchen und Abfragen auf die Metadaten durchzuführen [MongoDB 2015].

Beispiel:

Listing 3.2: GridFS Beispielcode [Lightcubesolutions 2010]

```
1 $mongo = new Mongo();
2         // connect to database
3 $database = $mongo->selectDB('example');
4         // select mongo database
5 $gridFS = $database->getGridFS();
6         // use GridFS class for handling files
7 $name = $_FILES['Filedata']['name'];
8         // optional - capture the name of the uploaded file
9 $id = $gridFS->storeUpload('Filedata', $name);
```

[7] http://docs.mongodb.org/manual/core/gridfs/

```
10          // load file into MongoDB
```

Bei der Verwendung von MongoDB ist es sehr einfach, Dateien in GridFS (siehe Beispielcode in Listing 3.2) abzulegen. Die fehlenden Funktionen, wie zum Beispiel, ACL oder Versionierung machen den Einsatz in symCloud allerdings schwierig. Auch der starre Aufbau mit nur einem Dateibaum macht die Anpassung der Datenstruktur nahezu unmöglich. Allerdings ist das Chunking der Dateien auch hier zentraler Bestandteil, daher wäre es möglich MongoFS für einen Teil des Speicher-Konzeptes zu verwenden.

3.5. Zusammenfassung

Am Ende dieses Abschnittes, werden die Vor- und Nachteile der jeweiligen Technologien zusammengefasst. Dies ist notwendig, um am Ende ein optimales Speicherkonzept für symCloud zu entwickeln.

Amazon S3 Speicherdienste, wie Amazon S3 sind für einfache Aufgaben bestens geeignet. Sie bieten alles an, was für ein schnelles Setup der Applikation benötigt wird. Jedoch haben gerade die Open-Source Alternativen wesentliche Mankos in Bereichen, die für das Projekt unbedingt notwendig sind. Zum einen sind es bei den Alternativen die fehlenden Funktionalitäten, wie zum Beispiel ACLs oder Versionierung, zum anderen ist auch Amazon S3 wenig flexibel, um eigene Erweiterungen hinzuzufügen. Jedoch können wesentliche Vorteile bei den Objekt-Speicherdiensten beobachtet werden. Wie zum Beispiel:

- Rest-Schnittstelle
- Versionierung
- Gruppierung durch Buckets
- Berechtigungssysteme

Diese Punkte werden im Kapitel 4 berücksichtigt.

Verteilte Dateisysteme Die verteilten Dateisysteme bieten durch ihre einheitliche Schnittstelle einen optimalen Abstraktionslayer für datenintensive Anwendungen. Die Flexibilität, die diese Systeme verbindet, kommen der Anwendung in symCloud entgegen. Jedoch sind fehlende Zugriffsrechte auf Anwendungsbenutzerebene (ACL) und die fehlende Versionierung ein Problem das auf Speicherebene nicht gelöst

wird. Daher kann ein solches verteiltes Dateisystem nicht als Ersatz für eine eigene Implementierung, sondern lediglich als Basis dafür herangezogen werden.

Datenbankgestützte Dateiverwaltung Systeme wie zum Beispiel GridFS sind für den Einsatz in Anwendungen geeignet, die die darunterliegende Datenbank verwenden. Die nötigen Erweiterungen, um Dateien in eine Datenbank zu schreiben, sind aufgrund der Integration sehr einfach umzusetzen. Sie bieten eine gute Schnittstelle, um Dateien zu verwalten. Die fehlenden Möglichkeiten von ACL und Versionierung macht jedoch die Verwendung von GridFS sehr aufwändig. Aufgrund des Aufbaues von GridFS gibt es in der Datenbank einen Dateibaum, indem alle BenutzerInnen ihre Dateien ablegen. Die Anwendung müsste selbst dafür sorgen, dass jede BenutzerIn nur seine Dateien sehen bzw. bearbeiten kann. Allerdings kann, gerade aus GridFS, mit dem Chunking von Dateien (siehe Kapitel 4.5) ein sehr gutes Konzept für eine effiziente Dateihaltung entnommen werden.

Da aufgrund verschiedenster Schwächen keine der Technologien eine umfassende Lösung für die Datenhaltung in symCloud bietet, wird im nächsten Kapitel versucht, ein optimales Speicherkonzept für das Projekt zu entwickeln.

4. Konzeption von symCloud

Dieses Kapitel befasst sich mit der Erstellung eines Speicher- und Architekturkonzeptes für symCloud. Das zentrale Element dieses Konzeptes ist die Objekt-Datenbank. Diese Datenbank unterstützt die Verbindung zu anderen Servern. Damit ist symCloud, als Ganzes gesehen ein verteiltes Dateiverwaltungssystem. Es unterstützt dabei die Replikation von Nutz- und Metadaten auf den verbundenen Servern. Die Datenbank beinhaltet eine Suchmaschine mit der es möglich ist, Metadaten effizient zu durchsuchen. Die Grundlagen zu dieser Architektur wurden im Kapitel 3.3.3 beschrieben. Es ist eine Abwandlung der Architektur, die in XtreemFS verwendet wird.

4.1. Überblick

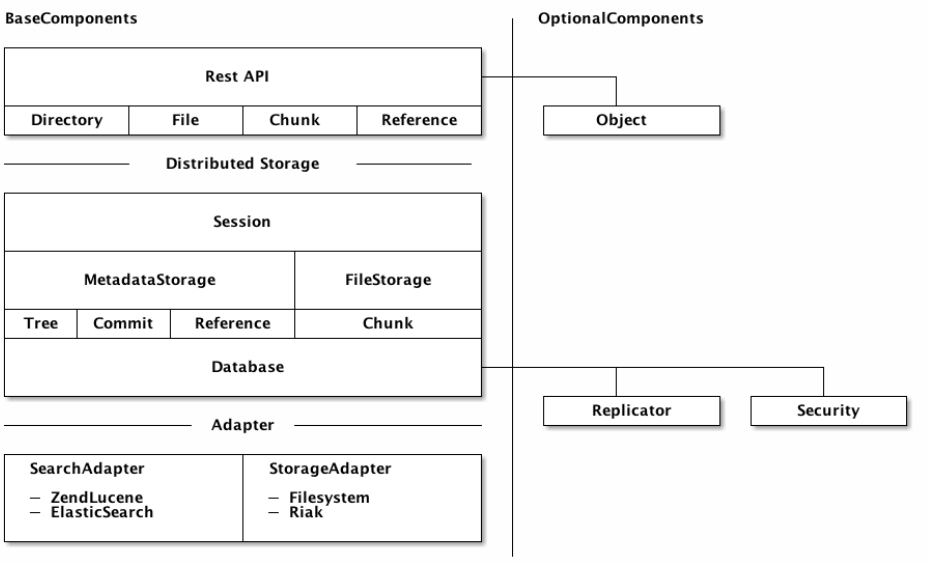

Abbildung 4.1.: Architektur für “symCloud Distributed-Storage”

Die Architektur ist gegliedert in Basiskomponenten und optionale Komponenten. In

Abbildung 4.1 sind die Abhängigkeiten der Komponenten untereinander dargestellt. Die Schichten sind jeweils über ein Interface entkoppelt, um den Austausch einzelner Komponenten zu vereinfachen. Über den "StorageAdapter" bzw. über den "SearchAdapter", lassen sich die Speichermedien der Daten anpassen. Für eine einfache Installation ist es ausreichend, die Daten direkt auf die Festplatte zu schreiben. Es ist allerdings auch denkbar, die Daten in eine verteilte Datenbank wie Riak oder MongoDB zu schreiben, um die Datensicherheit zu erhöhen.

Durch die Implementierung (siehe Kapitel 5) als PHP-Bibliothek ist es möglich, diese Funktionalitäten in jede beliebige Applikation zu integrieren. Durch eine Abstraktion der Benutzerverwaltung ist symCloud vom eigentlichen System komplett entkoppelt.

4.2. Datenmodell

Das Datenmodell wurde speziell für symCloud entwickelt, um die Anforderungen (siehe Kapitel 1.4) zu erfüllen. Dabei wurde großer Wert darauf gelegt, optimale und effiziente Datenhaltung zu gewährleisten. Abgeleitet wurde das Modell (siehe Abbildung 4.2) aus dem Modell, welches dem Versionskontrollsystem GIT (siehe Kapitel 2.4) zugrunde liegt. Dieses Modell unterstützt viele Anforderungen, welche symCloud an seine Daten stellt.

4.2.1. GIT

Das Datenmodell von GIT (genauere Beschreibung in Kapitel 2.4) erfüllt folgende Anforderungen an symCloud:

Versionierung Durch die Commits können Versionshistorien einfach abgebildet und effizient durchsucht werden. Will eine BenutzerIn sehen, wie ihr Dateibaum vor ein paar Wochen ausgesehen hat, kann das System nach einem geeigneten Commit durchsucht werden (anhand der Erstellungszeit) und anstatt des neuesten Commits, diesen Commit für die weiteren Datenbankabfragen verwenden.

Namensräume Mit den Referenzen, aus dem Datenmodell von GIT (siehe Kapitel 2.4), können für jede BenutzerIn mehrere Namensräume geschaffen werden. Jeder dieser Namensräume erhält einen eigenen Dateibaum und kann von mehreren BenutzerInnen verwendet werden. Damit können Shares einfach abgebildet werden.

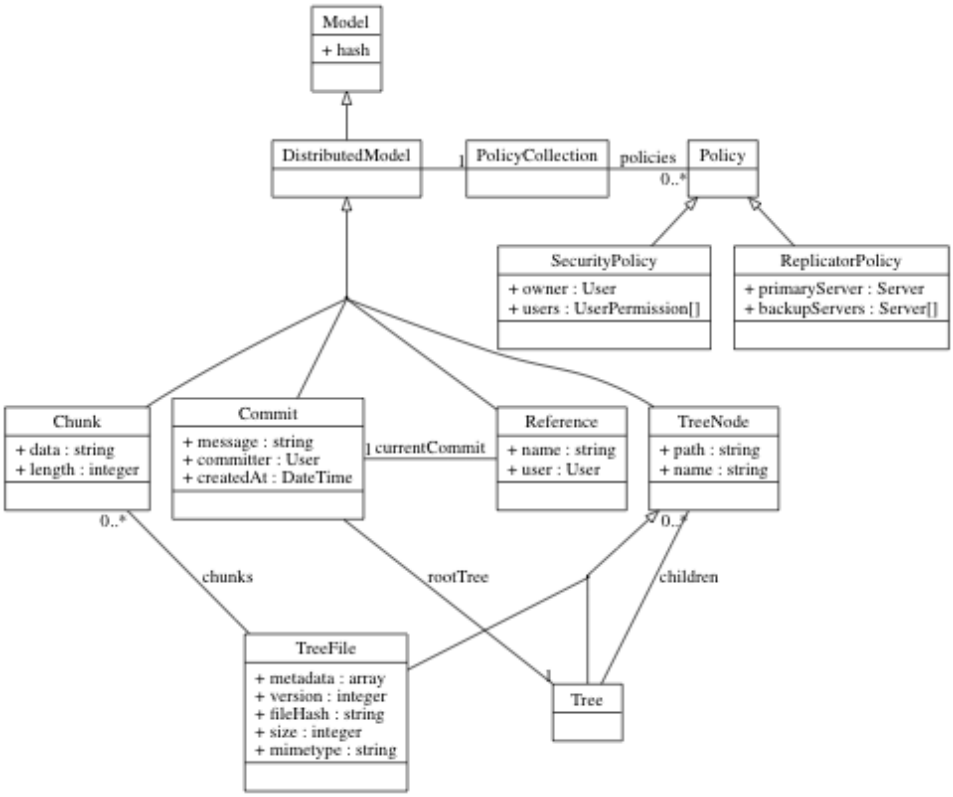

Abbildung 4.2.: Datenmodel für “symCloud-Distributed-Storage”

Jede Referenz kann für jede BenutzerIn eigene Berechtigungen erhalten. Dadurch kann ein Zugriffsberichtigungssystem implementiert werden.

Dieses Datenmodell ist aufgrund seiner Flexibilität eine gute Grundlage für ein verteiltes Dateiverwaltungssystem. Es ist auch in seiner ursprünglichen Form für die Verteilung ausgelegt [Chacon 2009, K. 1.1]. Dies macht es für symCloud interessant es als Grundlage für die Weiterentwicklung zu verwenden. Aufgrund der immutable Objekte können die Operationen Update und Delete komplett vernachlässigt werden, da Daten nicht aus der Datenbank gelöscht werden. Diese Art von Objekten bringt auch große Vorteile mit sich, wenn es um die Zwischenspeicherung (cachen) von Daten geht. Diese können auf allen Servern gecached werden, da diese nicht mehr verändert werden. Eine Einschränkung hierbei sind die Referenzen, die einen veränderbaren Inhalt aufweisen. Diese Einschränkung muss bei der Implementierung des Datenmodells bzw. der Datenbank berücksichtigt werden, wenn die Daten verteilt werden.

4.2.2. symCloud

Für symCloud wurde das Datenmodell von GIT angepasst und erweitert.

Chunks (Blobs) Der Inhalt von Dateien wird nicht an einem Stück in den Speicher geschrieben, sondern er wird in sogenannte Chunks aufgeteilt. Dieses Konzept wurde aus den Systemen GridFS (siehe Kapitel 3.4.1) und XtreemFS (siehe Kapitel 3.3.3) übernommen. Es ermöglicht das Übertragen von einzelnen Dateiteilen, die sich geändert haben[1]. Andere Vorteile für eine Unterteilung der Dateien in Chunks werden im Kapitel 4.5 aufgezählt.

Zugriffsrechte Im Datenmodel von GIT nicht berücksichtigt wurde, die Zuordnung der Referenzen zu einer BenutzerIn. Diese Zuordnung wird von symCloud verwendet, um die Zugriffsrechte zu realisieren. Eine BenutzerIn kann anderen BenutzerInnen die Rechte auf eine Referenz übertragen, auf die sie Zugriff besitzt. Dadurch können Dateien und Strukturen geteilt und zusammen verwendet werden (Shares).

Policies Die Policies werden verwendet, um zusätzliche Informationen zu den Benutzerrechten bzw. Replikationen in einem Objekt zu speichern. Es beinhaltet im Falle der Replikationen den primary Server bzw. eine Liste von Backupservern, auf denen das Objekt gespeichert wurde.

[1] Dieses Feature wurde in der Implementierung, die während dieser Arbeit entstanden ist, nicht umgesetzt.

4.3. Datenbank

Die Datenbank ist eine einfache “Hash-Value” Datenbank, die mithilfe eines Replikators zu einer verteilen Datenbank erweitert wird. Dieses Prinzip wird auch von der Versionsverwaltung GIT als Datenspeicher verwendet (siehe Kapitel 2.4). Die Datenbank serialisiert die Objekte und speichert sie mit Hilfe eines Adapters auf einem bestimmten Speichermedium. Zusätzlich spezifiziert jeder Objekt-Typ, welche Daten als Metadaten in einer Suchmaschine indiziert werden sollen. Dies ermöglicht eine schnelle Suche innerhalb dieser Metadaten, ohne auf das eigentliche Speichermedium zugreifen zu müssen.

SymCloud verwendet einen ähnlichen Mechanismus für die Replikationen, wie in Kapitel 3.3.4 beschrieben wurde. Es implementiert eine einfache Form des primärbasierten Protokolls. Dabei wird jedem Objekt der Server als primary zugewiesen, auf dem es erzeugt wurde. Aus einem Pool an Servern werden die Backupserver ermittelt. Dabei gibt es drei Arten diese zu ermitteln.

Full Die Backupserver werden per Zufallsverfahren ausgewählt. Dabei kann konfiguriert werden, auf wie vielen Servern ein Backup erstellt wird. Dieser Typ wird verwendet um die Chunks gleichmäßig auf die Server zu verteilen. Dadurch lässt sich die Last auf alle Server verteilen. Dies gilt sowohl für den Speicherplatz, als auch die Netzwerkzugriffe. Hierbei könnten auch bessere Verfahren verwendet werden, um den Primary bzw. Backupserver zu ermitteln. Diese Verfahren könnten zum Beispiel auf Basis des freien Speicherplatzes entscheiden, wo das Objekt gespeichert wird[2].

Permissions Wenn ein Objekt auf Basis der Zugriffsrechte verteilt wird, werden alle Server, die mindestens einen registrierten Benutzer mit Zugriff auf dieses Objekt haben, als Backupserver markiert. Dabei gibt es keine Maximalanzahl von Backupservern. Das bedeutet, dass Objekte mit diesem Typ, die nur einen Berechtigten besitzen, nicht verteilt werden und daher vor Verlust oder Zerstörung nicht sicher sind. Dieses Verfahren wird für kleinere Objekte, die zum Beispiel Datei- bzw. Ordnerstrukturen enthalten, verwendet. Es gibt dabei zwei mögliche Zeitpunkte der Verteilung: sofort oder bei Zugriff. Sofort bedeutet, dass das Objekt bei der Erstellung an jeden Server versendet wird. Die zweite Möglichkeit nennt man “Lazy loading” [Wiesmann u. a. 2000], da das Objekt erst dann von einem Server angefragt wird, wenn er dieses benötigt. Der Vorteil dieser Technik ist, dass die Server nicht immer erreichbar

[2]Dieses Feature wurde in der Implementierung, die während dieser Arbeit entstanden ist, nicht umgesetzt.

sein müssen. Allerdings kann es zu Inkonsistenzen kommen, wenn ein Server nicht die neuesten Daten verwendet, bevor er Änderungen durchführt. Wichtig ist bei diesem Verfahren, dass Änderungen der Zugriffsrechte automatisch zu einem neuen Objekt führen, damit die Backupserver diese Änderungen mitbekommen. Um die Datensicherheit für diese Objekte zu erhöhen, könnten aus dem Serverpool eine konfigurierbare Anzahl von Backupservern, wie bei dem Full Replikationstypen, ausgewählt werden. Allerdings müsste der Pool auf die zugriffsberechtigten Server beschränkt werden. Diese Methode wurde nicht vollständig implementiert, da der Prototyp keine Autorisierung für Objekte vorsieht. Allerdings werden Objekte, die nicht in der lokalen Datenbank vorhanden sind, nachgeladen [Wiesmann u. a. 2000]. Objekte dieses Typs könnten theoretisch gelöscht werden, wenn alle berechtigten BenutzerInnen gelöscht worden wären. Dies ist allerdings aufgrund der verteilten Architektur schwer zu erkennen und daher im implementierten Prototypen nicht umgesetzt.

Stubs Dieser Typ ist eigentlich kein Replikationsmechanismus, aber er ist wesentlicher Bestandteil des Verteilungsprotokolls von symCloud. Objekte, die mit diesem Typ verteilt werden, werden als sogenannte Stubs an alle bekannten Server verteilt. Dabei fungiert dieses als eine Art remote Objekt. Es besitzt keine Daten und darf nicht gecached werden. Bei jedem Zugriff erfolgt eine Anfrage an den primary Server, der die Daten zurückliefert, wenn die Zugriffsrechte zu dem Objekt gegeben sind. An dieser Stelle lassen sich Lock-Mechanismen implementieren, da diese Objekte immer nur auf dem primary Server geändert werden können. Falls es an dieser Stelle zu einem Konflikt kommt, betrifft es nur den einen Backupserver und nicht das komplette Netzwerk. Stubs können, wie auch der vorherige Typ, automatisch verteilt werden oder “Lazy” bei der ersten Verwendung nachgeladen werden. In der Implementierung, wurde dieser Typ nicht vorgesehen. Es wurde allerdings eine Methode implementiert, die es ermöglicht, Objekte im Netzwerk zu suchen und sie nachzuladen.

Im Kapitel 5.1 werden diese Vorgänge anhand von Ablaufdiagrammen genauer erläutert.

4.4. Metadatastorage

Der Metadatastorage verwaltet die Struktur der Dateien. Es beinhaltet folgende Punkte:

Dateibaum (Tree) Diese Objekte beschreiben wie die Dateien zusammenhängen. Diese Struktur ist vergleichbar mit einem Dateibaum auf einem lokalen Dateisystem. Es gibt pro Namensraum jeweils ein Root-Verzeichnis, welches andere Verzeichnisse und Dateien enthalten kann. Dadurch lassen sich beliebig tiefe Strukturen abbilden. In diesem Baum können zu einer Datei auch andere Werte, wie zum Beispiel Titel, Beschreibung und Vorschaubilder hinterlegt werden.

Versionen (Commit) Über die zusammenhängenden Commits kann der Dateiänderungsverlauf abgebildet werden. Jede Änderung im Baum bewirkt das Erstellen eines neuen Commits auf Basis des Vorherigen. Dabei wird der aktuelle Baum in die Datenbank geschrieben und ein neuer Commit mit einer Referenz auf das Root-Verzeichnis erstellt.

Referenzen Um den aktuellen Commit und damit den aktuellen Dateibaum der BenutzerIn nicht zu verlieren, werden Referenzen immer auf den neuesten Commit gesetzt. Dies erfordert das Aufbrechen des Konzepts der immutable Objekte. Um diese Objekt-Typen zu unterstützen werden, diese Objekte auf keinem Server gecached und die Backupserver erhalten automatische Updates zu Änderungen.

Diese drei Objekt-Typen werden im Netzwerk mit unterschiedlichen Replikationstypen verteilt. Die Strukturdaten (Tree und Commit) verwenden den Typ “Permission”. Das bedeutet, dass jeder Server, der Zugriff auf den Dateibaum besitzt, das Objekt in seine Datenbank ablegen kann. Im Gegensatz dazu werden Referenzen als Stub-Objekte im Netzwerk verteilt. Diese werden bei jedem Zugriff auf dessen primary Server angefragt. Änderungen an einer Referenz werden ebenfalls an den primary Server weitergeleitet.

4.5. Filestorage

Der Filestorage verwaltet die abstrakten Dateien im System. Diese Dateien werden als reine Datencontainer angesehen und besitzen daher keinen Namen oder Pfad. Eine Datei besteht nur aus Datenblöcken (Chunks), einer Länge, dem Mimetype und einem Hash für die Identifizierung. Diese abstrakten Dateien werden in den Tree des Metadatastorage eingebettet und stehen daher nur konkreten Dateien zur Verfügung. Das bedeutet, dass eine konkrete Datei, eine Liste von Chunks enthält, die die eigentlichen Daten repräsentieren. Die Trennung von Daten und Metadaten macht es möglich zu erkennen, wenn eine Datei an verschiedenen Stellen des Systems, vorkommt und dadurch wiederverwendet werden

kann. Theoretisch können auch Teile einer Datei in einer anderen Datei vorkommen. Dies ist aber je nach Größe der Chunks sehr unwahrscheinlich. Chunks besitzen keine Zugriffsrechte, daher spielt es keine Rolle, ob dieser von demselben oder von einer anderen BenutzerIn wiederverwendet wird. Wenn der Hash übereinstimmt, besitzen beide Dateien der BenutzerInnen denselben Datenblock und dürfen diesen verwenden.

Für symCloud bietet das Chunking von Dateien zwei große Vorteile:

Wiederverwendung Durch das Aufteilen von Dateien in Daten-Blöcke ist es theoretisch möglich, dass sich mehrere Dateien denselben Chunk teilen. Häufiger jedoch geschieht dies, wenn Dateien von einer Version zur nächsten nur leicht verändert werden. Nehmen wir an, dass eine große Text-Datei im Storage liegt, die die Größe eines Chunks übersteigt und nun weiterer Inhalt angehängt wird. Wird nun eine neue Version erstellt, besteht diese aus dem Chunk der ersten Version und aus einem neuen. Dadurch konnte sich das Storagesystem den Speicherplatz eines Chunks sparen. Mithilfe bestimmter Algorithmen könnte die Ersparnis optimiert werden[3] (siehe Kapitel 6.2) [Anglin 2011].

Streaming Um auch große Dateien zu verarbeiten, bietet das Chunking von Dateien die Möglichkeit, Daten immer nur Block für Block zu verarbeiten. Dabei können die Daten so verarbeitet werden, dass immer nur wenige Chunks im Speicher gehalten werden müssen. Zum Beispiel kann beim Streaming von Videodateien immer nur ein Chunk versendet und sofort wieder aus dem Speicher gelöscht werden, bevor der nächste Chunk aus der Datenbank geladen wird. Dies verkürzt die Zeit, um eine Antwort zu erzeugen. Moderne Video-Player machen sich dieses Verfahren zu Nutze und versenden viele HTTP-Request mit bestimmten Header-Werten, um den Response zu beschränken. Dabei wird der Request-Header “range” auf den Ausschnitt der Datei gelegt, die der Player gerade für die Ausgabe benötigt. Aus diesen Informationen kann das System die benötigten Chunks berechnen und genau diese aus dem Storage laden [Fielding 2014].

Im Filestorage werden zwei Arten von Objekten beschrieben. Zum einen sind dies die abstrakten Dateien, die nicht direkt in die Datenbank geschrieben werden, sondern primär der Kommunikation dienen und in den Dateibaum eingebettet werden können. Zum anderen sind es die konkreten Chunks, die direkt in die Datenbank geschrieben werden. Um die Chunks optimal zu verteilen, werden diese mit dem Replikationstyp “Full”

[3] Dieses Feature wurde in der Implementierung, die während dieser Arbeit entstanden ist, nicht umgesetzt.

persistiert. Dabei werden diese Objekte auf eine festgelegte Anzahl von Servern verteilt. Dadurch lässt sich der gesamte Speicherplatz des Netzwerkes, mit dem Hinzufügen neuer Server erweitern und ist nicht auf den Speicherplatz des kleinsten Servers beschränkt. Die Chunk Objekte werden dann auf den Remoteservern in einem Cache gehalten, um den Traffic zwischen den Servern so minimal wie möglich zu halten. Dieser Cache kann diese Objekte unbegrenzt lange speichern, da diese Blöcke unveränderbar sind und nicht gelöscht werden können. Dateien werden nicht wirklich gelöscht, sondern nur aus dem Dateibaum entfernt. Alte Versionen der Datei können auch später wiederhergestellt werden, indem die Commit-Historie zurückverfolgt wird.

4.6. Session

Als zentrale Schnittstelle auf die Daten fungiert die "Session". Sie ist als eine Art "High-Level-Interface" konzipiert und ermöglicht den Zugriff auf alle Teile des Systems über eine zentrale Schnittstelle. Zum Beispiel können Dateien hoch- bzw. heruntergeladen oder die Metadaten mittels Dateipfad abgefragt werden. Damit fungiert es als Zwischenschicht zwischen "Filestorage", "Metdatastorage" und Rest-API.

4.7. Rest-API

Die Rest-API ist als zentrale Schnittstelle nach außen gedacht. Sie wird zum Beispiel verwendet, um Daten für die Oberfläche in der Plattform zu laden oder Dateien mit einem Endgerät zu synchronisieren. Diese Rest-API ist über ein Benutzersystem gesichert. Die Zugriffsrechte können sowohl über Form-Login und Cookies, für Javascript Applikationen, als auch über OAuth2 für externe Applikationen überprüft werden. Dies ermöglicht eine einfache Integration in andere Applikationen, wie es zum Beispiel in der Prototypen-Implementierung (siehe Kapitel 5.2) passiert ist. Die OAuth2 Schnittstelle ermöglicht es auch, externe Applikationen mit Daten aus symCloud zu versorgen.

Die Rest-API ist in vier Bereiche aufgeteilt:

Directory Diese Schnittstelle bietet den Zugriff auf die Ordnerstruktur einer Referenz über den vollen Pfad: `/directory/<reference-name>/<directory>`. Bei einem GET-Request auf diese Schnittstelle, wird der angeforderte Ordner als JSON-

Objekt zurückgeliefert. Enthalten ist dabei unter anderem der Inhalt des Ordners (Dateien oder andere Ordner).

File Unter dem Pfad `/file/<reference-name>/<directory>/<filename>.<extension>` können Dateien heruntergeladen oder ihre Informationen abgefragt werden. Über Post-, Put- und Delete-Requests können Dateien erstellt, aktualisiert und gelöscht werden.

Reference Die Schnittstelle für die Referenzen erlaubt das Erstellen und Abfragen von Referenzen. Um mehrere Dateien gleichzeitig zu aktualisieren, ermöglicht die Referenz-API einen PATCH-Request mit einer Liste von Operationen. Diese Operationen werden auf dem Tree des neuesten Commits ausgeführt, ein neuer Commit angelegt und die Referenz aktualisiert.

Objekts Diese Objektschnittstelle verwendet der Replikator, um die Objekte zwischen den Servern zu verteilen. Dabei werden die HTTP-Befehle GET und POST verwendet, um Daten abzufragen oder zu erstellen.

Die genaue Funktion der Rest-API wird im Kapitel 5.2 beschrieben.

4.8. Zusammenfassung

Das Konzept von symCloud baut sehr stark auf der Verteilung der Daten innerhalb eines Netzwerkes auf. Dies ermöglicht eine effiziente und sichere Datenverwaltung. Allerdings kann die Software auch ohne dieses Netzwerk ihr volles Potenzial entfalten. Es erfüllt die in Kapitel 1.4 angeführten Anforderungen und bietet durch die erweiterbare Architektur die Möglichkeit andere Systeme und Plattformen zu verbinden. Über die verschiedenen Replikationstypen lassen sich verschiedene Objekte auf verschiedenste Weise im Netzwerk verteilen. Die einzelnen Server sind durch eine definierte Rest-API verbunden und daher unabhängig von der darunterliegenden Technologie.

Dieses Konzept vereint viele der im vorherigen Kapitel beschriebenen Vorzüge der beschriebenen Technologien.

5. Implementierung

In diesem Kapitel werden die einzelnen Komponenten, die für symCloud entwickelt wurden, genauer betrachtet. Es entstand während der Entwicklungsphase ein einfacher Prototyp, mit dem die Funktionsweise des im vorherigen Kapitel beschriebenen Konzeptes, gezeigt werden konnte.

Dabei sind drei Komponenten entstanden:

Bibliothek (distributed-storage) Die Bibliothek ist der Kern der Applikation und implementiert große Teile des Konzeptes von symCloud. Sie baut auf modernen Web-Technologien auf und verwendet einige Komponenten des PHP-Frameworks Symfony2[1]. Dieses Framework ist eines der beliebtesten Frameworks in der Open-Source Community von PHP.

Plattform (symCloud) Die Plattform bietet neben der Rest-API auch eine einfache Benutzungsschnittstelle an, mit der es möglich ist im Browser seine Dateien zu verwalten. Als Basis verwendet symCloud die Content-Management-Plattform SULU[2] der Vorarlberger Firma MASSIVE ART WebServices GmbH[3] aus Dornbirn. Diese Plattform bietet eine erweiterbare Admin-Benutzungsschnittstelle, eine Benutzerverwaltung und ein Rechtesystem. Diese Features ermöglichen symCloud eine schnelle Entwicklung der Oberfläche und deren zugrundeliegende Dienste.

Synchronisierungsprogramm (jibe) Das Synchronisierungsprogramm ist ein Konsolen-Tool, mit dem es möglich ist, Dateien aus einem lokalen Ordner mit dem Server zu synchronisieren. Es dient als Beispiel für die Verwendung der API mit einer externen Applikation.

Der Source-Code dieser drei Komponenten ist auf der beiliegenden CD (`/source`) oder auf Github https://github.com/symcloud zu finden.

[1] http://symfony.com/
[2] http://www.sulu.io
[3] http://www.massiveart.com/de

5.1. Distributed-Storage

Der Distributed-Storage ist der Kern der Anwendung und kann als Bibliothek in jede beliebige PHP-Anwendung integriert werden. Die Anwendung stellt die Authentifizierung und die Rest-API zur Verfügung, um mit den Kern-Komponenten zu kommunizieren.

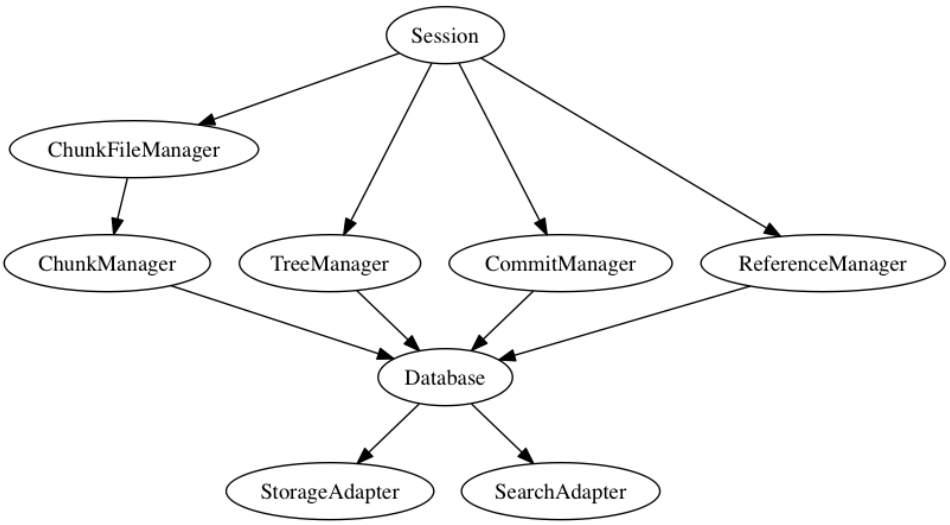

Abbildung 5.1.: Schichten von "Distributed Storage"

Der interne Aufbau der Bibliothek ist in vier Schichten (siehe Abbildung 5.1) aufgeteilt.

Session Zentrale Schnittstelle, die alle Manager vereint und einen gemeinsamen Zugriffspunkt bildet, um mit dem Storage zu kommunizieren.

Manager Verwalten den Lifecycle der Domain-Objekte. Der jeweilige Manager erstellt und verwaltet die Objekte eines bestimmten Typs (zum Beispiel `CommitMananger` verwaltet Commit-Objekte). Die Clients der Manager sind jeweils gegen die Interfaces der Objekte programmiert (siehe Manager Pattern[4]). Die Instanzen des Typs sind reine Daten-Container. Die Manager-Schicht ermöglicht eine spezifische Schnittstelle für jeden Objekt-Typ und eine gezielte Kommunikation mit der Datenbank.

Database Die Datenbank benutzt Mechanismen von PHP, um die Objekte zu serialisieren und zu speichern. Dabei kann über Metadaten festgelegt werden, welche Eigenschaften serialisiert bzw. welche Eigenschaften in der Suchmaschine indexiert werden. Beim Laden der Daten aus der Datenbank, können mit Hilfe dieser Metadaten die Objekte wieder deserialisiert werden.

[4] http://wiki.hsr.ch/APF/TheManagerPattern

Adapter Die Adapter dienen dazu, das Speichermedium bzw. die Suchmaschine zu abstrahieren. Durch die Implementierung eines Interfaces, kann jede beliebige Speichertechnologie bzw. Suchmaschine verwendet werden.

Die Datenbank ist durch den Einsatz von Events flexibel erweiterbar. Mit Hilfe dieser Events kann zum Beispiel die Replikator-Komponente folgende Abläufe realisieren.

Verteilung Bei einem "store" Event verteilt der Replikator das Objekt auf die ermittelten Backupserver. Um die Einstellungen des Replikators zu persistieren fügt der Eventhandler eine "ReplicatorPolicy" an das Modell an. Diese "Policy" wird zusätzlich mit dem Modell persistiert.

Nachladen Im Falle eines "fetch" Events, werden fehlende Daten von den bekannten Servern nachgeladen. Dieses Event wird sogar dann geworfen, wenn die Daten im lokalen "StorageAdapter" nicht vorhanden sind. Dies erkennt der Replikator und fragt bei allen bekannten Servern an, ob sie dieses Objekt kennen. Dies gilt für die Replikationstyps "Permission" und "Full". Über einen ähnlichen Mechanismus kann der Replikationstyp "stub" realisiert werden. Der einzige Unterschied ist, dass die Backupserver den primary Server kennen und nicht alle bekannten Server durchsuchen müssen.

5.1.1. Objekte speichern

Der Mittelpunkt des Speicher-Prozesses (siehe Abbildung 5.2) ist die Serialisierung zu Beginn. Hierfür werden die Metadaten des Objekts anhand seiner Klasse aus dem "MetadataManager" geladen und anhand dieser Informationen serialisiert. Diese Daten werden mit dem "EventDispatcher" aus dem Symfony2 Framework in einem Event zugänglich gemacht. Die Eventhandler haben die Möglichkeit die Daten zu bearbeiten und "Policies" zu dem Modell zu erstellen. Abschließend werden die Daten zuerst mithilfe des "StorageAdapter" persistiert und durch den "SearchAdapter" in den Suchmaschinenindex aufgenommen. Um verschiedene Objekttypen voneinander zu trennen und eigene Namensräume zu schaffen, definieren die Metadaten der Klasse einen eindeutigen Kontext. Dieser Kontext wird den Adaptern übergeben, um Kollisionen zwischen den Datensätzen zu verhindern.

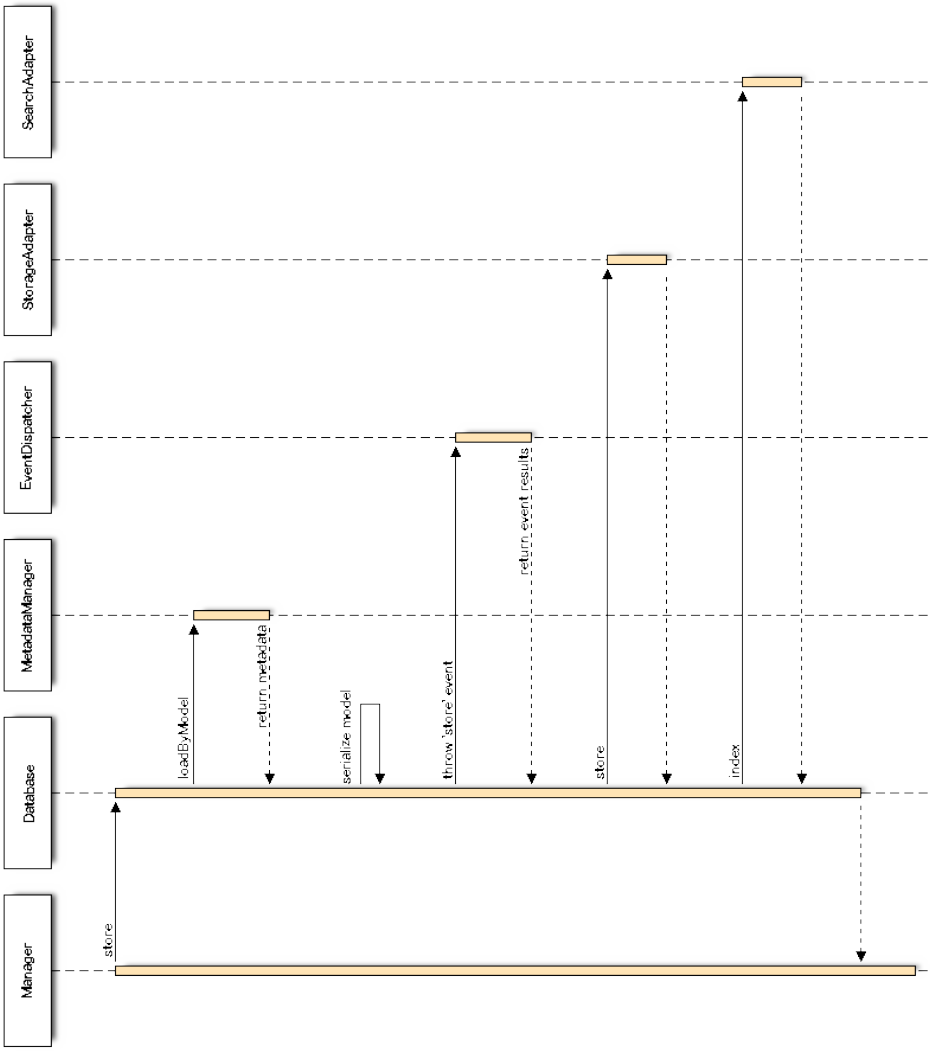

Abbildung 5.2.: Objekte speichern

5.1.2. Objekte abrufen

Wie zu erwarten war, ist der Abruf-Prozess (siehe Abbildung 5.3) von Daten ein Spiegelbild des Speicherprozesses. Zuerst wird versucht, mit dem Kontext des Objektes die Daten aus dem “Storage” zu laden. Diese Daten werden durch den “EventDispatcher” dem Eventhandler zur Verfügung gestellt. Diese haben die Möglichkeit, zum Beispiel fehlende Daten nachzuladen, Änderungen an der Struktur der Daten durchzuführen oder den Prozess abzubrechen, wenn keine Rechte vorhanden sind dieses Objekt zu lesen. Diese veränderten Daten werden abschließend für den Deserialisierungsprozess herangezogen.

Die beiden Abläufe, um Objekte zu speichern und abzurufen, beschreiben eine lokale Datenbank, die die Möglichkeit bietet, über Events die Daten zu verändern oder anderweitig zu verwenden. Sie ist unabhängig vom Datenmodell von symCloud und könnte für alle möglichen Objekte verwendet werden. Daher ist symCloud auch für künftige Anforderungen gerüstet.

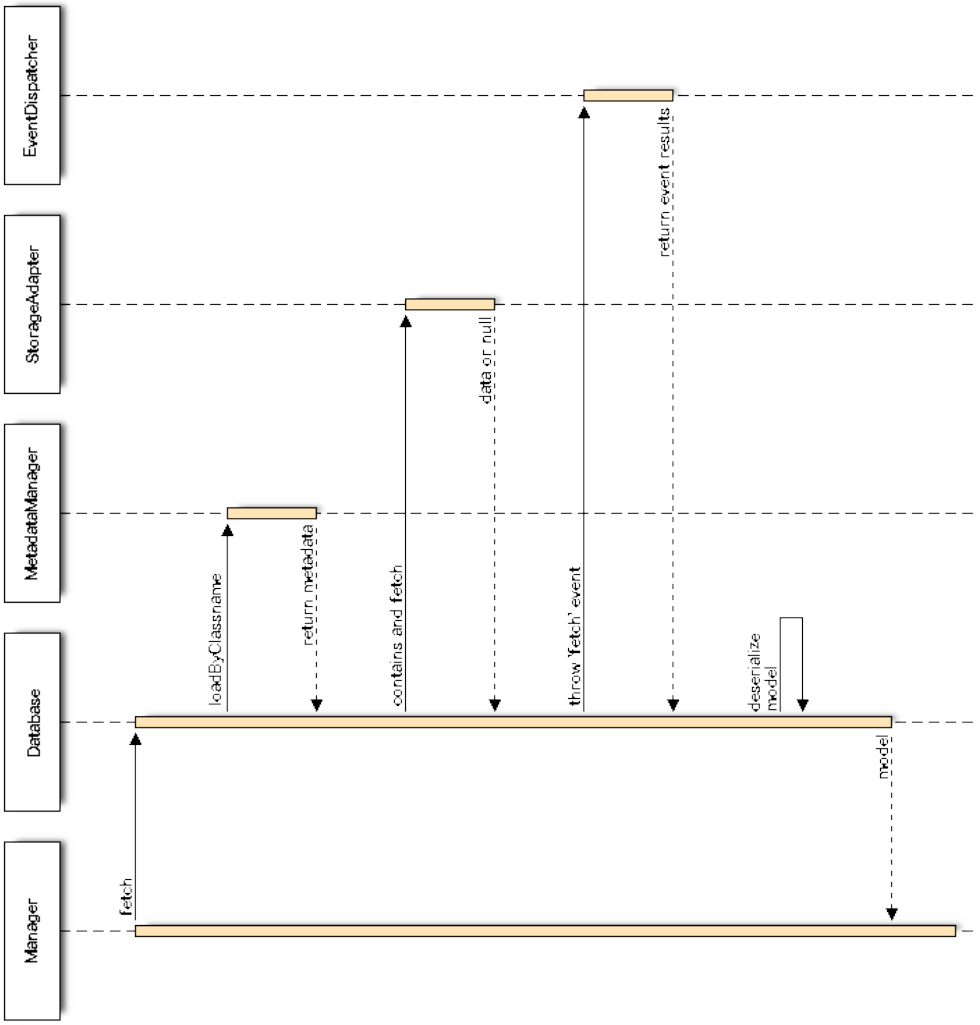

Abbildung 5.3.: Objekte abrufen

5.1.3. Replikator

Der Replikator verwendet Events, um die Prozesse des Ladens und Speicherns von Daten zu beeinflussen und damit die verteilten Aspekte für die Datenbank umzusetzen. Dabei implementiert der Replikator eine einfache Version des primärbasierten Protokolls. Für diesen Zweck wird der Replikator mit einer Liste von verfügbaren Servern initialisiert. Auf Basis dieser Liste werden die Backupserver für jedes Objekte ermittelt.

Wie schon im Kapitel 4.3 erwähnt, gibt es verschiedene Arten, die Backupserver für ein Objekt zu ermitteln. Implementiert wurde neben dem Typ “Full” auch ein automatisches “Lazy”-Nachladen für fehlende Objekte. Dieses Nachladen ist ein wesentlicher Bestandteil der beiden anderen Typs (“Permission” und “Stub”).

Full

Bei einem “store” Event werden die Backupserver per Zufall aus der Liste der vorhandenen Server ausgewählt und der Server, der das Objekt erstellt, als primary Server markiert. Anhand der Backupserver-Liste werden die Daten an die Server verteilt. Dazu werden der Reihe nach die Daten an die Server versendet und auf eine Bestätigung gewartet. Falls einer dieser Server nicht erreichbar ist, wird dieser ausgelassen und ein anderer Server als Backup herangezogen. Damit wird der konsistente Zustand der Datenbank verifiziert. Abschließend wird die erstellte “Policy” zu den Daten hinzugefügt, damit sie mit ihnen persistiert werden und später wiederverwendet werden können. Dieser Prozess wird in der Abbildung 5.4 visualisiert.

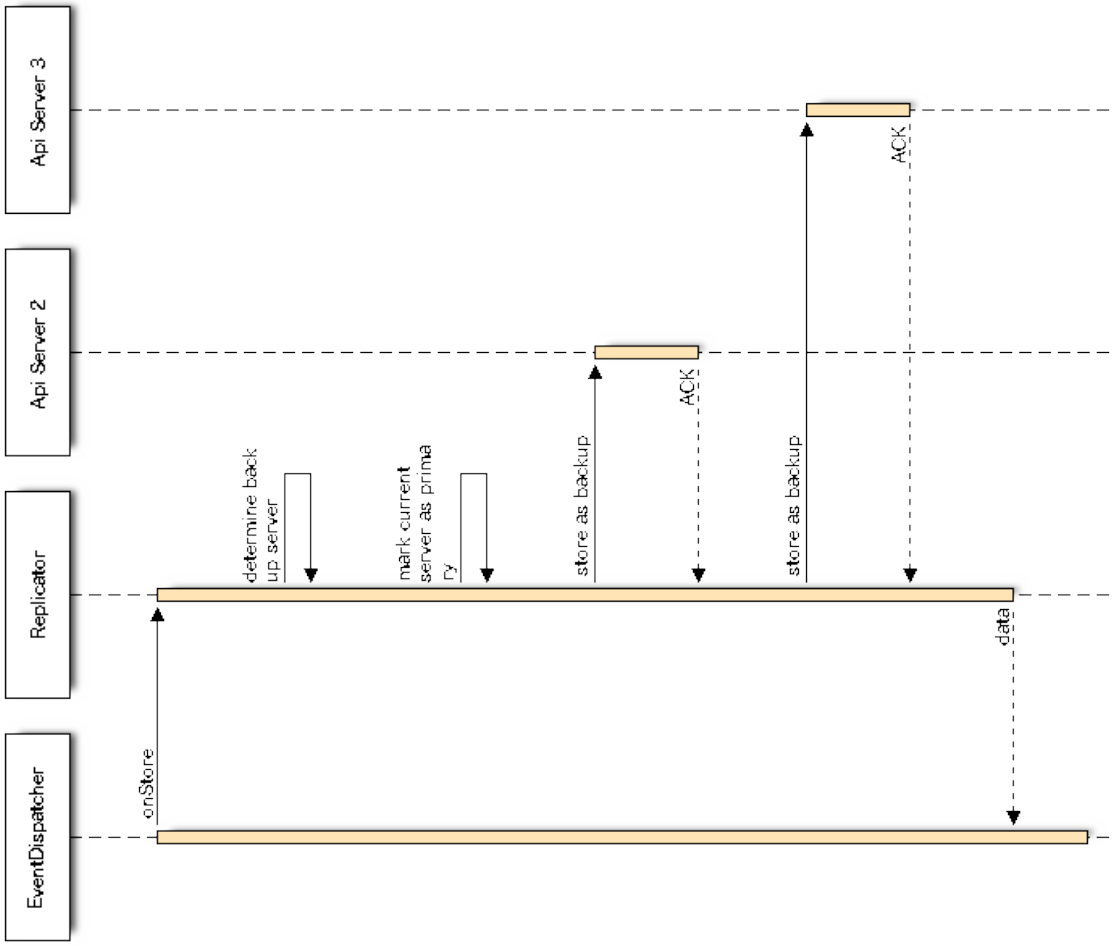

Abbildung 5.4.: Replikationtyp “Full”

Lazy

Um fehlende Daten im lokalen Speicher nachzuladen, werden der Reihe nach alle bekannten Server abgefragt. Dabei gibt es vier mögliche Antworten (siehe Abbildung 5.5), auf die der Replikator reagieren kann. Der Status kann anhand des HTTP-Response-Codes erkannt werden.

404 Das Objekt ist auf dem angefragten Server nicht bekannt.

302 Das Objekt ist bekannt, aber der angefragte Server ist nur als Backupserver markiert. Dieser Server kennt allerdings die genaue Adresse des primary Servers und leitet auf diesen weiter.

403 Das Objekt ist bekannt und der angefragte Server als primary Server für dieses Objekt markiert. Der Server überprüft die Zugangsberechtigungen, weil diese aber nicht gegeben sind, wird der Zugriff verweigert. Der Replikator erkennt, dass der Benutzer nicht berechtigt ist, die Daten zu lesen und verweigert den Zugriff.

200 Wie bei 403 ist der angefragte Server, der primary Server des Objektes. Allerdings ist die BenutzerIn berechtigt das Objekt zu lesen und der Server gibt direkt die Daten zurück. Diese Daten dürfen auch gecached werden. Die Berechtigungen für andere Benutzer werden direkt mitgeliefert, um später diesen Prozess nicht noch einmal ausführen zu müssen.

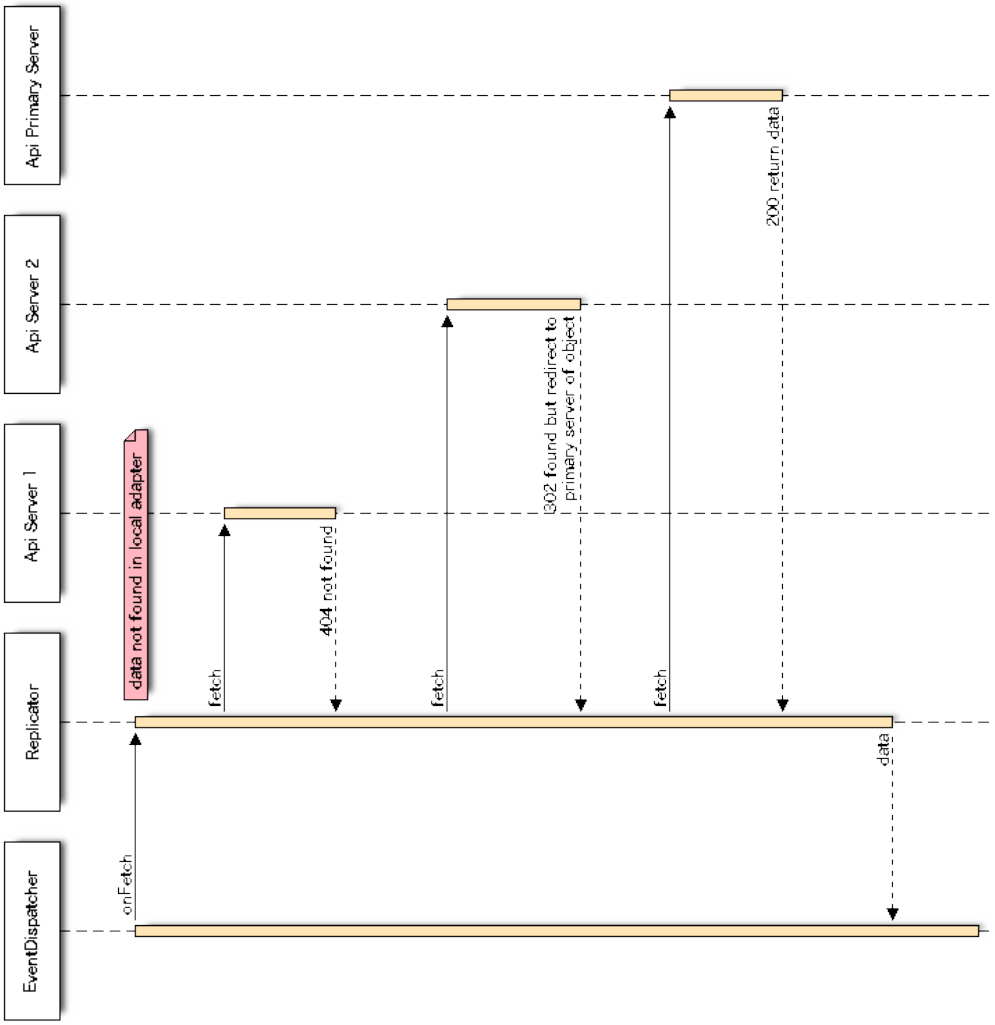

Abbildung 5.5.: Replikator "Lazy"-Nachladen

Mithilfe dieses einfachen Mechanismuses kann der Replikator Daten von anderen Servern nachladen, ohne zu wissen, wo sich die Daten befinden. Dieser Prozess bringt allerdings Probleme mit sich. Zum Beispiel muss jeder Server angefragt werden, bevor der Replikator endgültig sagen kann, dass das Objekt nicht existiert. Bei einem sehr großen Netzwerk kann dieser Prozess sehr lange dauern. Aufgrund des Datenmodells sollte dieser Fall allerdings nur selten vorkommen, da Daten nicht gelöscht werden und daher keine Deadlinks entstehen können.

5.1.4. Adapter

Für die Abstrahierung des Speichermediums verwendet die Datenbank das Adapter-Pattern. Mithilfe dieses Patterns, kann jede symCloud-Installation sein eigenes Speichermedium verwenden. Dabei gibt es zwei Arten von Adaptern:

Storage Der "StorageAdapter" wird verwendet, um serialisierte Objekte lokal zu speichern oder zu laden. Er implementiert im Grunde einen einfachen Befehlssatz: `store`, `fetch`, `contains` und `delete`. Jeder dieser Befehle erhält - neben anderen Parametern - einen Hash und einen Kontext. Der Hash ist sozusagen der Index des Objektes. Der Kontext wird verwendet, um Namensräume für die Hashes zu schaffen. Dies implementiert der Dateisystemadapter, indem er für jeden Kontext einen Ordner erstellt und für jeden Hash eine Datei. So kann schnell auf ein einzelnes Objekt zugegriffen werden.

Beispiel:
Der Befehl aus Listing 5.1 erzeugt aus dem übergebenen Hash-Key den Pfad `ab/cd/12/34.symcloud.dat`. In dieser Datei werden die serialisierten Daten abgelegt. Bei einer Anfrage der Daten mittels Hash-Key, können die Daten direkt aus der Datei gelesen und retourniert werden. Dieser Mechanismus wird von der Cache-Bibliothek von Doctrine[5] wiederverwendet.

Listing 5.1: Storage-Adapter store

```
1 $storageAdapter->store('abcd1234', array('name' => 'Storage-Example'),
      'example');
```

[5] http://doctrine-orm.readthedocs.org/en/latest/reference/caching.html

Search Der "SearchAdapter" wird verwendet, um die Metadaten zu den Objekten zu indexieren. Dies wird benötigt, wenn die Daten durchsucht werden. Jeder "SearchAdapter" implementiert folgende Befehle: `search`, `index` und `deindex`. Wobei auch hier mit Hash und Kontext gearbeitet wird. Über den Suchbefehl, können alle oder bestimmte Kontexte durchsucht werden. Für die Entwicklung des Prototyps wurde die Bibliothek Zend-Search-Lucene[6] verwendet, da diese ohne weitere Abhängigkeiten verwendet werden kann.

Bei der Verwendung des Replikators gibt es einen zusätzlichen Adapter, der mithilfe der Server-Informationen mit dem Remoteserver kommunizieren kann. Dieser API-Adapter implementiert den Befehlssatz: `fetch` und `store`. Diese beiden Methoden werden verwendet, um Remote-Objekte abzufragen oder zu erstellen.

Die Adapter sind Klassen, die die Komplexität des Speichermediums bzw. der API von der restlichen Applikation trennen, um dadurch die Bibliothek unabhängig von der Applikation implementieren zu können.

5.1.5. Manager

Die Manager sind die Schnittstelle, um mit den einzelnen Schichten des Datenmodells zu kommunizieren. Jeder dieser Manager implementiert ein Interface mit dem es möglich ist, mit den jeweiligen Datenobjekten zu interagieren. Grundsätzlich sind dies Befehle, um ein Objekt zu erstellen oder abzufragen. Im Falle des "ReferenceManager" oder "TreeManager" bietet sie auch die Möglichkeit, Objekte zu bearbeiten. Der "ReferenceManager" bearbeitet dabei auch wirklich ein Objekt in der Datenbank, indem er es einfach überschreibt. Diese Operation ist, durch den Replikationstyp Stub, auch in einem verteilten Netzwerk möglich. Der TreeManager klont das Objekt und erstellt unter einem neuen Hash ein neues Objekt, sobald es mit einem Commit zusammen persistiert wird.

5.1.6. Kurzfassung

Die Bibliothek Distributed-Storage bietet eine einfache und effiziente Implementierung des in Kapitel 4 beschriebenen Konzeptes. Es baut auf eine erweiterbare Hash-Value Datenbank auf. Diese Datenbank wird mittels eines Eventhandlers (Replikator) zu einer verteilten Datenbank. Dabei ist es für die Datenbank irrelevant, welcher Transportlayer

[6] http://framework.zend.com/manual/1.12/de/zend.search.lucene.html

oder welches Protokoll verwendet wird. Dieser kann neben HTTP, jeden beliebigen anderen Transportlayer verwenden. Der konsistente Zustand der Datenbank kann mittels Bestätigungen bei der Erstellung, blockierenden Vorgängen und nicht löschbaren Objekten garantiert werden. Nicht veränderbare Objekte lassen sich dauerhaft und ohne Updates verteilen. Alle anderen Objekte können so markiert werden, dass sie immer bei einem primary Server angefragt werden müssen und nur für die Datensicherheit an die Backupserver verteilt werden.

5.2. Plattform

Die Plattform bzw. die Anwendung stellt die Rest-API und die Authentifizierung zur Verfügung. Dies ermöglicht der Bibliothek die Kommunikation mit anderen Servern und Applikationen. Zusätzlich beinhaltet sie die Oberfläche, um mit den Daten in einem Browser zu interagieren.

5.2.1. Authentifizierung

Die Authentifizierung und die Benutzerverwaltung stellt die Plattform SULU zur Verfügung. Hierfür wird der "UserProvider" von SULU dem "Distributed-Storage" bekannt gemacht. Allerdings stellt die Plattform nur eine Authentifizierung mittels HTML-Formular (Benutzername und Passwort) oder HTTP-Basic standardmäßig zur Verfügung. Um die Verwendung der API auch für Dritt-Entwickler Applikationen zu ermöglichen, wurde das Protokoll OAuth2 in SULU integriert. Eine genauere Beschreibung dieses Protokolls wird im Anhang B gegeben.

Eine Autorisierung zwischen den Servern ist momentan nicht vorgesehen. Dies wurde in der ersten Implementierungsphase nicht umgesetzt, wäre aber für eine produktiven Einsatz unerlässlich.

5.2.2. Rest-API

Die Rest-API ist, wie schon im Kapitel 4.7 beschrieben, in vier verschiedene Schnittstellen aufgeteilt. Dabei werden die SULU-internen Komponenten verwendet, um die Daten für die Übertragung zu serialisieren und RESTful[7] aufzubereiten. Für eine verteilte

[7] http://restcookbook.com/

Installation implementiert die Plattform den “ApiAdapter”, um die Rest-API für die Bibliothek zu abstrahieren.

5.2.3. Benutzungsschnittstelle

Die Architektur der Benutzungsschnittstelle von SULU ist als “Single-Page-Application” ausgeführt. In dieser Architektur ist die Oberfläche der Website aus individuellen Komponenten zusammengesetzt, die unabhängig aktualisiert und ersetzt werden können [Mesbah; Deursen 2007]. Das bedeutet, dass die Oberfläche aus nur einem klassischen Request aufgebaut wird. In diesem ist die Grundstruktur definiert und die grundlegenden JavaScript Dateien eingebunden. Diese Scripts laden dann alle anderen JavaScript Dateien nach, die die Oberfläche Stück für Stück zusammensetzen.

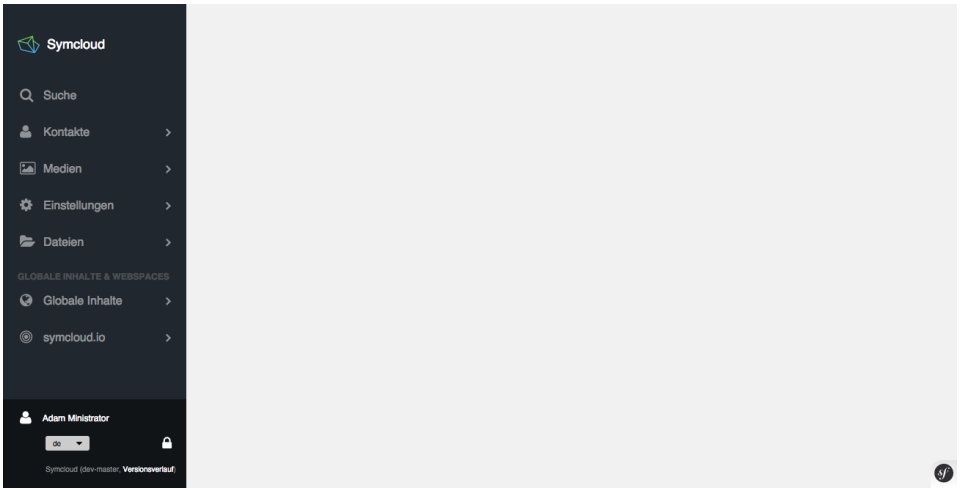

Abbildung 5.6.: Grundlegender Aufbau des SULU-Admin

In der Abbildung 5.6 ist der grundlegende Aufbau des SULU-UI zu erkennen. Im rechten Bereich ist eine erweiterbare Navigation, die bereits den symCloud Punkt “Dateien” enthält, links ist der sogenannte “Content”-Bereich. Dieser Bereich kann von den nachgeladenen Komponenten gefüllt werden. Um das UI einheitlich zu gestalten, bietet SULU vordefinierte Komponenten an, die zum Beispiel eine Liste abstrahieren. Dieser Listen-Komponente wird im Grunde eine URL übergeben, unter welcher die Daten heruntergeladen werden können. Die Liste generiert daraufhin eine Tabelle mit den Daten aus dem Response der angegebenen URL (siehe Abbildung 5.7).

Über der Liste befindet sich eine Toolbar, mit der es möglich ist, neue Dateien zu erstellen. Über die beiden anderen Schaltflächen lassen sich die Reihenfolge und Sichtbarkeit der

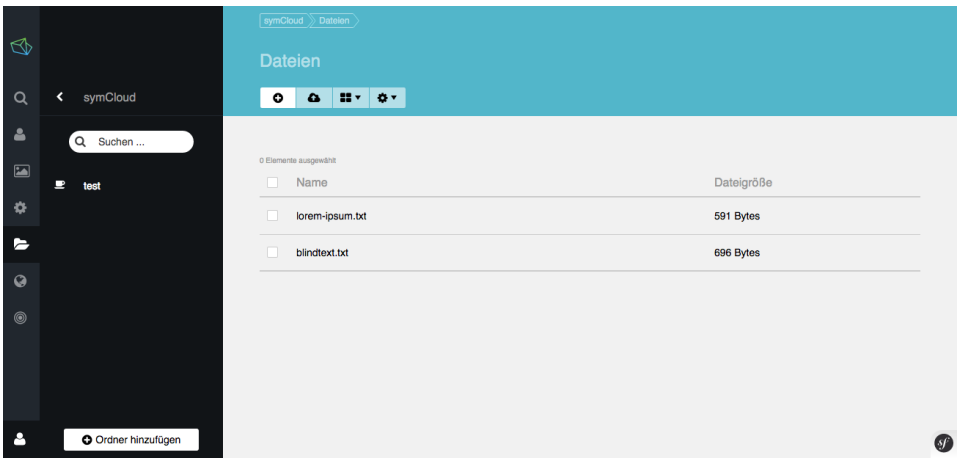

Abbildung 5.7.: Dateiliste von symCloud

Spalten umschalten. Um Dateien zu löschen oder zu bearbeiten, erscheinen neben dem Namen zwei Schaltflächen, sobald die Maus über den Namen bewegt wird (siehe Abbildung 5.8).

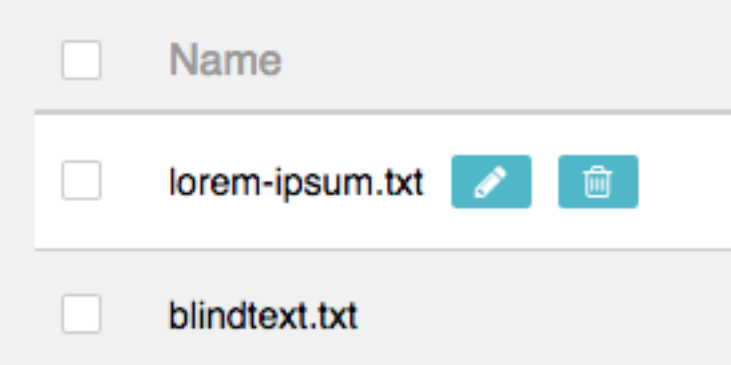

Abbildung 5.8.: Schaltflächen um eine Datei zu bearbeiten oder zu löschen

Das Formular für neue Dateien ist einfach gestaltet (siehe Abbildung 5.9). Es bietet zwei Formularfelder, mit denen der Name und Inhalt der Datei bearbeitet werden kann. Mit demselben Formular können Dateien auch bearbeitet werden.

5.2.4. Kurzfassung

Die Plattform ist ein reiner Prototyp der zeigen soll, ob das Konzept (aus dem Kapitel 4) funktionieren kann. Es bietet in den Grundzügen alle Funktionen an, um zu einem

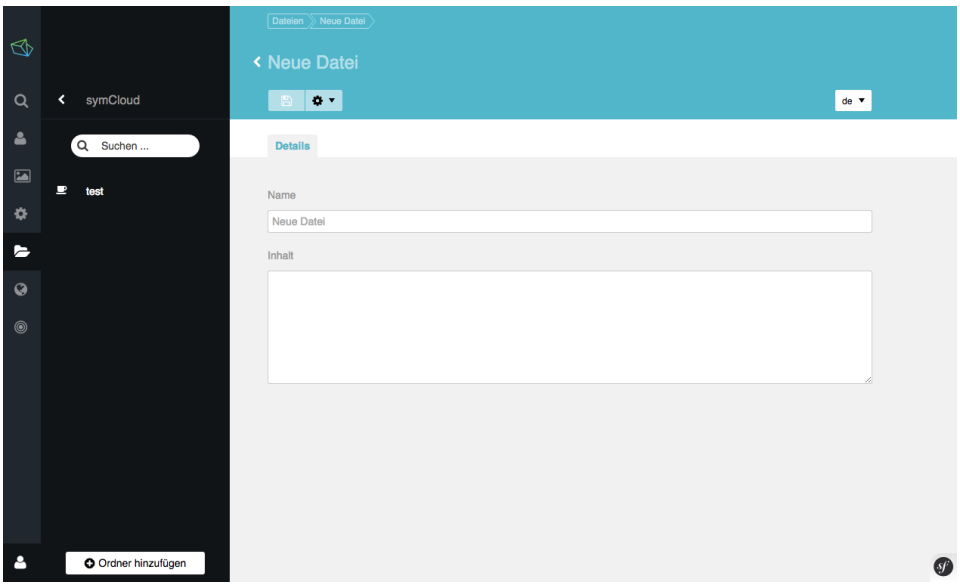

Abbildung 5.9.: Formular um eine neue Datei zu erstellen

späteren Zeitpunkt[8] diesen Prototyp zu einer vollständigen Plattform heranwachsen zu lassen.

5.3. Synchronisierungsprogramm: Jibe

Jibe ist das Synchronisierungsprogramm zu einer symCloud-Installation. Es ist ein einfaches PHP-Konsolen Tool mit dem es möglich ist, Daten aus einer symCloud-Installation mit einem Endgerät zu synchronisieren. Das Programm wurde mithilfe der Symfony Konsole-Komponente[9] umgesetzt. Diese Komponente ermöglicht eine schnelle und unkomplizierte Entwicklung solcher Konsolen-Programme.

Ausgeliefert wird das Programm in einem sogenannten PHAR-Container[10]. Dieser Container enthält alle benötigten Source-Code- und Konfigurationsdateien. Das Format ist vergleichbar mit dem JAVA-Container JAR. PHAR-Container werden in der PHP-Community oft verwendet, um komplexe Applikationen, wie zum Beispiel PHPUnit[11] (ein Test Framework für PHP) auszuliefern.

[8] Dieses Feature wurde in der Implementierung, die während dieser Arbeit entstanden ist, nicht umgesetzt.
[9] http://symfony.com/doc/current/components/console/introduction.html
[10] http://php.net/manual/de/intro.phar.php
[11] https://phpunit.de/

Über den ersten Parameter kann spezifiziert werden, welches Kommando ausgeführt werden soll. Alle weiteren Parameter sind Argumente für das angegebene Kommando. Über den Befehl `php jibe.phar sync` kann der Synchronisierungsvorgang gestartet werden.

Listing 5.2: Ausführen des 'configure' Befehls

```
1 $ php jibe.phar configure
2 Server base URL: http://symcloud.lo
3 Client-ID: 9_1442hepr9cpw8wg8s0o40s8gc084wo8ogso8wogowookw8k0sg
4 Client-Secret: 4xvv8pn29zgoccos0c4g4sokw0ok0sgkgkso04408k0ckosk0c
5 Username: admin
6 Password:
```

Im Listing 5.2 ist die Ausführung des "Konfigurieren"-Kommandos dargestellt. Argumente können sowohl an den Befehl angehängt oder durch den Befehl abgefragt werden. Eine Validierung, zum Beispiel der URL, kann direkt in einem Kommando implementiert werden.

Diese Kommandos stehen dem Benutzer zur Verfügung:

configure Konfiguriert den Zugang zu einer symCloud-Installation. Falls notwendig, koordiniert sich das Tool mit der Installation, um andere Informationen zu Repliken oder verbundenen Installationen zu erhalten.

refresh-token Aktualisiert das Access-Token von OAuth2. Dies ist notwendig, da die Access-Tokens über einen Ablaufzeitpunkt verfügen. Ist auch das Refresh-Token abgelaufen, muss der Befehl "configure" erneut ausgeführt werden.

status Gibt den aktuellen Status des Access-Token auf der Konsole aus. Dieses Kommando wird standardmäßig aufgerufen, wenn kein anderes Kommando angegeben wurde.

sync Startet den Synchronisierungsvorgang. Über die Option `-m` kann eine Nachricht zu dem erstellten Commit angefügt werden.

5.3.1. Architektur

Der zentrale Bestandteil von Jibe ist eine "CommandQueue" (siehe Abbildung 5.10). Sie sammelt alle nötigen Kommandos ein und führt sie nacheinander aus. Diese "Queue" ist

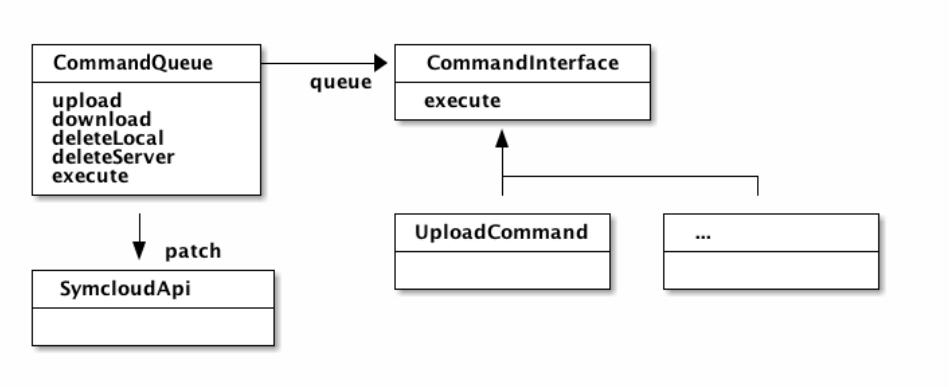

Abbildung 5.10.: Architektur von Jibe

nach dem "Command-Pattern" entworfen. Folgende Befehle können dadurch aufgerufen werden:

Upload Eine neue Datei auf den Server hochladen.

Download Die angegebene Datei wird vom Server heruntergeladen und lokal gespeichert.

DeleteServer Die Datei wird auf dem Server gelöscht.

DeleteLocal Die Lokale Datei wird gelöscht.

Aus diesen vier Kommandos lässt sich nun ein kompletter Synchronisierungsvorgang abbilden.

5.3.2. Abläufe

Für einen kompletten Synchronisierungsvorgang werden folgende Informationen benötigt:

Lokale Hashwerte Diese werden aus den aktuellen Dateibeständen generiert.

Zustand der Dateibestände Der Zustand nach der letzten Synchronisierung. Wenn diese Hashwerte mit den aktuellen Hashwerten verglichen werden, kann zuverlässig ermittelt werden, welche Dateien sich geändert haben. Zusätzlich kann die Ausgangsversion der Änderung erfasst werden, um Konflikte zu erkennen.

Aktueller Serverzustand enthält die aktuellen Hashwerte und Versionen aller Dateien, die auf dem Server bekannt sind. Diese werden verwendet, um zu erkennen, dass sich Dateien auf dem Server verändert haben bzw. gelöscht wurden.

Diese drei Informationspakete kann man sehr einfach ermitteln. Einzig und alleine der Zustand der Dateien muss nach einer Synchronisierung beim Client gespeichert werden, um diese beim nächsten Vorgang wiederzuverwenden.

Die Tabelle 5.1 gibt Aufschluss über die Erkennung von Kommandos aus diesen Informationen.

Tabelle 5.1.: Evaluierung der Zustände

	Hash	Version	Description	Download	Upload	Del L	Del S	Conflict
1	(Z = X) = Y	n = m	Nothing to be done					
2	(Z = X) != Y	n < m	Server file changed, download new version	x				
3	(Z != X) != Y	n = m	Client file change, upload new version		x			
4	(Z != X) != Y	n < m	Client and Server file changed, conflict					x
5	(Z = X) = Y	n < m	Server file changed but content is the same					
6	X		New client file, upload it		x			
7	Y	m	New server file, download it	x				
8	X	n	Server file deleted, remove client version			x		
9	Y	n = m	Client file deleted, remove server version				x	

Tabelle 5.2.: Legende zu Tabelle 5.1

Zeichen	Beschreibung
X	Lokaler Hashwert der Datei
Z	Lokaler Hashwert der Datei bei der letzten Synchronisierung
Y	Remote Hashwert der Datei bei der letzten Synchronisierung
n	Lokale Version der Datei
m	Remote Version der Datei
Del	Delete - Löschen
L	Local - Lokal
S	Server

Nicht angeführte Werte in der Tabelle, sind zu dem Zeitpunkt nicht verfügbar bzw. werden nicht benötigt, was zum Beispiel bedeutet, dass wenn der lokale Hash nicht angeführt ist, die Datei nicht vorhanden ist (gelöscht oder noch nicht angelegt).

Beispiel der Auswertungen anhand des Falles Nummer vier (aus der Tabelle 5.1):

1. Lokale Datei hat sich geändert: Alter Hashwert unterscheidet sich zu dem aktuellen Wert.
2. Serverversion ist größer als lokale Version.
3. Aktueller und Server-Hashwert stimmen nicht überein.

Das bedeutet, dass sich sowohl die Serverdatei als auch die lokale Kopie geändert haben. Dadurch entsteht ein Konflikt, der aufgelöst werden muss. Das Auflösen solcher Konflikte ist nicht Teil dieser Arbeit, er wird allerdings in Kapitel 6.5 kurz behandelt.

5.3.3. Kurzfassung

Der Synchronisierungsclient ist ein Beispiel dafür, wie die Rest-API von anderen Applikationen verwendet werden kann, um die Daten aus symCloud zu verwenden. Es wären viele andere Anwendungsfälle denkbar.

In diesem Beispiel wurde auch die Komplexität des Synchronisierungsprozesses durchleuchtet und eine Lösung geschaffen, um schnell und effizient einen Ordner mit symCloud zu synchronisieren.

5.4. Zusammenfassung

Die Prototypenimplementierung umfasst die wichtigsten Punkte des im vorherigen Kapitel verfassten Konzeptes. Es umfasst neben dem Datenmodell und einer Datenbank, die in der Lage ist die Daten über eine Menge von Servern zu verteilen, auch eine einfache Plattform, mit der man Dateien in einer einfachen Weboberfläche bearbeiten kann. Um die Dateien mit einem Endgerät zu synchronisieren, wurde der Client Jibe implementiert, der über ein einfache Rest-API in der Lage ist, die Dateien zu synchronisieren.

Vorgesehen, aber nicht implementiert, wurden die Bereiche:

- Zugriffskontrolle
- Replikationen auf Basis von Benutzerberechtigungen

Diese Bereiche können als Eventhandler implementiert werden. Alle dafür benötigten Komponenten sind vorhanden. Die Zugriffskontrollliste könnte als Policy zu jedem Objekt gespeichert werden. Auf Basis dieser Policy könnte sowohl der Replikator entscheiden, welche Server Zugriff auf das Objekt besitzen, als auch ein Eventhandler entscheiden, ob das Objekt für den Benutzer, der das Objekt angefragt hat, verfügbar ist.

6. Ergebnisse und Ausblick

Ziel dieser Arbeit war es, ein Konzept für eine verteilte Speicherverwaltung aufzustellen. Aus diesem entstand ein einfacher Prototyp, mit dem die Umsetzbarkeit dieses Konzeptes bewiesen wurde.

Im ersten Teil der Arbeit wurde, neben der Motivation und der Projektbeschreibung, eine Liste der Anforderungen an das Konzept und das Projekt aufgestellt. Diese Anforderungen umfassten die Punkte Datensicherheit, Funktionalitäten und Architektur. Auf diese Anforderungen wurden in den darauf folgenden Kapiteln eingegangen und Lösungsansätze erarbeitet. Das vorliegende Konzept erfüllt alle Anforderungen und ist durch seine Flexibilität auf andere Plattformen portierbar.

Verschiedene Applikationen wurden auf die Erfüllbarkeit der Anforderungen untersucht. Dabei wurden die Themen “verteilte Systeme”, “Cloud-Datenhaltung”, “verteilte Daten” und “verteilte Datenmodelle” jeweils anhand von Beispielen analysiert.

Um eine solide Grundlage für das Konzept zu erarbeiten, wurden im Evaluierungskapitel verschiedene Möglichkeiten der Datenhaltung in verteilten Systemen analysiert und auf die Tauglichkeit als Basis für das Konzept überprüft. Es wurde keine passende Technologie gefunden, jedoch konnten diverse Aspekte der evaluierten Technologien im Konzept verwendet und umgesetzt werden. Ein Beispiel dafür ist die Dateireplikation aus XtreemFS und das dort verwendete primärbasierte Protokoll. Eine vereinfachte Version dieses Protokolls wurde im Konzept eingebaut und später im Prototyp umgesetzt.

In der Konzeptionsphase wurde aus den Vorteilen der analysierten Anwendungen, Technologien und Dienste, auf Basis eines verteilten Datenmodells, das Konzept einer verteilten Anwendung für die Dateiverwaltung erstellt. Als Grundlage für dieses Datenmodell wurde die verteilte Versionsverwaltung GIT herangezogen. GIT ermöglicht eine sichere und effiziente Verteilung der Daten. Für die Datenspeicherung wurde eine eigene Datenbank konzipiert, die es ermöglicht, Objekte anhand eines Hashwertes effizient in verschiedene Speichermedien abzulegen. Um diese Hashwerte schnell zu finden, werden die Metadaten

dieser Objekte in einer Suchmaschine indexiert. Dadurch kann die Suche über die Daten effizient ausgeführt werden.

Die Implementierung des Prototypen wurde in drei Abschnitte untergliedert. Im ersten Teil wurde eine Bibliothek entwickelt, die das Datenmodell, die Datenbank und eine Zugriffsschicht implementiert. Diese Bibliothek ist unabhängig von der restlichen Anwendung und kann in alle PHP-Applikationen eingebunden werden, die mit dem Netzwerk von symCloud kommunizieren wollen. Der zweite Teil umfasst die Plattform, die als funktionierender Prototyp in die bestehende Plattform SULU eingebunden wurde. Neben der Rest-API bietet die Plattform auch eine einfache Benutzungsoberfläche, mit der Änderungen an den Dateien möglich sind. Über die Authentifizierungsschicht können SULU-BenutzerInnen Dateien in symCloud ablegen und verwalten. Der dritte Teil ist als Beispiel für eine Dritt-Hersteller Applikation konzipiert. Dabei handelt es sich um eine Synchronisierungssoftware, die es ermöglicht, Dateien von einem Rechner mit symCloud zu synchronisieren.

Auch wenn der entwickelte Prototyp nicht alle Facetten des Konzepts umsetzt, ist er ein Beweis für die Funktionstüchtigkeit dieses Konzepts. Einige Punkte wurden im Konzept (siehe 1.4.4) allerdings nicht betrachtet. Einer dieser Punkte ist die Performance des Systems, welcher in der aktuellen Implementierung die größten Herausforderungen darstellt. Hier könnten weiterführende Analysen und Entwicklungen gerade in der Verteilung der Replikationen erhebliche Fortschritte bringen. Zum Abschluss dieser Arbeit soll noch ein kurzer Ausblick gegeben werden.

6.1. Performance von Replikationen

Die Performance der Replikationen ist stark von der Performance des Übertragungsmediums abhängig. Der Prototyp implementiert einen Prozess, der den eigentlichen Prozess blockiert und dadurch die Antwortzeit an den Client stark beeinflusst.

Um genau diese Verzögerungen zu vermeiden, implementiert das verteilte Konfigurationsmanagement ZooKeeper[1] das Protokoll Zab. Dieses basiert auf Broadcast-Nachrichten, um die Änderungen in einem Netzwerk zu verteilen. Dabei können alle Replikationen abstimmen, ob die Änderung durchführbar ist. Dieses Votum und die Änderungen auf allen Replikationen laufen parallel ab. Bei Zookeeper kümmert sich ein eigener Prozess

[1] https://zookeeper.apache.org/

um diesen Broadcast, dies vermindert die Antwortzeit an die Clients und erhöht den Durchsatz des Systems [Reed; Junqueira 2008].

6.2. Rsync Algorithmus

Algorithmen wie Rsync sind darauf ausgelegt, die Effizienz der Datenhaltung und Übertragung zu erhöhen. Wenn zwei Dateien auf verschiedenen, jedoch über ein Netzwerk verbundenen Rechnern, synchronisiert werden sollen, ermittelt der Rsync-Algorithmus jene Teile der Datei, die auf beiden Rechnern identisch sind. Diese Teile müssen sich nicht an derselben Stelle in den Dateien befinden. Bei diesem Algorithmus erstellt einer der beiden Rechner eine sogenannte Signatur. Dies geschieht, indem die Datei in Blöcke einer bestimmten Länge unterteilt werden. Von diesen Blöcken werden die Prüfsummen ermittelt. Diese Summen werden in eine Datei geschrieben und an den zweiten Rechner gesendet. Dieser unterteilt nun seine Datei ebenfalls in Blöcke derselben Größe. Allerdings nicht nur der Reihe nach, sondern von jeder beliebigen Stelle der Datei aus, um möglichst viele Treffer zu erzielen. Als Ergebnis sendet der zweite Rechner eine Liste von Operationen an seinen Partner. Mithilfe dieser kann der erste Rechner die Datei "nachbauen". Dieser Algorithmus erhöht nicht nur die Übertragungsgeschwindigkeit von Dateiänderungen, sondern auch die Speichernutzung von Version zu Version. Es müssen nur die Änderungen gespeichert werden, selbst dann, wenn Änderungen in der Mitte der Datei durchgeführt wurden [Tridgell; Mackerras 1996].

6.3. Lock-Mechanismen

Es gibt diverse Lock-Mechanismen, die auf einem Server optimal funktionieren. Allerdings ist es ungleich schwerer diese Mechanismen über ein Netzwerk zu verteilen. Das Team von XtreemFS entwickelte den sogenannten "Flease"-Algorithmus. Dieser Algorithmus ist ein dezentraler und fehlertolerante Koordination von "lease" also Objekt-Locks in verteilten Systemen. Der Algorithmus arbeitet ohne zentrale Schnittstelle und gewährleistet einen exklusiven Zugriff auf eine Ressource in einem skalierbaren Umfeld [B. Kolbeck 2010].

6.4. Protokolle

Um die Kommunikation zwischen den Servern zu vereinheitlichen, gibt es einige offene Protokolle, die als Erweiterung zum Konzept von symCloud geeignet wären. Zwei dieser Protokolle sind:

Webfinger Webfinger ist ein Protokoll, um Informationen über Objekte zu übertragen, die über eine URL identifiziert werden. Als Medium für die Übertragung wird das Standard HTTP-Protokoll verwendet. Die Antwort auf eine Anfrage ist ein JSON-Objekt mit dem Mimetyp "application/jrd+json". Dieses JRD (JSON Ressource Descriptor) enthält alle relevanten Informationen zu einem Objekt und weiterführende Links [Jones 2013]. Dieses Protokoll wird von Diaspora verwendet und könnte daher als Bindeglied zwischen den beiden Applikationen dienen.

PubSubHubbub Andere PubSub-Protokolle verlangen von den Clients, dass sie in regelmäßigen Abständen neue Beiträge abfragen. Dies führt zu einer Zeitdifferenz zwischen Erstellen und Anzeigen. Außerdem fällt, falls es keine neuen Beiträge gibt, viel Overhead an. PubSubHubbub integriert daher einen sogenannten Hub zwischen Server und Client. Die Feed-Server pingen bei einem neuen Beitrag die Hubs an, diese holen sich den Beitrag und leiten ihn via Push an die Clients weiter. Die Aufgabe, die Nachrichten sicher an die Clients zu versenden, übernimmt der Hub [Fitzpatrick 2014]. In symCloud könnte ein solcher Hub das Bindeglied zwischen primary- und Backupserver sein. Dies würde die Performance des Erstellens und Verteilens eines Objektes enorm beschleunigen. Jedoch müssten sich die Hubs darum kümmern, die Änderungen und neue Objekte sicher an die Backupserver weiterzuleiten.

Neben der Interoperabilität zwischen verschiedenen Applikationen (Webfinger) bietet gerade das Protokoll PubSubHubbub eine enorme Steigerung der Datensicherheit und Performance.

6.5. Konfliktbehandlung

Sowohl die Erkennung von Konflikten als auch die sinnvolle Lösung des Konflikts ist eine wichtige Aufgabe. Unterschieden werden drei Stufen der Konfliktbehandlung [Bleiholder 2004]:

Konflikte werden ignoriert Es werden weder Unsicherheiten beseitigt, noch Widersprüche aufgelöst. Dabei gilt oft, dass die letzte Änderung übernommen wird, ohne die Änderungen davor zu betrachten.

Konflikte werden vermieden Die Unsicherheiten werden beseitigt, Widersprüche können nicht gelöst werden, jedoch durch die geschickte Auswahl von Werten umgangen. Konflikte werden Teilweise zusammengeführt, wenn die Daten es zulassen.

Konflikte werden gelöst Alle Unsicherheiten können beseitigt und die Widersprüche sinnvoll aufgelöst werden. Die Daten werden vollständig zusammengeführt.

Um die beiden Versionen zusammenzuführen, können verschiedene Operatoren (wie JOIN, UNION oder MERGE) verwendet werden. Diese Operatoren führen aber nur Objekte sinnvoll zusammen, wenn die Änderungen jeweils andere Eigenschaften betreffen [Bleiholder 2004].

Den Inhalt einer Datei zusammenzuführen ist ungleich schwieriger. Um dies zu bewerkstelligen, verwenden Anwendungen wie Dropbox oder ownCloud einen einfachen Mechanismus, bei dem beide Versionen nebeneinander erstellt werden. Dies geschieht mit einem Zusatz im Dateinamen. Somit werden die Konflikte nicht automatisch gelöst [Dropbox 2015b].

A. Amazon S3 System-spezifische Metadaten

Tabelle A.1.: Objekt Metadaten [Amazon-Web-Services 2015c]

Name	Description
Date	Object creation date.
Content-Length	Object size in bytes.
Last-Modified	Date the object was last modified.
Content-MD5	The base64-encoded 128-bit MD5 digest of the object.
x-amz-server-side-encryption	Indicates whether server-side encryption is enabled for the object, and whether that encryption is from the AWS Key Management Service (SSE-KMS) or from AWS-Managed Encryption (SSE-S3).
x-amz-version-id	Object version. When you enable versioning on a bucket, Amazon S3 assigns a version number to objects added to the bucket.
x-amz-delete-marker	In a bucket that has versioning enabled, this Boolean marker indicates whether the object is a delete marker.
x-amz-storage-class	Storage class used for storing the object.
x-amz-website-redirect-location	Redirects requests for the associated object to another object in the same bucket or an external URL.
x-amz-server-side-encryption-	If the x-amz-server-side-encryption is present and has the value of aws:kms, this indicates

Name	Description
aws-kms-key-id	the ID of the Key Management Service (KMS) master encryption key that was used for the object.
x-amz-server-side-encryption-customer-algorithm	Indicates whether server-side encryption with customer-provided encryption keys (SSE-C) is enabled.

B. Exkurs: OAuth2

Für die Authentifizierung wurde das Protokoll OAuth in der Version 2 implementiert. Dieses offene Protokoll erlaubt eine standardisierte, sichere API-Autorisierung für Desktop, Web und Mobile-Applikationen. Initiiert wurde das Projekt von Blaine Cook und Chris Messina [Hammer 2010].

Die BenutzerIn kann einer Applikation den Zugriff auf seine Daten autorisieren, die von einer anderen Applikation zur Verfügung gestellt wird. Dabei werden nicht alle Details seiner Zugangsdaten preisgegeben. Typischerweise wird die Weitergabe eines Passwortes an Dritte vermieden [Hammer 2010].

B.1. Begriffe

In OAuth2 werden folgende vier Rollen definiert:

Resource owner BesitzerIn einer Ressource, die er für eine Applikation bereitstellen will [Dick 2012, S. 5].

Resource server Der Server, der die geschützten Ressourcen verwaltet. Er ist in der Lage Anfragen zu akzeptieren und die geschützten Ressourcen zurückzugeben, wenn ein geeignetes und valides Token bereitgestellt wurde [Dick 2012, S. 5].

Client Die Applikation stellt Anfragen im Namen des Ressourceneigentümers an den "resource server". Sie holt sich vorher die Genehmigung von einer berechtigten BenutzerIn [Dick 2012, S. 5].

Authorization server Der Server, der die Zugriffs-Tokens nach der erfolgreichen Authentifizierung des Ressourceneigentümers bereitstellt [Dick 2012, S. 5].

Neben diesen Rollen spezifiziert OAuth2 folgende Begriffe:

Access-Token Die Access-Tokens fungieren als Zugangsdaten zu geschützten Ressourcen. Es besteht aus einer Zeichenkette, die als Autorisierung für einen bestimmten Client

ausgestellt wurde. Sie repräsentieren die “Scopes” und die Dauer der Zugangsberechtigung, die durch die BenutzerIn bestätigt wurde [Dick 2012, S. 9].

Refresh-Token Diese Tokens werden verwendet, um neue Access-Tokens zu generieren, wenn das alte Access-Token abgelaufen ist. Wenn der Autorisierungsserver diese Funktionalität zur Verfügung stellt, liefert er es mit dem Access-Token aus. Der Refresh-Token besitzt eine längere Lebensdauer und berechtigt nicht den Zugang zu den anderen API-Schnittstellen [Dick 2012, S. 9].

Scopes Mithilfe von Scopes, lassen sich Access-Token für bestimmte Bereiche der API beschränken. Dies kann sowohl auf Clientebene als auch auf Access-Token Ebene spezifiziert werden [Dick 2012, S. 22].

Die Interaktion zwischen Ressourcenserver und Autorisierungsserver ist nicht spezifiziert. Diese beiden Server können in der selben Applikation betrieben werden, aber auch eine verteilte Infrastruktur wäre möglich. Dabei würden die beiden auf verschiedenen Servern betrieben werden. Der Autorisierungsserver könnte in einer verteilten Infrastruktur Tokens für mehrere Ressourcenserver bereitstellen [Dick 2012, S. 5].

B.2. Protokoll Ablauf

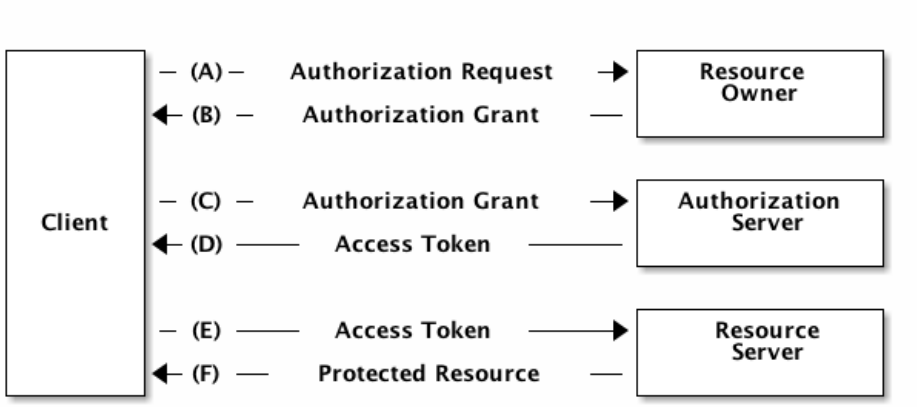

Abbildung B.1.: Ablaufdiagramm des OAuth [Dick 2012, S. 7]

Der Ablauf einer Autorisierung [Dick 2012, S. 7ff] mittels Oauth2, der in der Abbildung B.1 abgebildet ist, enthält folgende Schritte:

A) Der Client fordert die Genehmigung des Ressourcenbesitzers. Diese Anfrage kann direkt an die BenutzerIn gestellt werden (wie in der Abbildung dargestellt) oder

vorzugsweise indirekt über den Autorisierungsserver (wie zum Beispiel bei Facebook).

B) Der Client erhält einen "authorization grant". Er repräsentiert die Genehmigung des Ressourcenbesitzers, die geschützten Ressourcen zu verwenden.

C) Der Client fordert einen Token beim Autorisierungsserver mit dem "authorization grant" an.

D) Der Autorisierungsserver authentifiziert den Client, validiert den "authorization grant" und gibt einen Token zurück.

E) Der Client fordert eine geschützte Ressource und autorisiert die Anfrage mit dem Token.

F) Der Ressourcenserver validiert den Token und gibt die Ressource zurück.

B.3. Zusammenfassung

OAuth2 wird verwendet, um es externen Applikationen zu ermöglichen, auf die Dateien der BenutzerIn zuzugreifen. Das Synchronisierungsprogramm Jibe verwendet dieses Protokoll, um die Autorisierung zu erhalten, die Dateien der BenutzerInnen zu verwalten.

C. Installation

Dieses Kapitel enthält eine kurze Anleitung wie symCloud (inklusive JIBE) installiert und konfiguriert werden kann. Es umfasst eine einfache Methode auf einem System und ein verteiltes Setup mit beliebig vielen verbundenen Installationen.

C.1. Systemanforderungen

Folgende Anforderungen werden an den Server und die Endgeräte gestellt, auf denen symCloud verwendet wird:

- Betriebssystem: Mac OSX oder Linux
- Webserver: apache oder nginx[1] mit aktiviertem "URL rewriting"
- PHP: 5.5 oder höher
- Datenbank: MySQL oder PostgreSQL
- Tools: git, composer

Diese Anforderungen werden in weiterer Folge an das System gestellt. Die Installation dieser Komponenten werden in diesem Kapitel nicht beschrieben.

C.2. Lokal

Um eine nicht verteilte Installation von symCloud durchzuführen, müssen folgende Schritte (siehe Listing C.1) ausgeführt werden:

Listing C.1: Herunterladen von symCloud

```
1 git clone git@github.com:symcloud/symcloud-standard.git
2 cd symcloud-standard
```

[1] http://docs.sulu.io/en/latest/book/getting-started/vhost.html

```
3 git checkout 0.1
4 cp app/config/admin/symcloud.yml.dist app/config/admin/symcloud.yml
5 cp app/Resources/pages/overview.xml.dist app/Resources/pages/overview.
    xml
```

Die Konfiguration der Installation erfolgen über die Dateien `app/admin/config/admin/symcloud.yml` und `app/Resources/webspaces/symcloud.io.xml`[2]. Diese beiden Dateien enthalten die Informationen über die URLs und die verbundenen Installationen.

Listing C.2: Installieren von symCloud

```
1 composer install
```

Diese beiden Scripts (Listing C.1 und C.2) laden die nötigen Quellcode herunter und installiert die Abhängigkeiten. Um die Installation abzuschließen werden je nach System folgende Scripts ausgeführt, um die richtigen Rechte zu setzen.

Verwende folgendes Script um die Rechte auf Linux (siehe Listing C.3) zu setzen:

Listing C.3: Berechtigungen setzen in Linux

```
1 rm -rf app/cache/*
2 rm -rf app/logs/*
3 mkdir app/data
4 mkdir app/data/symcloud
5 sudo setfacl -R -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
    uploads/media web/uploads/media app/data
6 sudo setfacl -dR -m u:www-data:rwx -m u:`whoami`:rwx app/cache app/logs
     uploads/media web/uploads/media app/data
```

Für Mac OSX (siehe Listing C.4) folgendes Script:

Listing C.4: Berechtigungen setzen in Mac OSX

```
1 rm -rf app/cache/*
2 rm -rf app/logs/*
3 mkdir app/data
4 mkdir app/data/symcloud
5 APACHEUSER=`ps aux | grep -E '[a]pache|[h]ttpd' | grep -v root | head
    -1 | cut -d\  -f1`
```

[2] http://docs.sulu.io/en/latest/book/getting-started/setup.html#webspaces

```
6 sudo chmod +a "$APACHEUSER allow delete,write,append,file_inherit,
      directory_inherit" app/cache app/logs uploads/media web/uploads/
      media app/data
7 sudo chmod +a "`whoami` allow delete,write,append,file_inherit,
      directory_inherit" app/cache app/logs uploads/media web/uploads/
      media app/data
```

Anschließend werden über folgendes Kommando (siehe Listing C.5) die Datenbank initialisiert, eine Administrator BenutzerIn eingerichtet und der Speicher für die AdministratorIn vorbereitet.

Listing C.5: SULU und symCloud konfigurieren

```
1 app/console doctrine:database:create
2 app/console sulu:build dev
3 app/console symcloud:storage:init admin
```

Die Ausgabe des letzten Befehls sollte notiert werden, da dies für die Einrichtung des Synchronisierunsclient gebraucht wird. Abschließend kann sich die AdministratorIn über `http://symcloud.lo/admin` einloggen (Benutzername: “admin”, Passwort: “admin”) und das System benutzen.

C.3. Jibe

Für den Client muss zuerst folgender Schritte (siehe C.6) auf dem Server ausgeführt werden:

Listing C.6: OAuth2 Client erstellen

```
1 app/console symcloud:oauth2:create-client jibe www.example.com
```

An die Endgeräte werden die selben Anforderungen wie an den Server gestellt. Der PHAR Container kann unter der URL <???> heruntergeladen werden. Die Ausgabe der Kommandos aus Listing C.6 und C.5 werden benötigt, um den Client zu Konfigurieren (siehe C.7). Die beiden Platzhalter `<hash-algorithm>` und `<hash-key>` werden ersetzt mit den in der Installation (siehe Listing C.2) angegeben werden.

Listing C.7: Jibe konfigurieren und starten

```
1 php jibe.phar configure --hash-algorithm <hash-algorithm> --hash-key <
    hash-key>
2 php jibe.phar sync
```

Das zweite Kommando startet direkt eine Synchronisierung des aktuellen Ordners.

C.4. Verteilt

Um eine verteilte Installation durchzuführen, werden die Schritte auf den vorangegangen Abschnitt auf mindestens zwei verschiedenen Servern durchgeführt (oder der selbe Server mit verschiedenen VHosts). Um die Installationen zu verbinden wird in der Konfigurationsdatei `app/admin/config/admin/symcloud.yml` die verbunden Server abgelegt.

Listing C.8: Verteilung in symCloud konfigurieren

```
1 symcloud_storage:
2     servers:
3         primary: {host: my.symcloud.lo}
4         backups:
5             - {host: your-1.symcloud.lo}
6             - {host: your-2.symcloud.lo}
```

Im Listing C.8 werden die verbundenen Server angegeben. Wobei für den primary Server die URL des aktuellen Servers und unter den Backups eine Liste von weiteren Servern angegeben werden.

C.5. Zusammenfassung

Dieses Kapitel beschreibt den Installationsprozess von symCloud. Es zeigt, dass die Installation ohne große Abhängigkeiten und zeitlicher Aufwand erledigt werden kann. Auch die Konfiguration in einer verteilten Umgebung ist mit nur wenigen Schritten möglich.

Literaturverzeichnis

Accenture [2012]: Hauptbedenken der Nutzer von Cloud-Diensten in Österreich im Jahr 2012. Online im Internet: http://de.statista.com/statistik/daten/studie/297078/umfrage/bedenken-bei-der-nutzung-von-cloud-diensten-in-oesterreich/

Amazon-Web-Services [2015a]: Amazon S3. Online im Internet: http://aws.amazon.com/de/s3/ [Zugriff am: 08.04.2015].

Amazon-Web-Services [2015b]: Introduction to Amazon S3. Online im Internet: http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html [Zugriff am: 13.05.2015].

Amazon-Web-Services [2015c]: Object Key and Metadata. Online im Internet: http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html [Zugriff am: 08.04.2015].

Amazon-Web-Services [2015d]: Using Versioning. Online im Internet: http://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html [Zugriff am: 08.04.2015].

Anglin, M.J. [2011]: Data deduplication by separating data from meta data. Google Patents Online im Internet: https://www.google.com/patents/US7962452

Atwood, Jeff [2009]: The Xanadu Dream. Online im Internet: http://blog.codinghorror.com/the-xanadu-dream [Zugriff am: 17.06.2015].

B. Kolbeck, J. Stender, M. Högqvist [2010]: Fault-Tolerant and Decentralized Lease Coordination in Distributed Systems. Online im Internet: http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733

Basho Technologies, Inc. [2015]: Riak CS. Online im Internet: http://docs.basho.com/riakcs/latest [Zugriff am: 12.04.2015].

Birrell, Andrew; Needham, Roger [1980]: „A Universal File Server“ IEEE Transactions on Software Engineering 1980/5, S. 450–453 Online im Internet: https://birrell.org/andrew/papers/UniversalFileServer.pdf

Bleiholder, Jens [2004]: Techniken des Data Merging in Integrationssystemen. Online

im Internet: https://static.aminer.org/pdf/PDF/000/237/730/techniken_des_data_merging_in_integrationssystemen.pdf

Chacon, S. [2009]: Pro Git. Apress [= Expert's voice in computer software development]. Online im Internet: https://books.google.at/books?id=HrTOa8-HPRYC

Chacon, Scott [2015]: Git Book - The Git Object Model. Online im Internet: http://schacon.github.io/gitbook/1_the_git_object_model.html

Coulouris, G.F.; Dollimore, J.; Kindberg, T. [2003]: Verteilte Systeme: Konzepte und Design. Pearson Education Deutschland [= Informatik - Pearson Studium]. Online im Internet: http://books.google.at/books?id=FfsQAAAACAAJ

Diasporafoundation [2014a]: Federation protocol overview. Online im Internet: https://wiki.diasporafoundation.org/Federation_protocol_overview [Zugriff am: 26.05.2015].

Diasporafoundation [2014b]: Installation Guide. Online im Internet: https://wiki.diasporafoundation.org/Installation/Ubuntu/Precise [Zugriff am: 24.06.2015].

Diasporafoundation [2015]: Was ist Dezentralisierung. Online im Internet: https://diasporafoundation.org/about [Zugriff am: 05.03.2015].

Dick, Hardt [2012]: The OAuth 2.0 authorization framework. Online im Internet: https://tools.ietf.org/html/rfc6749

Donauer, Jürgen [2014]: Server2Server - Sharing. Online im Internet: https://www.bitblokes.de/2014/07/server-2-server-sharing-mit-der-owncloud-7-schritt-fuer-schritt [Zugriff am: 24.07.2014].

Dropbox [2015a]: Core API Dokumentation. Online im Internet: https://www.dropbox.com/developers/core/docs [Zugriff am: 26.03.2015].

Dropbox [2015b]: Was versteht man unter einem Dateikonflikt. Online im Internet: https://www.dropbox.com/help/36 [Zugriff am: 24.06.2015].

Dropbox [2015c]: Wie funktioniert der Dropbox-Service. Online im Internet: https://www.dropbox.com/help/1968 [Zugriff am: 26.03.2015].

Dropbox, The Next Web, TechCrunch [2014]: Anzahl der Dropbox-Nutzer weltweit zwischen Januar 2010 und Mai 2014 (in Millionen). Online im Internet: http://de.statista.com/statistik/daten/studie/326447/umfrage/anzahl-der-weltweiten-dropbox-nutzer/

eco [2014]: Zustimmung zu der Aussage: Der NSA-Skandal hat das Vertrauen in Cloud-Dienste beschädigt. Online im Internet: http://de.statista.com/statistik/daten/studie/316667/umfrage/vertrauensverlust-bei-cloud-diensten-durch-nsa-skandal-in-deutschland/

Fielding, R. [2014]: Hypertext Transfer Protocol (HTTP/1.1): Range Requests. Online im Internet: https://tools.ietf.org/html/rfc7233

Fitzpatrick, B. [2014]: PubSubHubbub Core 0.4 - Working Draft. Online im Internet: http://pubsubhubbub.github.io/PubSubHubbub/pubsubhubbub-core-0.4.html

Fuchs, Jochen G. [2013]: Cloud-Dienste für Startups „Automatisierung ist Pflicht“. Online im Internet: http://t3n.de/news/cloud-dienste-startups-amazon-web-services-486480/ [Zugriff am: 13.05.2015].

Gohring, Nancy [2013]: Amazon-Web-Services We'll go to court to fight gov't requests for data. Online im Internet: http://www.itworld.com/article/2705826/cloud-computing/amazon-web-services--we-ll-go-to-court-to-fight-gov-t-requests-for-data.html [Zugriff am: 19.06.2013].

Gray, Jim u. a. [1996]: The Dangers of Replication and a Solution. Online im Internet: http://doi.acm.org/10.1145/233269.233330

Hammer, Eran [2010]: OAuth Core 1.0. Online im Internet: http://hueniverse.com/oauth/guide/history/ [Zugriff am: 14.05.2015].

Hewlett-Packard [2013]: Open Source Private Cloud Software - AWS-Compatible - HP Helion Eucalyptus. Online im Internet: https://www.eucalyptus.com/eucalyptus-cloud/iaas [Zugriff am: 14.05.2015].

HostedFTP [2009]: Amazon S3 and EC2 Performance Report – How fast is S3. Online im Internet: https://hostedftp.wordpress.com/2009/03/02/

ISO [2011]: System and software quality models. Online im Internet: http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733

Jones, P. [2013]: WebFinger. Online im Internet: https://tools.ietf.org/html/rfc7033

Keepers, Brandon [2012]: Git: The NoSQL Database. Online im Internet: http://devslovebacon.com/conferences/bacon-2012/talks/git-the-nosql-database

Kolbeck, Björn [2014]: XtreemFS Installation and User Guide. Online im Internet: http://www.xtreemfs.org/xtfs-guide-1.5/index.html

Lightcubesolutions [2010]: MongoDB and PHP - A Quick Look at GridFS. Online im Internet: http://www.lightcubesolutions.com/blog/?p=209 [Zugriff am: 26.06.2015].

Mesbah, A.; Deursen, A. van [2007]: Migrating Multi-page Web Applications to Single-page AJAX Interfaces. Online im Internet: http://www.st.ewi.tudelft.nl/~arie/papers/spci/csmr2007.pdf

MongoDB [2015]: GridFS. Online im Internet: http://docs.mongodb.org/manual/core/gridfs/ [Zugriff am: 27.03.2015].

Nelson, Theodor Holm [1981]: Literary Machines: The Report On, and Of, Project Xanadu Concerning Word Processing, Electronic Publishing, Hypertext, Thinkertoys. XOC Online im Internet: https://books.google.at/books?id=LCFAXwAACAAJ

Nelson, Theodor Holm; Smith, Robert Adamson; Mallicoat, Marlene [2007]: Back to the Future: Hypertext the Way It Used to Be. Online im Internet: http://doi.acm.org/10.1145/1286240.1286303

OpenHUB [2009]: The Wizbit Open Source Project on Open Hub. Online im Internet: https://www.openhub.net/p/wizbit [Zugriff am: 13.05.2015].

ownCloud [2015a]: ownCloud Architecture Overview. Online im Internet: https://owncloud.com/de/owncloud-architecture-overview

ownCloud [2015b]: ownCloud Features. Online im Internet: https://owncloud.org/features [Zugriff am: 05.03.2015].

Paul, Ryan [2008]: Wizbit a Linux filesystem with distributed version control. Online im Internet: http://arstechnica.com/information-technology/2008/10/wizbit-a-linux-filesystem-with-distributed-version-control/

Reed, Benjamin; Junqueira, Flavio [2008]: A Simple Totally Ordered Broadcast Protocol. Online im Internet: http://doi.acm.org/10.1145/1529974.1529978

Schütte, Prof. Dr. Alois [2014]: Verteilte Dateisysteme. Online im Internet: http://www.fbi.h-da.de/~a.schuette/Vorlesungen/VerteilteSysteme/Skript/6_VerteilteDateisysteme/VerteilteDateisysteme.pdf [Zugriff am: 21.05.2015].

Seidel, Udo [2013]: „Dateisystem-Überblick“ Linux Magazin 2013/2

Stahlknecht, P.; Hasenkamp, U. [2013]: Einführung in die Wirtschaftsinformatik. Springer Berlin Heidelberg [= Springer-Lehrbuch]. Online im Internet: https://books.google.at/books?id=9B/_0BgAAQBAJ

Tanenbaum, A.S.; Steen, M. van [2003]: Verteilte Systeme: Grundlagen und Paradigmen. Pearson Education Deutschland GmbH [= I : Informatik]. Online im Internet: https://books.google.at/books?id=qXGnOgAACAAJ

Tridgell, Andrew; Mackerras, Paul [1996]: The rsync algorithm. anu-address: dcs-anu Online im Internet: https://digitalcollections.anu.edu.au/bitstream/1885/40765/3/TR-CS-96-05.pdf

Wiesmann, M. u. a. [2000]: Understanding replication in databases and distributed systems. Online im Internet: http://dx.doi.org/10.1109/ICDCS.2000.840959

XtreemFS [2014a]: Under the Hood: File Replication. Online im Internet: http://xtreemfs.org/how_replication_works.php [Zugriff am: 20.05.2015].

XtreemFS [2014b]: XtreemFS - architecture, internals and developer's documentation. Online im Internet: http://www.xtreemfs.org/arch.php [Zugriff am: 13.05.2015].