Evaluierung und Entwicklung eines Verteilten Speicherkonzeptes als Grundlage für eine Filehosting und Collaboration Platform

Wachter Johannes

Zusammenfassung

TEST

Abstract

TEST

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 seinem Mitmenschen zu teilen. Jedoch gibt es große Bedenken der BenutzerInnen im Bezug auf den Datenschutz, wenn sie Ihre Daten aus der Hand geben. Dieser Konflikt zeigen auch verschiedene Studien. Sie zeigen, dass es immer mehr NutzerInnen in die Cloud zieht (siehe Abbildung 1.1), dabei aber die Bedenken gegen, genau diese Anwendungen, zunehmen (siehe Abbildung 1.2).

Die Statistik aus der Abbildung 1.1 zeigt wie die Nutzerzahlen des Kommerziellen Cloud-Dienstes Dropbox[1] in den Jahren 2010 bis 2014 von anfänglich 4 Millionen auf 300 Millionen im Jahre 2014 angestiegen sind.

Im Gegensatz dazu wurde im Jahre 2012 in Österreich erhoben, dass nur etwa 17% der AnwenderInnen diese Dienste ohne Bedenken verwendet. Das meistgenannte Bedenken dieser Studie ist: **Fremdzugriff auf die Daten ohne Information**.

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

Diese Statistiken zeigen, dass immer mehr Menschen das Bedürfnis verspüren, die Kon-

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

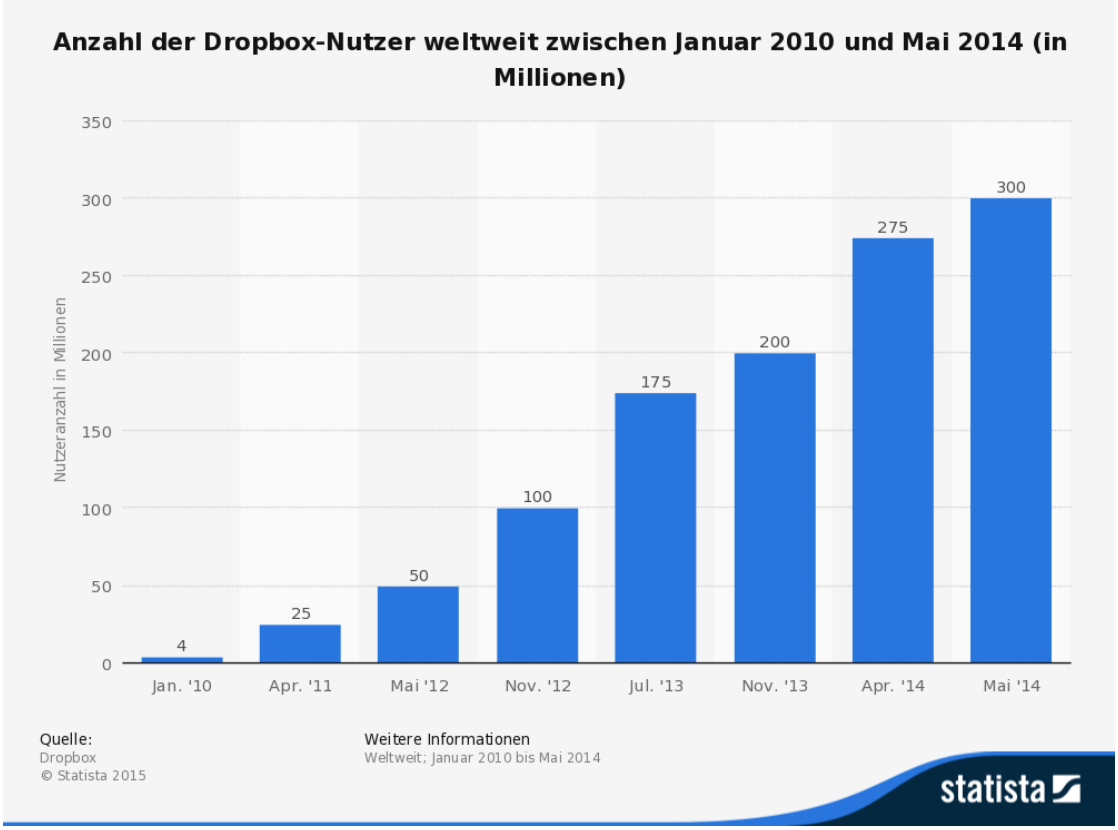

Abbildung 1.1: Anzahl der Dropbox-Nutzer weltweit zwischen Januar 2010 und Mai 2014 (in Millionen) [Dropbox 2014]

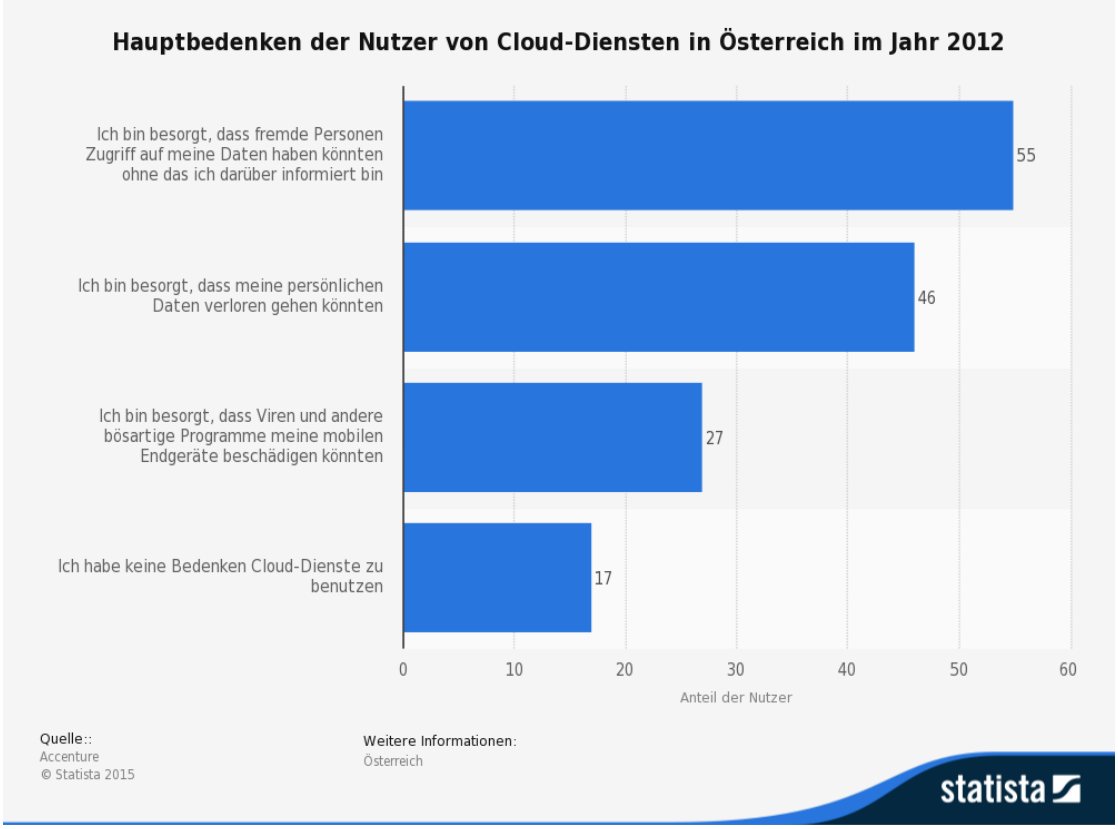

Abbildung 1.2: Hauptbedenken der Nutzer von Cloud-Diensten in Österreich im Jahr 2012 [Accenture 2012]

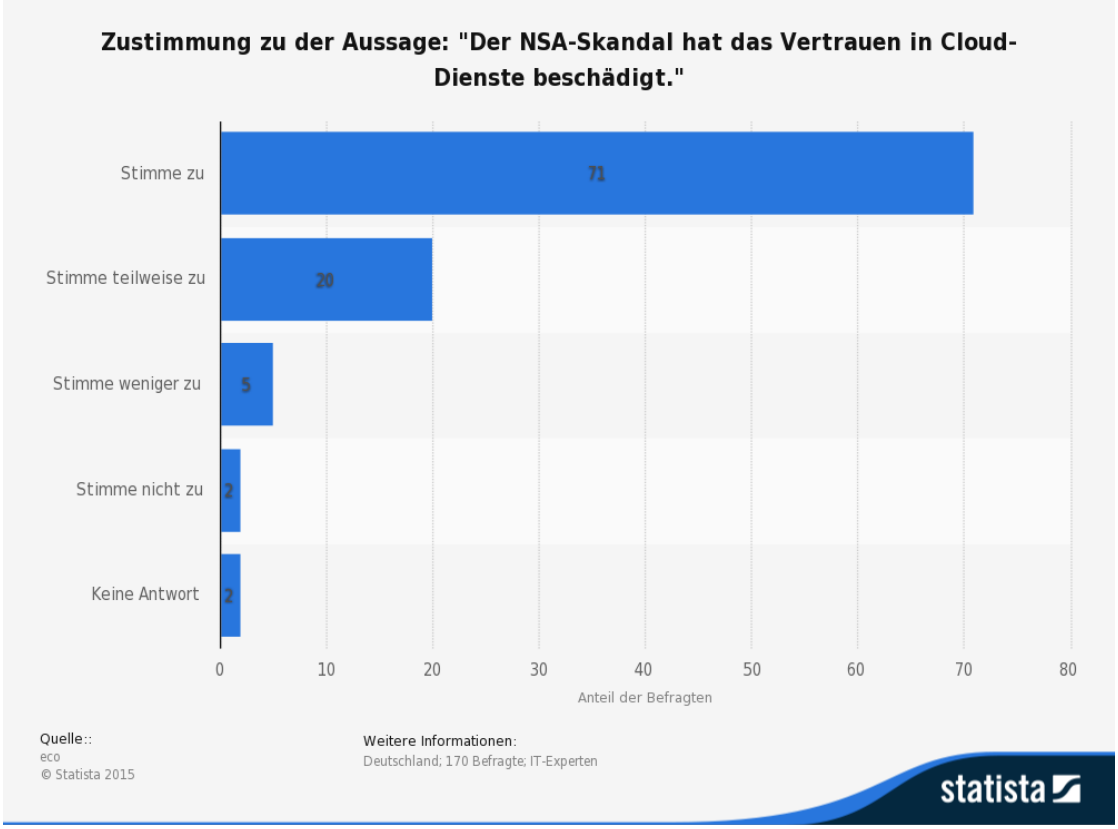

Abbildung 1.3: Zustimmung zu der Aussage: “Der NSA-Skandal hat das Vertrauen in Cloud-Dienste beschädigt.” [eco 2014]

trolle über ihre Daten zu behalten, aber trotzdem die Vorzüge solcher Dienste nutzen wollen. Aufgrund dessen erregen Projekte wie Diaspora[2], ownCloud[3] und ähnliche Softwarelösungen immer mehr Aufmerksamkeit.

Diaspora Diaspora ist ein dezentrales soziales Netzwerk. Die BenutzerInnen von diesem sozialen Netzwerk sind durch die verteilte Infrastruktur nicht von einem Dienstleister abhängig. Es ermöglicht, seinen Freunden bzw. der Familie, eine private "social-media" Plattform anzubieten und diese nach seinen Wünschen zu gestalten. Das Interessante daran sind die sogenannten Pods (dezentrale Knoten), die sich beliebig untereinander vernetzen lassen. Dies ermöglicht es auch Benutzern, die nicht auf dem Selben Servern registriert sind, miteinander zu kommunizieren. Pods können von jedem installiert und betrieben werden; dabei kann der Betreiber bestimmen, wer in sein Netzwerk eintreten darf und welche Server mit seinem 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 wurde in der Programmiersprache Ruby.

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 gängiger kommerzieller Lösungen an [ownCloud 2015b]. Zusätzlich bietet es eine Kollaborationsplattform, mit der zum Beispiel Dokumente über einen online Editor, von mehreren Benutzern gleichzeitig, bearbeitet werden können. Diese Technologie basiert auf der JavaScript Library WebODF[4]. Implementiert ist dieses Projekt Hauptsächlich in der Programmiersprache PHP.

[2] https://diasporafoundation.org/
[3] https://owncloud.org/
[4] http://webodf.org/

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. Aufgrund dieses Umstandes, ist es für Anwendungen oft sehr schwer sich für die breite Masse zu etablieren. In dieser Arbeit wird speziell auf die Anforderungen von Anwendungen eingegangen, die es ermöglichen soll, Dateien zu verwalten, 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.

1.1 Projektbeschreibung

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 den Benutzer irrelevant, woher die Daten stammen.

Wobei es bei der Arbeit weniger um die Plattform, als um die Konzepte geht, die es ermöglichen an 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, wird die Implementierung dieser Konzepte in einer eigenständigen Library entwickelt. Dieser Umstand ermöglicht eine Weiterverwendung in anderen Plattformen und Anwendungen, die ihren BenutzerInnen ermöglichen wollen, Dateien zu erstellen, verwalten, bearbeiten oder teilen. Damit kann das erstellte Konzept als Grundlage für eine “Spezifikation” von derartigen Prozessen weiterverwendet werden.

In der ersten Phase, in der diese Arbeit entsteht, werden Grundlegende Konzepte aufge-

stellt. Diese beginnen mit der Festlegung eines Datenmodells und der Implementierung einer Datenbank, die diese Daten mit anderen Server teilen kann. Dieses teilen von Daten soll voll konfigurierbar sein, was bedeutet, das 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 Benutzer (und damit die Server auf denen die Benutzer registriert sind) das Objekt tatsächlich verwenden dürfen.

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

1.2 Inspiration

Als Inspirationsquelle für das Konzept von symCloud dienten neben den schon erwähnten Applikationen auch das Projekt Xanadu[5]. Dieses Projekt wurde im Jahre 1960 von Ted Nelson gegründet und wurde 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:

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

Verbindungen Text besteht oft aus einer Menge von anderen Texten wie zum Beispiel Zitate oder Querverweise. Dies lässt 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 in bestimmte Form, welche später weder verändert noch erweitert werden kann.

Hypertext sollte nicht das Medium sondern die BenutzerInnen in den Vordergrund stellen. Durch verschiedene 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 Projekt Xanadu beschreiben sollten [Nelson 1981]. Einige davon wurden durch Tim Berners-Lee in der Erfindung des Internets umgesetzt, andere jedoch vernachlässigt [Atwood 2009]. Einige dieser Thesen, die vernachlässigt wurden, sind interessante Denkanstöße für ein Projekt wie symCloud.

1. Every Xanadu server can be operated independently or in a network.

Xanadu Server können für sich alleine arbeiten oder in einem Netzwerk interagieren.

2. Every user is uniquely and securely identified.

Jeder Benutzer ist eindeutig und sicher identifizierbar.

3. Every user can search, retrieve, create and store documents.

Jeder Benutzer kann Dokumente durchsuchen, herunterladen, erstellen und speichern.

4. Every document can have secure access controls.

Jedes Dokument besitzt eine Zugriffskontrolliste. In dieser Liste wird aufgeführt, welcher

Benutzer welche Rechte auf ein Dokument besitzt.

5. Every document can be rapidly searched, stored and retrieved without user knowledge of where it is physically stored.

Jedes Dokument kann schnell durchsucht, gespeichert und heruntergeladen werden, ohne das der Benutzer weiß wo das Dokument physikalisch gespeichert ist.

6. Every document is automatically stored redundantly to maintain availability even in case of a disaster.

Jedes Dokument wird redundant gespeichert, um den Verlust bei unvorhergesehenen Ereignissen zu verhindern.

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

1.3 Anforderungen

Aufgrund der beschriebenen Projekte, die als Inspiration verwendet wurden, werden in diesem Abschnitt die Anforderungen, an ein System wie 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 die Anforderung an die Architektur geprägt von verteilten Aspekten.

Nicht Ziele Diese Punkte sind wichtige Anforderungen an ein System wie symCloud, sie sind allerdings nicht Teil dieser Arbeit.

1.3.1 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 waren.

Konkrete Bedrohungen sind, die eines dieser drei Gefahren auslösen kann, 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 Projekt Xanadu bieten Ansätze, wie diese Anforderungen umgesetzt werden können. Durch die Redundanz (These sechs) 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. Die Thesen zwei und vier bieten einen Schutz vor dem Verlust derVertraulichkeit, indem die BenutzerIn eindeutig identifiziert werden kann und ein Zugriffsberechtigungssystem die Berechtigung überprüft. Dadurch kann ausgeschlossen werden, dass sich dritte über die Schnittstellen des System, zugriff auf Daten verschaffen, die sie nicht sehen dürften.

1.3.2 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 und 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, immer den selben Inhalt besitzen und über die selbe URL erreichbar sind. Dies ist speziell für Zitierungen wichtig. Die Zusammenarbeit zwischen den Benutzern und die Zugriffsberechtigungen sind ebenfalls zentrale Bestandteile von Xanadu.

1.3.3 Architektur

Inspiriert von der Architektur von Diaspora sollte es möglich sein, verschiedene Installation von symCloud zu einem Netzwerk zusammenschließen zu können. Dabei liegt der Fokus auf der Datenverteilung und nicht auf einer Lastverteilung. Dadurch können Daten gezielt im Netzwerk verteilt werden. Aufgrund der Datensicherheitsanforderungen sollten die Daten nicht wahllos im Netzwerk verteilt werden sondern Konzepte ausgearbeitet werden, um Daten aufgrund der Zugriffsberechtigungen auf das Netzwerk zu teilen.

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.

1.3.4 Nicht Ziele

Wichtige, aber in dieser Arbeit nicht betrachtete Ziele bzw. Anforderungen, sind:

Effizienz und Performance Die Effizienz und die Performance eines Systems ist meist nicht der Grunde für einen Erfolg, allerdings meist einer wichtigste bei einem Misserfolg.

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

Diese Ziele sind, wie schon erwähnt außerhalb des Fokuses dieser Arbeit und des Konzeptes, dass während dieser Arbeit entsteht. 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.

1.4 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 Datenbank gestützte Dateisysteme mit 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 einer Prototypen Implementierung umgesetzt. 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 Begriffe verteilte Systeme und verteilte Dateisysteme definiert. Anschließend werden drei Anwendungen beschrieben, die als Inspiration für das Projekt Symcloud verwendet werden.

2.1 Verteilte Systeme

Andrew Tanenbaum definiert verteilte 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 eine 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 der Benutzer, 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 an und umfassen Rechnerleistungen, Speicher, Datenbanken und Netzwerke. Der Anwender kommuniziert hierbei immer nur mit einem

System, allerdings verbirgt sich hinter diesen Anfragen 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. Dies ermöglicht eine Verbesserung von Datensicherheit, durch Replikation über verschiedene Server und Steigerung der 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 zum Beispiel 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 die Effizienz der Anfragen [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 Dropbox[1] oder Google Drive[2]. Andere jedoch sind frei verfügbar und wie zum Beispiel ownCloud[3] sogar Open-Source. Zwei dieser Applikationen werden hier etwas genauer betrachtet und soweit es möglich ist die Speicherkonzepte analysiert.

2.2.1 Dropbox

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

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

Alle diese Clients stellen Verbindungen mit sicheren Servern her, über die sie Zugriff auf Dateien haben und Dateien für andere Nutzer freigeben können. Wenn Daten auf einem Client geändert werden, werden diese automatisch mit dem Server synchronisiert. Verknüpfte Geräte aktualisieren sich 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 [„Wie funktioniert der Dropbox-Service“ 2015].

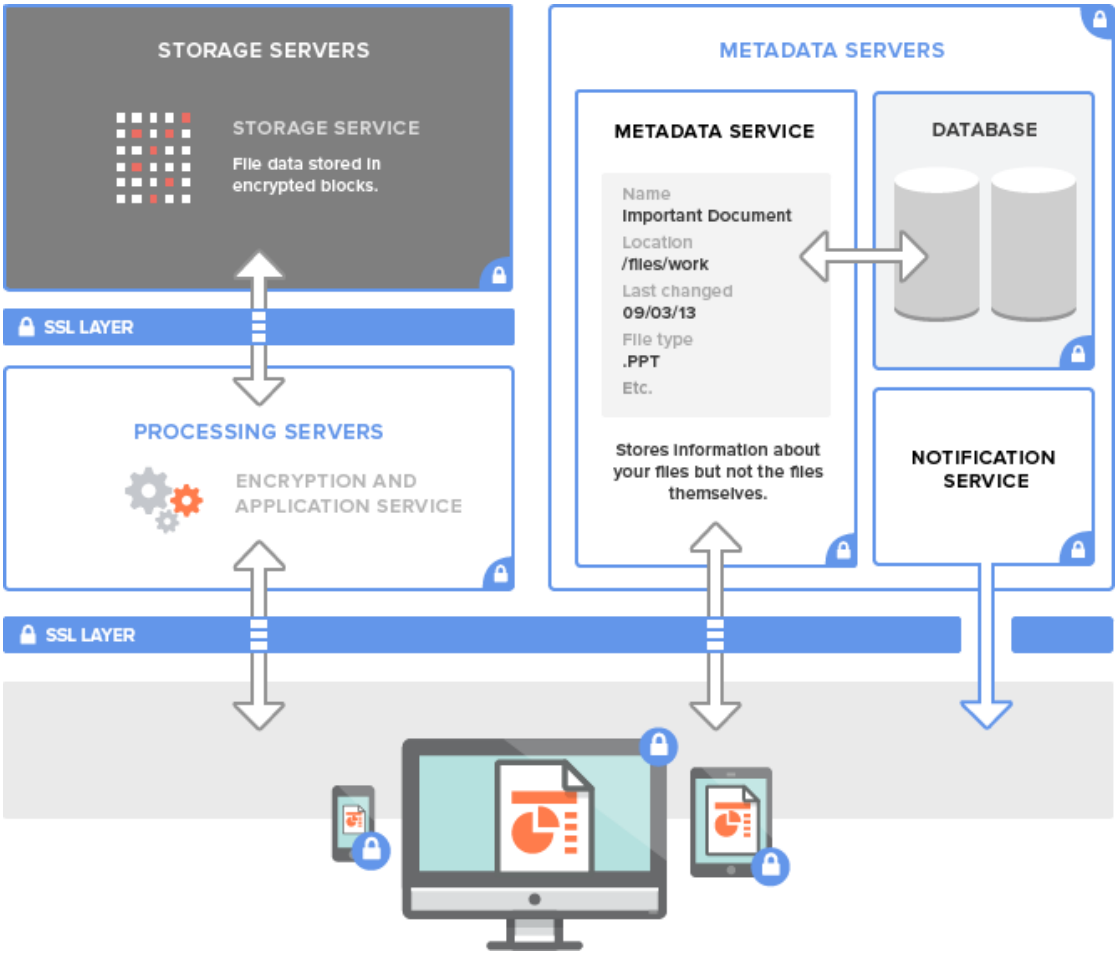

Abbildung 2.1: Blockdiagramm der Dropbox Services [„Wie funktioniert der Dropbox-Service“ 2015]

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 Informationen zu ihrem Speicherort in einer Datenbank, aber der Inhalt der Daten liegt 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 [„Core API Dokumentation“ 2015]. Diese Authentifizierung wird für alle Services verwendet, auch für den Metadata Service, 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 Dritthersteller Anwendungen zu entwickeln, die mit Daten aus der Dropbox arbeiten. Für diesen Zweck gibt es im Authentifizierungsprotokoll OAuth2 sogenannte Scopes (siehe Kapitel 7.5). Eine weitere Aufgabe, die diese Schicht erledigt, ist die Verschlüsselung der Anwendungsdaten [„Wie funktioniert der Dropbox-Service“ 2015].

Die Nachteile von Dropbox im Bezug auf die im Kapitel 1.3 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 verwenden.

2.2.2 ownCloud

Nach den neuesten Entwicklungen arbeitet ownCloud an einem ähnlichen Feature wie Symcloud. Unter dem Namen “Remote shares” wurde in der Version 7 eine Erweiterung in den Core übernommen, mit dem es möglich sein soll, sogenannte “Shares” mittels einem Link auch in einer anderen Installation einzubinden. Dies ermöglicht es, Dateien auch über die Grenzen des eigenen Servers hinweg zu teilen [„Server2Server - Sharing“ 2015].

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

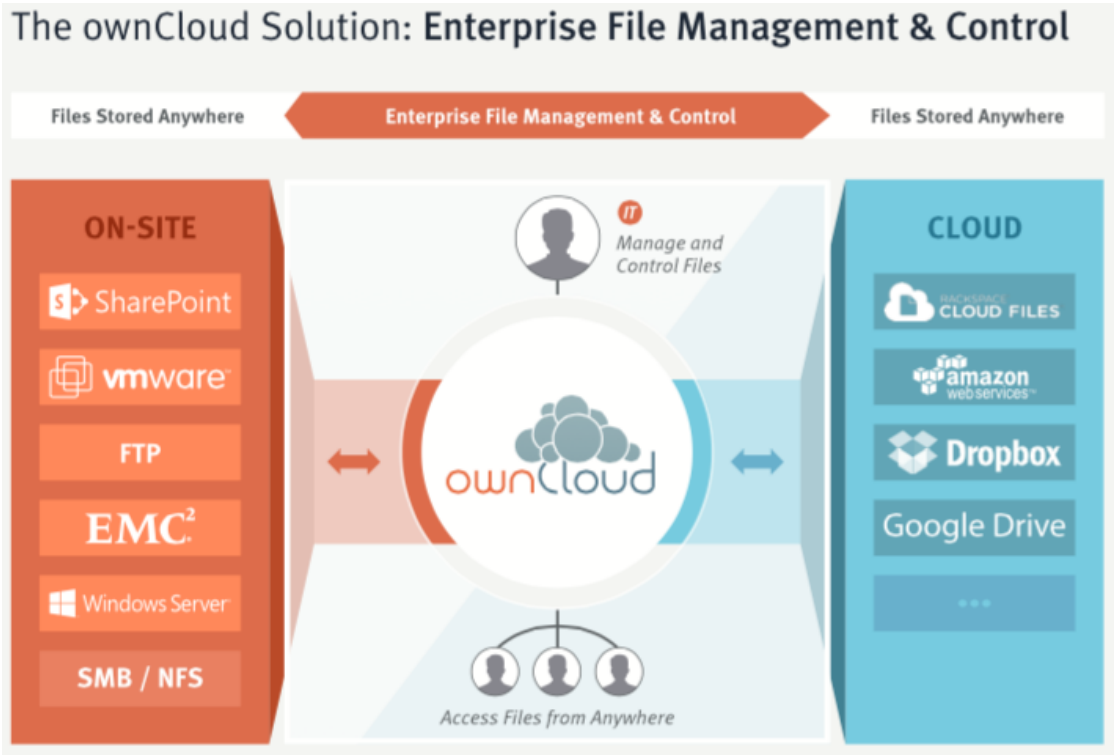

Abbildung 2.2: ownCloud Enterprise Architektur Übersicht [ownCloud 2015a]

Um die Integration in ein Unternehmen zu erleichtern, bietet es verschiedenste Services an. Unter anderem ist es möglich, Benutzerdaten über LDAP oder ActiveDirectory zu verwalten und damit ein doppeltes Verwalten der Benutzer zu vermeiden [ownCloud 2015a, S. 2].

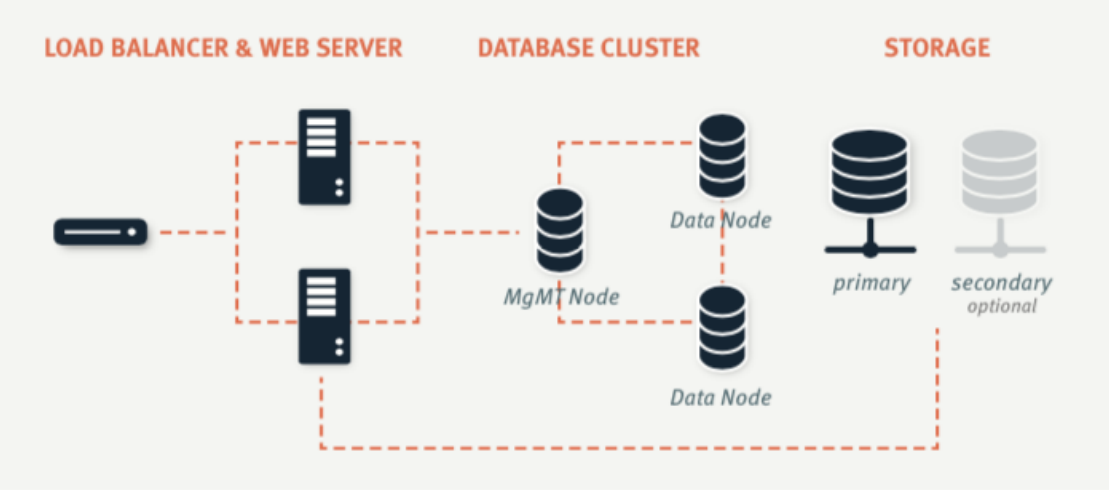

Abbildung 2.3: Bereitstellungsszenario von ownCloud [ownCloud 2015a]

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 an 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. Dies ermöglicht auch bei stark frequentierten Installationen eine horizontale Skalierbarkeit. Zusätzlich sind die Webserver mit dem File-Storage verbunden. Auch hier ist es möglich, diesen redundant bzw. skalierbar aufzubauen, um die Effizienz und Sicherheit zu erweitern [ownCloud 2015a, S. 3–4].

Die Nachteile von ownCloud im Bezug auf die im Kapitel 1.3 aufgezählten Anforderungen sind:

Architektur Die Software ist dafür ausgelegt, um die Anforderungen, auf einem einzigen Server zu erfüllen. Es ermöglicht zwar eine verteilte Architektur, allerdings nur, um

die Last auf verschiedene Server zu verteilen. Im Gegensatz dazu versucht symCloud die Daten zwischen verschiedenen Instanzen zu verteilen um die Zusammenarbeit zwischen Benutzern zu ermöglichen, die auf verschiedenen Servern registriert sind.

Moderne Programmierung Aufgrund der Tatsache, dass ownCloud schon im Jahre 2010 und sich die Programmiersprache PHP und die Community rasant weiterentwickelt, ist der Kern von ownCloud in einem überholten Stil programmiert.

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

2.3 Verteilte Daten - Beispiel Diaspora

Diaspora ist ein gutes Beispiel für Applikationen, die ihre Daten über die Grenzen eines Servers hinweg verteilt. Diese Daten werden mithilfe von Standardisierten Protokollen über einen sicheren Transport-Layer 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 [„Federation protocol overview“ 2015]. In folgenden Situationen wird dieses Protokoll verwendet:

- Um Benutzerinformationen zu finden, die auf anderen Servern registriert sind.
- Erstellte Informationen an Benutzer zu versenden, mit denen sie geteilt wurden.

Diaspora verwendet das Webfinger Protokoll, um zwischen den Servern zu kommunizieren. Das Webfinger Protokoll wird verwendet, um Informationen über Benutzer oder anderen Objekte abfragen zu können. Identifiziert werden diese Objekte über eine eindeutige URI. Es verwendet den HTTP-Standard als Transport-Layer über eine sichere Verbindung. Als Format für die Antworten wird JSON verwendet [Jones 2013, S. 1].

Beispiel [„Federation protocol overview“ 2015]:

Alice (alice@alice.diaspora.example.com) versucht mit Bob `bob@bob.diaspora.example.com` in Kontakt zu treten. Zuerst führt der Pod von Alice `alice.diaspora.example.com` einen Webfinger lookup auf den Pod von Bob (bob.diaspora.example.com) aus. Dazu führt Alice eine Anfrage auf die URL `https://bob.diaspora.example.com/.well-known/host-meta`[4] (siehe Listing 2.1) aus und erhält einen Link zum LRDD (“Link-based Resource Descriptor Document”[5]).

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" />
```

Unter diesem Link können Objekte auf dem Server von Bob gesucht werden. Als nächster Schritt führt der Server von Alice einen GET-Request auf den LRDD mit den kompletten Benutzernamen von Bob als Query-String aus. Der Response retourniert folgendes Objekt:

Listing 2.2: LRDD Inhalt von Bob

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <XRD xmlns="http://docs.oasis-open.org/ns/xri/xrd-1.0">
3   <Subject>acct:bob@bob.diaspora.example.com</Subject>
4   <Alias>"http://bob.diaspora.example.com/"</Alias>
5   <Link rel="http://microformats.org/profile/hcard"
6         type="text/html"
7         href="http://bob.diaspora.example.com/hcard/users/((guid))"/>
8   <Link rel="http://joindiaspora.com/seed_location"
9         type="text/html" href="http://bob.diaspora.example.com/"/>
10  <Link rel="http://joindiaspora.com/guid" type="text/html"
11        href="((guid))"/>
```

[4] https://tools.ietf.org/html/rfc6415#section-2
[5] https://tools.ietf.org/html/rfc6415#section-6.3

```
12   <Link rel="http://schemas.google.com/g/2010#updates-from"
13         type="application/atom+xml"
14         href="http://bob.diaspora.example.com/public/bob.atom"/>
15   <Link rel="diaspora-public-key" type="RSA"
16         href="((base64-encoded rsa public key))"/>
17 </XRD>
```

Das Objekt enthält die Links zu weiteren, in Diaspora gespeicherten, Informationen des Benutzers, welcher im Knoten “Subject” angeführt wird.

Dieses Beispiel zeigt, wie Diaspora auf einfachste Weise Daten auf einem sicheren Kanal austauschen kann.

2.4 Verteilte Datenmodelle - Beispiel GIT

GIT[6] 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. Es werden Objekte in Form einer Datei abgespeichert, in der jeweils der Inhalt des Objekts abgespeichert wird, wobei der Name der Datei den Key des Objektes enthält. Dieser Key wird berechnet, indem ein sogenannter SHA berechnet wird. Der SHA ist ein mittels “Secure-Hash-Algorithm” berechneter Hashwert der Daten [Chacon 2009, S. 9.2]. Das Listing 2.3 zeigt, wie ein SHA in einem Unix-Terminal berechnet werden kann [Keepers 2012].

Listing 2.3: Berechnung des SHA eines Objektes

```
1 $ OBJECT='blob 46\0{"name": "Johannes Wachter", \
2   "job": "Web-Developer"}'
3 $ echo -e $OBJECT | shasum
4 6c01d1dec5cf5221e86600baf77f011ed469b8fe -
```

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

Im Listing 2.4 wird ein GIT-Objekt vom Typ BLOB erstellt und in den “objects” Ordner geschrieben.

Listing 2.4: Erzeugung eines GIT-BLOB

```
1 $ git init
2 $ OBJECT='blob 46\0{"name": "Johannes Wachter", \
3   "job": "Web-Developer"}'
4 $ echo -e $OBJECT | git hash-object -w --stdin
5 6c01d1dec5cf5221e86600baf77f011ed469b8fe
6 $ find .git/objects -type f
7     .git/objects/6c/01d1dec5cf5221e86600baf77f011ed469b8fe
```

Die Objekte in GIT sind immutable, was soviel 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, S. 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 keinen Namen. Dieser Name wird

zu jeder Referenz auf einen TREE oder auf einen BLOB gespeichert (siehe Listing 2.5) [Chacon 2009, S. 9.2].

Listing 2.5: 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 einen den ROOT-TREE des Repositories zu einem bestimmten Zeitpunkt.

Listing 2.6: 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 2.6):

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

Zeile	Name	Beschreibung
2	commit	SHA des Objektes

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

Anmerkungen (zu der Tabelle 2.1):

- Ein COMMIT kann mehrere Vorgänger haben, wenn sie zusammengeführt werden. Zum Beispiel würde dieser Mechanismus bei einem MERGE verwendet werden, um die beiden Vorgänger zu speichern.
- Autor und Ersteller des COMMITs können sich unterscheiden: Wenn zum Beispiel ein Benutzer einen PATCH erstellt, ist er der Verantwortliche für die Änderungen und damit der Autor. Der Benutzer, der den Patch nun auflöst und den `git commit` Befehl ausführt, ist der Ersteller 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 kurzes Beispiel für ein Repository visualisiert. Die Ordnerstruktur, die dieses Beispiel enthält, ist im Listing 2.7

Listing 2.7: Ordernstruktur zum Repository Beispiel

```
1 |-- README
2 |-- lib
3     |-- inc
4     |   |-- tricks.rb
5     |-- mylib.rb
```

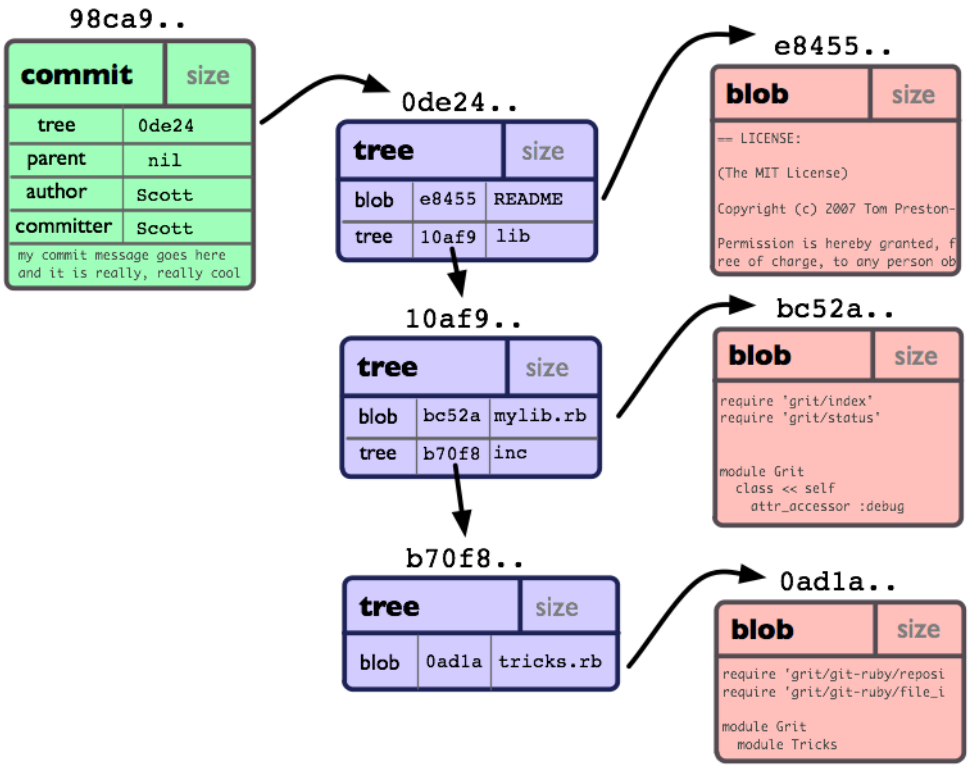

Abbildung 2.4: Beispiel eines Repositories [Chacon 2015]

Wobei `README`, `tricks.rb` und `mylib.rb` Dateien und die beiden anderen `lib` und `lib/inc` Ordner sind. Der COMMIT enthält eine Referenz auf den ROOT-TREE, der wiederum auf den ein TREE-Objekt mit dem Namen `lib`, welcher schlussendlich eine Referenz auf das TREE-Objekt mit dem Namen `inc` enthält. Jeder dieser drei TREE-Objekte enthält jeweils eine Referenz auf einen BLOB mit den jeweiligen Namen der Datei.

Die Nachteile von GIT im Bezug auf die im Kapitel 1.3 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. Ein weiteres Problem, dass bei der Verwendung von GIT entstehen würde, ist die fehlende Möglichkeit Zugriffsberechtigungen festzulegen.

Da die Anwendung GIT für die Verwendung als Datenspeicher, aufgrund der Fehlenden Verteilungsmechanismen, für das Projekt ungeeignet ist, aber das Datenmodell die meisten der Anforderungen erfüllen würde, wird dieses Datenmodell, in Kapitel 4.2, als Grundlage für das Datenmodell von symCloud herangezogen.

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 kann.

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 können.

Aufgrund der Anforderungen, siehe Kapitel 1.3 des Projektes werden folgende Kriterien an die Speicherlösung gestellt.

Ausfallsicherheit Die Speicherlösung ist das Fundament einer 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 eine wichtige Anforderung an eine Speicherlösung die Skalierbarkeit.

Datenschutz Der Datenschutz ist ein wichtiger Punkt beim Betreiben der eigenen Cloud-Anwendung. Meist gibt es eine kommerzielle Konkurrenz, die mit günstigen Preisen die Anwender anlockt, um ihre Daten zu verwerten. Die Möglichkeit, Daten privat auf dem eigenen Server zu speichern, sollte somit gegeben sein. Damit Systemadministratoren nicht auf einen Provider angewiesen sind.

Flexibilität Um Daten flexibel speichern zu können, sollte es möglich sein, Verlinkun-

gen 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 ist ein wichtiger Aspekt an eine 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 zu reduzieren diesen aktuell zu halten.

3.1 Datenhaltung in Cloud-Infrastrukturen

Es gibt unzählige Möglichkeiten, um die Datenhaltung in Cloud-Infrastrukturen umzusetzen. Insbesondere werden in diesem Kapitel drei grundlegende Technologien und Beispiele dafür analysiert.

Objekt-Speicherdienste Speicherdienste wie zum Beispiel Amazon S3[1], ermöglichen das Speichern von sogenannten 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 [„Introduction to Amazon S3“ 2015].

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

[1] http://aws.amazon.com/de/s3/

`fclose`. Dies ergibt sich aus der Transparenz Anforderung [Coulouris ; Dollimore ; Kindberg 2003, S. 369], die im Kapitel 3.3.1 beschrieben wird.

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 abzuspeichern [„GridFS“ 2015].

Aufgrund der vielfältigen Möglichkeiten werden zu jedem der drei Technologien ein oder zwei Beispiele als Referenz hergenommen.

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 Datenmengen [„Amazon S3“ 2015]. 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 speichern auch automatisch aktualisiert [„Object Key and Metadata“ 2015]. Für eine vollständige Liste dieser Metadaten siehe Anhang 7.5.

Zusätzlich zu diesen systemdefinierten Metadaten ist es möglich, benutzerdefinierte

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

Metadaten zu speichern. 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

```
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 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 [„Using Versioning“ 2015]. Dies veranschaulicht die Abbildung 3.1.

Abbildung 3.1: Versionierungsschema von Amazon S3 [„Using Versioning“ 2015]

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 mit dem Kunden abgesprochen wird [„Amazon Web Services: We'll go to court to fight gov't requests for data | ITworld“ o. J.]. Dies gilt aber vermutlich nicht für Anfragen der NSA, den diese beruhen in der Regel auf den Anti-Terror Gesetzen und verpflichten daher den Anbieter zur absoluten Schweigepflicht. Um dieses Problem zu kompensieren, können Systemadministratoren sogenannte “Availability Zones” auswählen und damit steuern, wo ihre Daten gespeichert werden. Zum Beispiel werden Daten aus einem Bucket mit der Zone Irland, auch wirklich in Irland gespeichert. Zusätzlich ermöglicht Amazon die Verschlüsselung der Daten [„Cloud-Dienste für Startups: „Automatisierung ist Pflicht“ [Interview] | t3n“ o. J.].

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

3.2.4 Alternativen zu Amazon S3

Es gibt einige Amazon S3 kompatible Anbieter, die einen ähnlichen Dienst bieten. Diese sind allerdings meist auch US-Amerikanische Firmen und daher an die selben 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 eine Installation von einer privaten Cloud-Lösungen herum.

[3] Amazons Zuständiger für die Zusammenarbeit zwischen den Partner

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) [„Open Source Private Cloud Software | AWS-Compatible | HP Helion Eucalyptus" o. J.]. Dieser S3 kompatible Dienst bietet allerdings keine Versionierung.

Riak Cloud Storage ist eine Software, mit der es möglich ist, einen verteilten Objekt-Speicherdienst zu betreiben. Es implementiert die Schnittstelle von Amazon S3 und ist damit kompatibel zu der aktuellen Version [Basho Technologies 2015]. Es 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 Perfomance Report über ihre Erfahrungen mit der Performance zwischen EC2 (Rechner Instanzen) und S3 [„Amazon S3 and EC2 Performance Report – How fast is S3" 2009]. Über ein Performance Modell wurde festgestellt, dass die Zeit für den Download einer Datei in zwei Bereiche aufgeteilt werden kann.

Feste Transaktionszeit ist ein fixer Zeitabschnitt, der für die Bereitstellung oder Erstel-

lung der Datei benötigt wird. Beeinflusst wird diese Zeit kaum, allerdings kann es aufgrund schwankender Auslastung zu Verzögerungen kommen.

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 über die Dateigröße aufweist. Diese These wird von den Daten unterstützt. Aus dem Diagramm (Abbildung 3.2) kann die feste Transaktionszeit von ca. 140ms abgelesen werden.

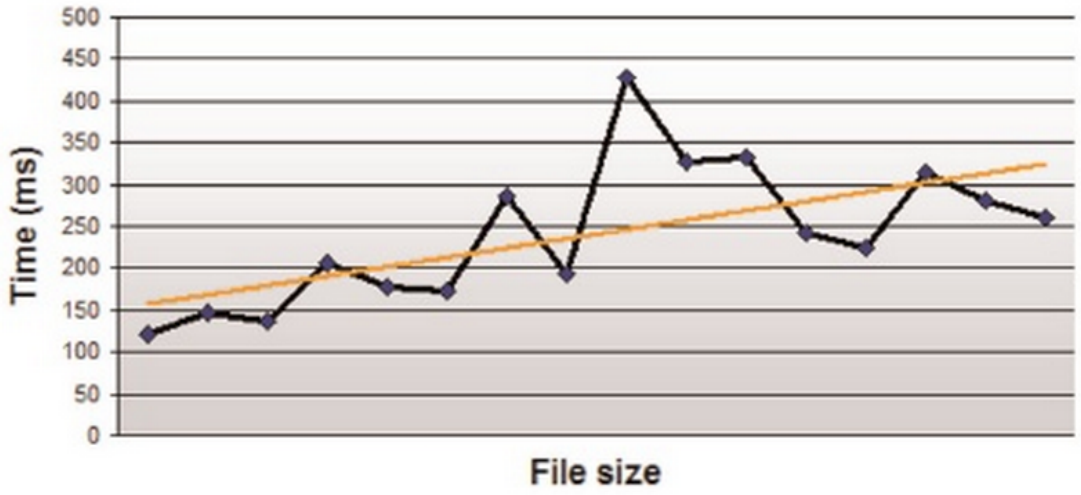

Abbildung 3.2: Upload Analyse zwischen EC2 und S3 [„Amazon S3 and EC2 Performance Report – How fast is S3“ 2009]

Für den Download von Dateien entsteht laut den Daten aus dem Report 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 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 anzupassen. Die Schreibzugriffe bzw. Lesezugriffe erfolgen über ganz normale `system-calls` [Coulouris ; Dollimore ; Kindberg 2003, S. 363ff.].

Dies ist auch ein großer Vorteil 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 verteilt oder lokal, über die selbe Operationsmenge verfügen. Es sollte egal sein, ob Daten aus einem verteilten oder lokalen Dateisystem stammen. Dadurch können Programme unverändert weiterverwendet werden, wenn seine Dateien verteilt werden [Coulouris ; Dollimore ; Kindberg 2003, S. 369ff].

Ortstransparenz Es sollte keine Rolle spielen, wo die Daten physikalisch gespeichert werden [Schütte o. J., S. 5]. Das Programm sieht immer den selben Namensraum, egal wo er ausgeführt wird [Coulouris ; Dollimore ; Kindberg 2003, S. 369ff].

Nebenläufige Dateiaktualisierungen Dateiänderungen, die von einem Client ausgeführt

werden sollten die Operationen anderer Clients, die die selbe Datei verwenden, nicht stören. Um diese Anforderung zu erreichen, 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 es erhöht die Fehlertoleranz. Wenige Dateisysteme unterstützen vollständige Replikationen, aber die meisten unterstützen 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 o. J., S. 5].

Konsistenz In konventionellen Dateisystemen werden Zugriffe auf Dateien auf eine einzige Kopie der Daten geleitet. Wird nun diese Datei auf mehrere Server 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 [Cou-

louris ; Dollimore ; Kindberg 2003, S. 369ff].

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

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

Aufgrund der Tatsache, dass Festplatten heutzutage nicht mehr so teuer sind, wie in den 1980ern, ist das erste Ziel nicht mehr von zentraler Bedeutung. Jedoch ist die Vorstellung von einem Dienst, der die Anforderung verschiedenster Clients mit unterschiedlichen Aufgabenstellungen erfüllt, 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 und über diese Dateien des lokalen Speichers den Benutzern zur Verfügung stellt. 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 können durch NFS eine heterogene Menge von Computern verbunden

werden. Dabei ist es sowohl für den Benutzer 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].

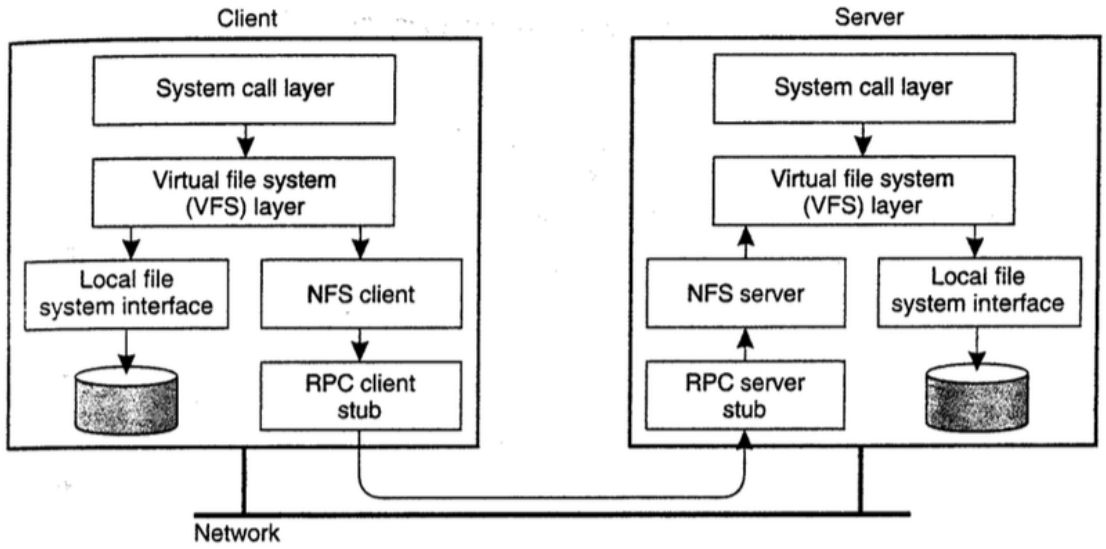

Abbildung 3.3: NFS Architektur [Tanenbaum ; Steen 2003, S. 647]

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

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

kümmert. Dabei fungiert der Client als eine Art Stub-Implementierung der Schnittstelle und leitet alle Anfragen an den entfernten Server weiter (RPC). Diese Abläufe werden aufgrund des VFS-Konzeptes vollkommen transparent für den Benutzer 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 physikalischen 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 [„XtreemFS Installation and User Guide“ o. J., S. 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), welche automatisch verwendet werden, wenn ein Server ausfällt [„XtreemFS Installation and User Guide“ o. J., S. 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 [„XtreemFS Installation and User Guide" o. J., S. 2.3].

Security Um die Sicherheit der Dateien zu gewährleisten, unterstützt XtreemFS sowohl Benutzer Authentifizierung als auch Berechtigungen. 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] [„XtreemFS Installation and User Guide" o. J., S. 2.3].

Architektur

XtreemFS implementiert eine Objektbasierte Datei-Systemarchitektur, das 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 Servern gespeichert. Diese Server organisieren die Dateien in eine Menge von sogenannten "Volumes". Jedes Volume ist ein eigener Namensraum mit einem eigenen Dateibaum. Die Metadaten speichern zusätzlich eine Liste von Chunk-IDs mit den jeweiligen Servern, auf denen dieser Chunk zu finden ist und eine Richtlinie, wie diese Datei aufgeteilt und auf die Server verteilt werden soll. Daher kann die Größe der Metadaten von Datei zu Datei unterschiedlich sein [„XtreemFS Installation and User Guide" o. J., S. 2.4].

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

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

DIR Der "Directory-Service" ist das Zentrale Registern, indem alle anderen Services aufgelistet werden. Die Clients oder andere Services verwenden ihn, um zum Beispiel die "Object-Storage-Devices" oder "Metadata- and Replica-Catalogs" zu finden [„XtreemFS Installation and User Guide" o. J., S. 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" seine Chunks abgelegt wurden. Zusätzlich authentifiziert und autorisiert er dem Benutzer den Zugriff auf die Dateien bzw. Ordner [„XtreemFS Installation and User Guide" o. J., S. 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 [„XtreemFS Installation and User Guide" o. J., S. 2.4].

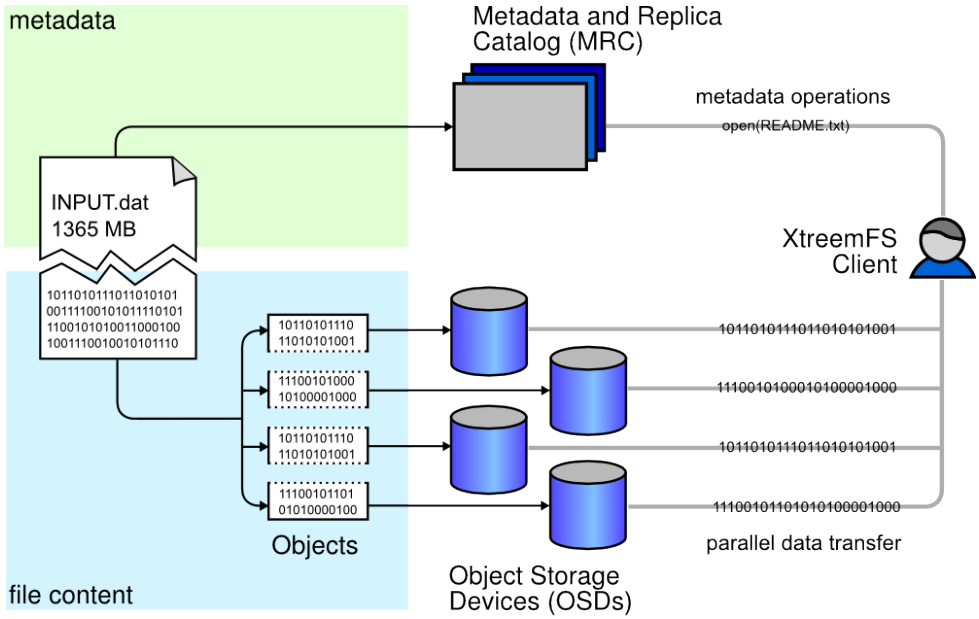

Abbildung 3.4: XtreemFS Architektur [„XtreemFS - architecture, internals and developer's documentation" o. J.]

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. Wenn Daten repliziert werden ist es unter Umständen möglich weiterzuarbeiten, wenn eine Replik ausfällt. Der Benutzer lädt sich die Daten von einem anderen Server herunter. Zusätzlich dazu können durch Repliken fehlerhafte Dateien erkannt werden. Wenn eine Datei zum Beispiel auf drei Servern gespeichert wurde und alle Schreib- bzw. Lesezugriffe auf alle drei Server ausgeführt wurden, 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, der eine bezieht sich auf die gesamte Last eines einzigen Servers und der andere auf 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 Lesezugriff 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 umzusetzen. Diese Leistungssteigerung ist allerdings nicht linear zu den verwendeten Servern. Denn 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 eingesetzt [„XtreemFS Installation and User Guide“ o. J., S. 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 Lokalen-Schreiben.

Entferntes-Schreiben

Es gibt zwei Arten zur Implementierung dieses Protokolls. Das eine ist ein nicht replizierendes Protokoll, bei dem alle Schreib- und Lesezugriffe auf denn primären Server des Objektes ausgeführt werden. Und 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 [„XtreemFS Installation and User Guide“ o. J., S. 6.1.3].

Der Prozess, der eine Schreiboperation (siehe Abbildung 3.5) auf das Objekt ausführen will, gibt sie an denn 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 weiter. Nachdem alle Backups die Aktualisierung durchgeführt haben, gibt auch der primäre Server eine Bestätigung an den ausführenden Server weiter. Dieser Server kann nun sicher sein, dass die Aktualisierung auf allen Servern ausgeführt wurde 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

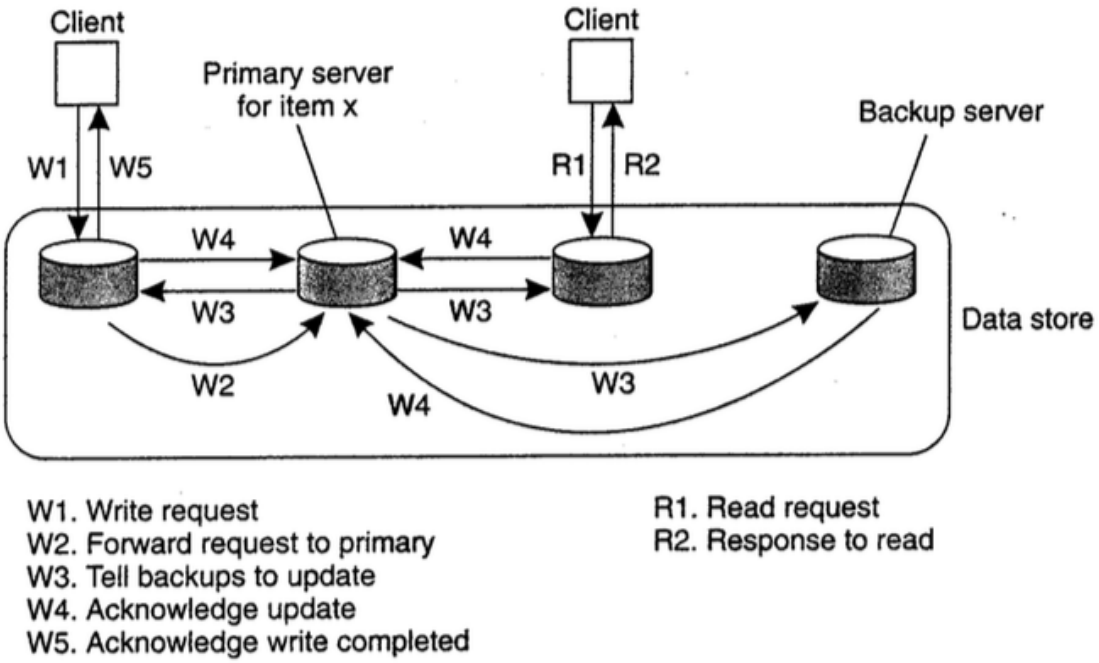

Abbildung 3.5: Primary-Backup-Protokoll: Entferntes-Schreiben [Tanenbaum ; Steen 2003, S. 385]

bedeutet, dass der primäre Server die Bestätigung direkt nach dem lokalen Ausführen der Operation zurückgibt und erst danach die Aktualisierungen an die Backups weitergibt [„Under the Hood: File Replication“ o. J.]. Aufgrund der Tatsache, dass alle Schreiboperationen auf einem Server ausgeführt werden, können diese einfach abgesichert werden 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 eine ist ein nicht replizierendes Protokoll, bei dem vor einem Schreibzugriff das Objekt auf den ausführenden Server verschoben wird und dadurch der primäre Server des Objekts geändert wird. Nachdem die Schreiboperation ausgeführt wurde, bleibt das Objekt auf diesem Server solange, bis ein anderer Server schreibend auf das Objekt zugreifen

will. Die andere Möglichkeit ist ein “Primäres-Backup Protokoll” (siehe Abbildung 3.6), bei dem der primäre Server des Objektes zu dem ausführenden Server migriert wird [Tanenbaum ; Steen 2003, S. 386ff].

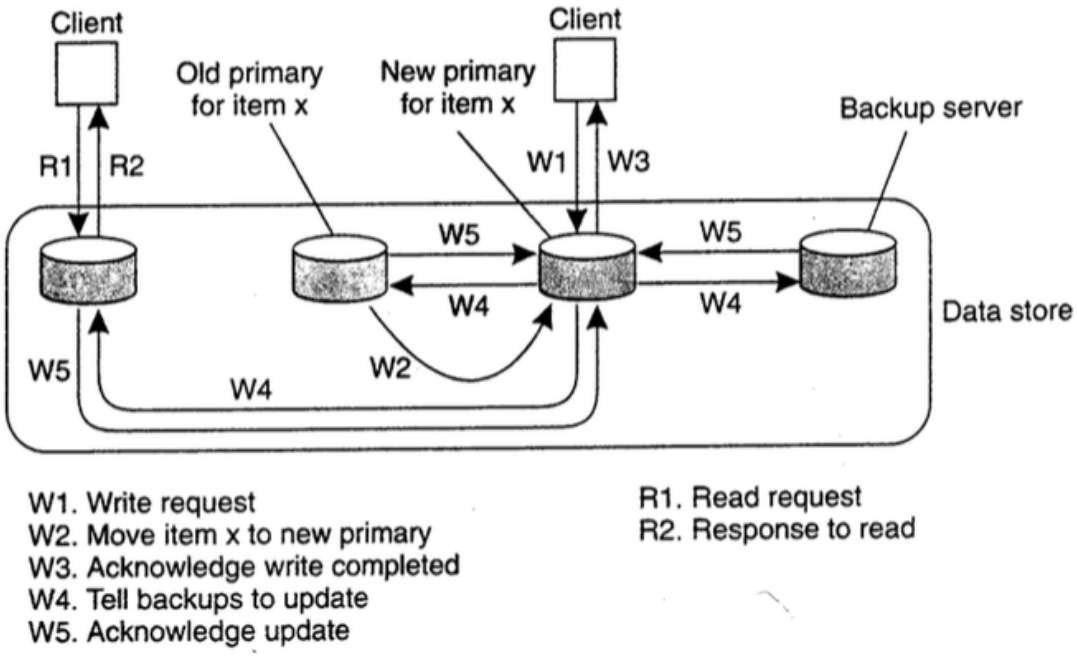

Abbildung 3.6: Primary-Backup-Protokoll: Lokales-Schreiben [Tanenbaum ; Steen 2003, S. 387]

Dieses Protokoll ist auch für mobile Computer geeignet, die in einem Offline-Modus verwendet werden können. Dazu wird es 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 Repliken 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 wieder in einen konsistenten Zustand übergehen kann [Tanenbaum ; Steen 2003, S. 386ff].

3.3.5 Zusammenfassung

Im Bezug auf die Anforderungen (siehe Kapitel 1.3) 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, das Versionierung mitliefern sollte [„Wizbit: a Linux filesystem with distributed version control | Ars Technica“ 2008]. Dieses Projekt wurde allerdings seit Ende 2009 nicht mehr weiterentwickelt [„The Wizbit Open Source Project on Open Hub“ o. J.]. Die benötigten Zugriffsberechtigungen werden zwar auf der Systembenutzerebene durch ACL unterstützt, jedoch müssten dann die Anwendungen für jeden Anwendungsbenutzer einen Systembenutzer anlegen [„XtreemFS Installation and User Guide“ o. J., S. 7.2]. Dies wäre zwar auf 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 werden und in ein Gesamtkonzept mit eingebunden werden.

TODO fehlender Bezug zum Projekt

3.4 Datenbankgestützte Dateiverwaltungen

Einige Datenbanksysteme, wie zum Beispiel MongoDB[7], bieten eine Schnittstelle an, 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 wird diese Technologie im folgenden Kapitel genauer betrachtet.

[6] https://www.openhub.net/p/wizbit
[7] http://docs.mongodb.org/manual/core/gridfs/

3.4.1 MongoDB & GridFS

MongoDB bietet die Möglichkeit, BSON-Dokumente in der Größe von 16MB zu speichern. Dies ermöglicht die Verwaltung kleinerer Dateien ohne zusätzliche Layer. Für größere Dateien und zusätzliche Features bietet MongoDB mit GridFS eine Schnittstelle an, mit der es möglich ist, größere Dateien und ihre 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. Außerdem bietet das System die Möglichkeit, über Indexe schnell zu suchen und Abfragen auf die Metadaten durchzuführen.

Beispiel:

Listing 3.2: GridFS Beispielcode

```
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);
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 zu S3 wesentliche Mankos, die gerade für das aktuelle 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 der Art der Datenhaltung beobachtet werden. Wie zum Beispiel:

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

Diese Punkte werden im Kapitel 4 berücksichtigt werden.

Verteilte Dateisysteme Die verteilten Dateisysteme, bieten durch ihre einheitliche Schnittstelle einen optimalen Abstraktionslayer für datenintensive Anwendungen. Die Flexibilität, die diese Systeme verbindet, kommen der der Anwendung in

Symcloud entgegen. Jedoch sind fehlende Zugriffsrechte auf Anwendungsbenutzerebene (ACL) und die fehlende Versionierung ein Problem, das auf Speicherebene nicht gelöst wird. Aufgrund dessen könnte ein solches verteiltes Dateisystem nicht als Ersatz für eine eigene Implementierung, sondern lediglich als Basis dafür hergenommen werden.

Datenbankgestützte Dateiverwaltung Systeme wie zum Beispiel GridFS sind für den Einsatz in Anwendungen geeignet, die die darunterliegende Datenbank verwendet. 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 Benutzer ihre Dateien ablegen. Die Anwendung müsste dafür sorgen, dass jeder Benutzer 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 adäquate Lösung für die Datenhaltung in Symcloud bietet, wird im nächsten Kapitel versucht ein optimales Speicherkonzept für das aktuelle 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 unter den verbundenen Servern. Die Datenbank beinhaltet eine Suchmaschine, mit der es möglich ist, die 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

Die Architektur ist gegliedert in Kern- und optionale Komponenten. In der Abbildung 4.1 sind die Abhängigkeiten der Komponenten untereinander zu erkennen. 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, wenn die Daten direkt auf die Festplatte geschrieben werden. 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

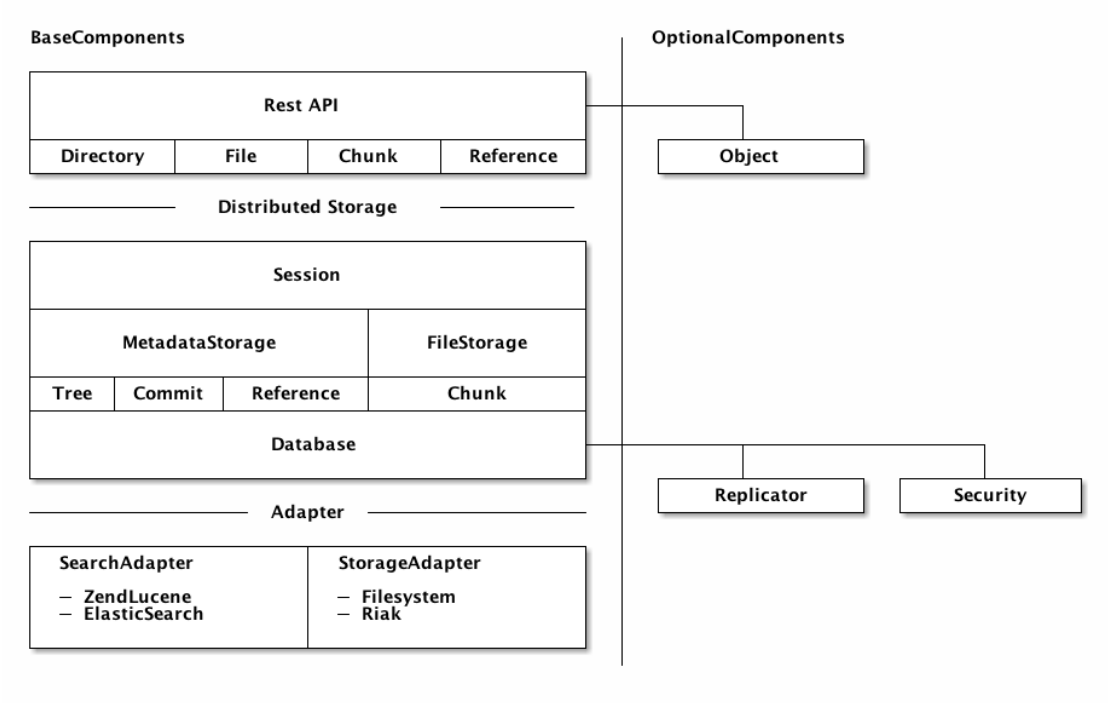

Abbildung 4.1: Architektur für "Symcloud-Distributed-Storage"

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 1.3) 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 erfüllt folgende Anforderungen von Symcloud:

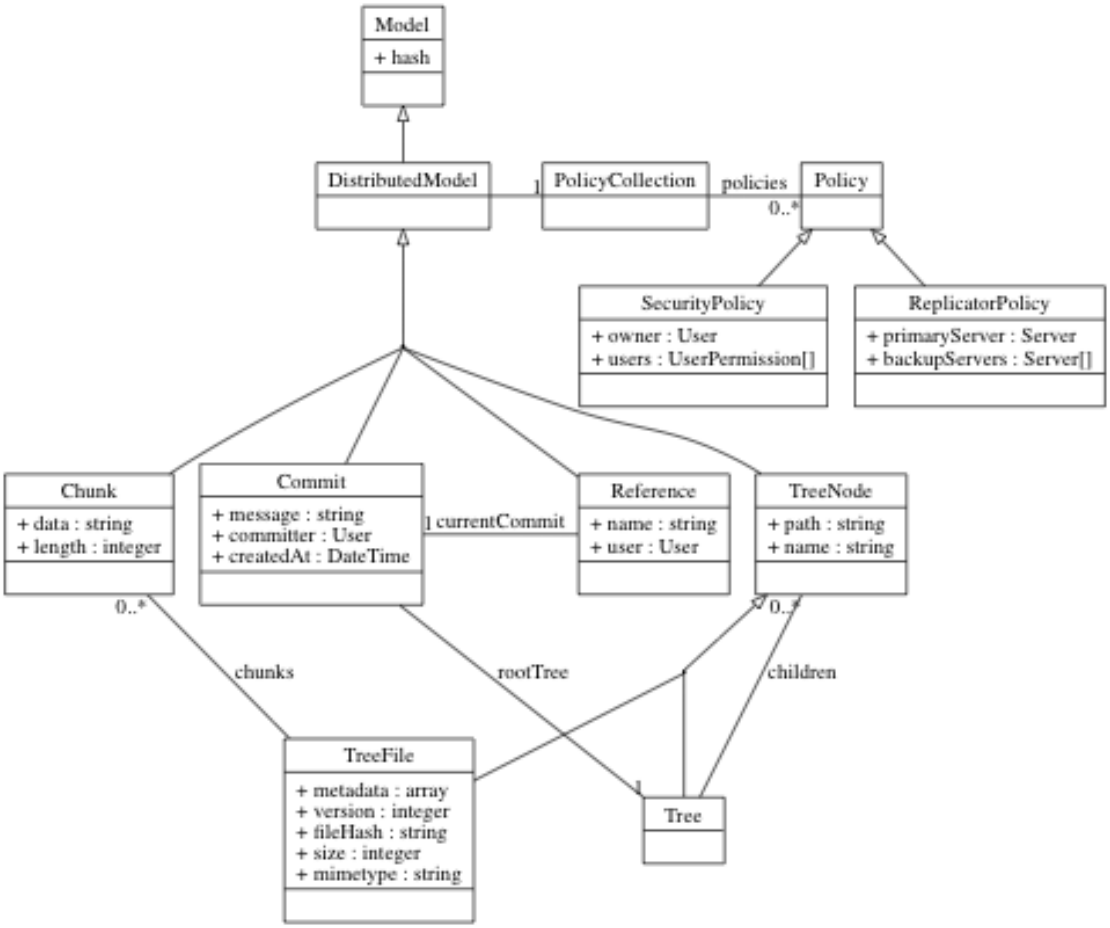

Abbildung 4.2: Datenmodel für “Symcloud-Distributed-Storage”

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

Namensräume Mit den Referenzen können für jeden Benutzer mehrere Namensräume geschaffen werden. Jeder dieser Namensräume erhält einen eigenen Dateibaum und kann von mehreren Benutzern verwendet werden. Damit können Shares einfach abgebildet werden. Jede Referenz kann für Benutzer 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, S. 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 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) oder 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 Nicht berücksichtigt wurde, im Datenmodel von GIT, die Zuordnung der Referenzen zu einem Benutzer. Diese Zuordnung wird von Symcloud verwendet, um die Zugriffsrechte zu realisieren. Ein Benutzer kann einem anderen Benutzer die Rechte auf eine Referenz übertragen, auf die er 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 backup Servern, auf denen das Objekt gespeichert wurde.

4.3 Datenbank

Die Datenbank ist eine einfache “Hash-Value” Datenbank, die mithilfe eines Replikators zu einer verteilen Datenbank erweitert wird. Die Datenbank serialisiert die Objekte und speichert sie mithilfe 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 zuzugreifen.

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

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

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 verteilt 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 (siehe Kapitel 7.2).

Permissions Wenn ein Objekt auf Basis der Zugriffsrechte verteilt wird, wird es auf allen Servern erstellt, die mindestens einen Benutzer registriert haben, der Zugriff auf dieses Objekt besitzt. Dabei gibt es keine Maximalanzahl der Backupserver. Dieses Verfahren wird für kleinere Objekte, die zum Beispiel Datei- bzw. Ordnerstrukturen enthalten, verwendet. Die Verteilung kann sofort ausgeführt werden oder die Objekte werden beim ersten Zugriff jeden Backupservers "Lazy" nachgeladen. Der Vorteil der "Lazy" Technik ist es, dass die Server nicht immer erreichbar 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 Änderung mitbekommen. Um die Datensicherheit für diese Objekte zu erhöhen, könnten aus dem Serverpool eine konfigurierbare Anzahl von Backupservern, wie bei dem Full Typen, 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.

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. Dies bedeutet, dass das Objekt als eine Art remote Objekt fungiert. 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 nachzuladen.

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

4.4 Metadatastorage

Der Metadatastorage verwaltet die Struktur der Daten. 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 des Benutzers, 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 werden und die Backupserver automatische Updates zu Änderungen erhalten.

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 diesen 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, mit eingebettet und stehen daher nur konkreten Dateien zur Verfügung. Das bedeutet, dass eine konkrete Datei, eine Liste von Chunks besitzt, die die eigentlichen Daten repräsentieren. Diese 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. Da diese

keine Zugriffsrechte besitzen, spielt es keine Rolle, ob dieser von dem selben oder von einem anderen Benutzer wiederverwendet wird. Wenn der Hash übereinstimmt, besitzen beide Dateien der Benutzer den selben 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 den selben 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, wird an diese Datei nun weiterer Inhalt angehängt. Die neue Version besteht 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[2] (siehe Kapitel 7.4) [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].

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

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" 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 beschränkt auf den Speicherplatz des kleinsten Servers. 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 werden 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 sind dabei unter anderem der Inhalt des Ordners (Dateien oder andere Ordner).

File Unter dem Pfad `/file/<reference-name>/<directory>/<filename>.<extension>` können Dateien heruntergeladen werden oder ihre Informationen abgefragt werden.

Reference Die Schnittstelle für die Referenzen erlaubt das Erstellen und Abfragen von Referenzen. Zusätzlich können mittels PATCH-Requests Dateien, aus dem Namensraum einer bestimmten Referenz, geändert und diese gesammelt versioniert werden.

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 1.3 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 wichtige 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 ein einfaches UI an, mit dem 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 ein erweiterbares Admin-UI, eine Benutzerverwaltung und ein Rechtesystem an. Diese Features ermöglichen symCloud eine schnelle Entwicklung der Oberfläche und deren zugrundeliegenden Services.

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

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

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

5.1 Distributed-Storage

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

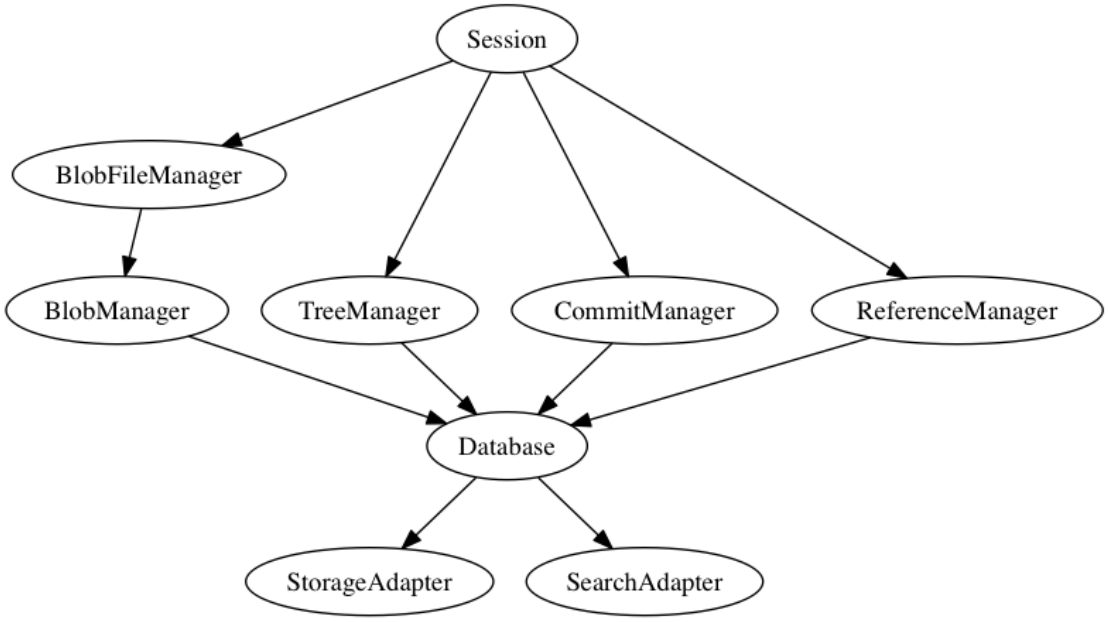

Abbildung 5.1: Schichten von “Distributed Storage”

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 Um die Komplexität der jeweiligen Objekte zu abstrahieren, implementieren die Manager die jeweilige Funktionalität, um mit diesen Objekten zu kommunizieren. Die Objekte sind dabei reine Daten-Container.

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 mithilfe dieser Metadaten die Objekte wieder deserialisiert werden.

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. Mithilfe 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 Replikationstypen "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 Model 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, geben die Metadaten der Klasse, einen eindeutigen Kontext zurück. Dieser Kontext wird den Adaptern übergeben, um Kollisionen zwischen den Datensätzen zu verhindern.

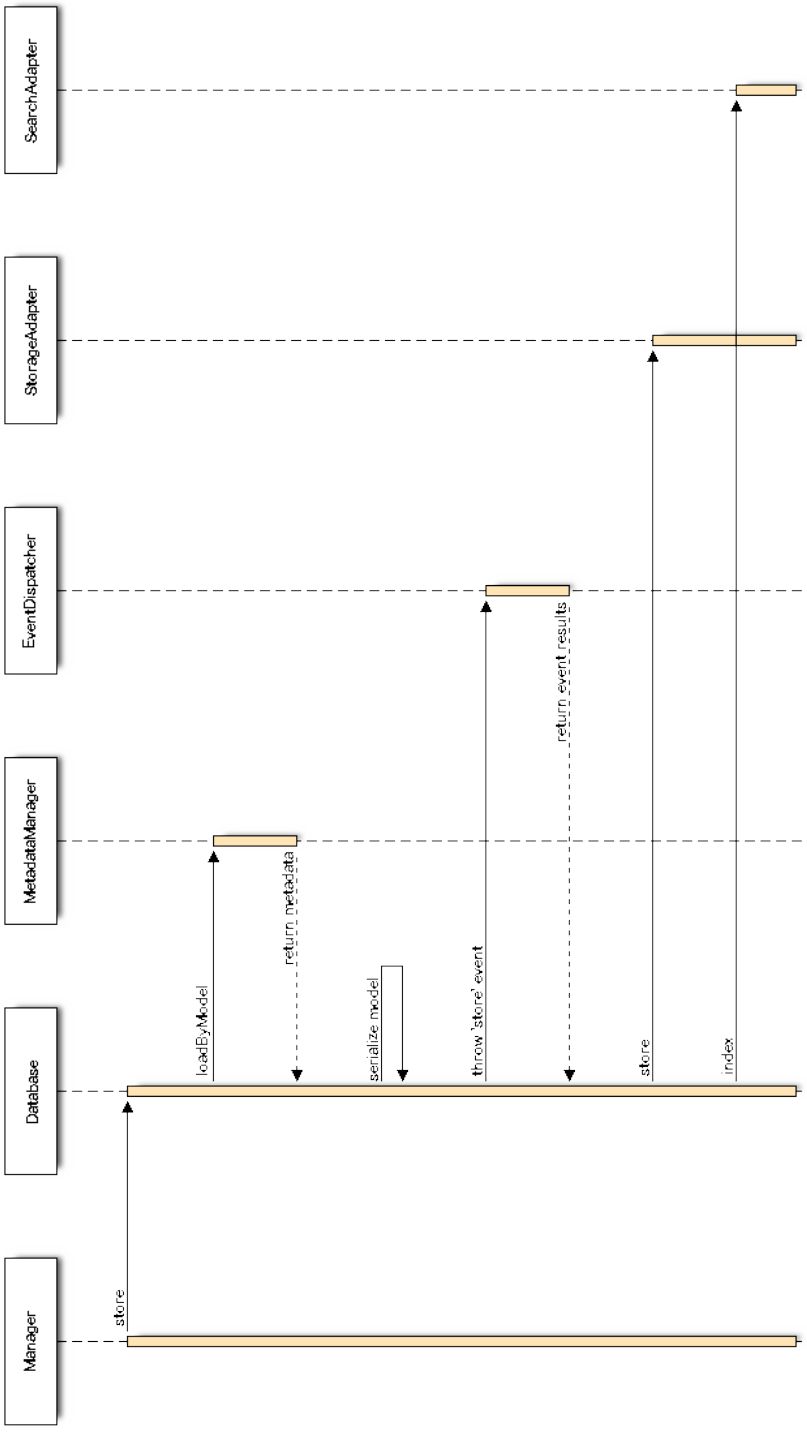

Abbildung 5.2: Objekte speichern

5.1.2 Objekte abrufen

Wie zu erwarten 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.

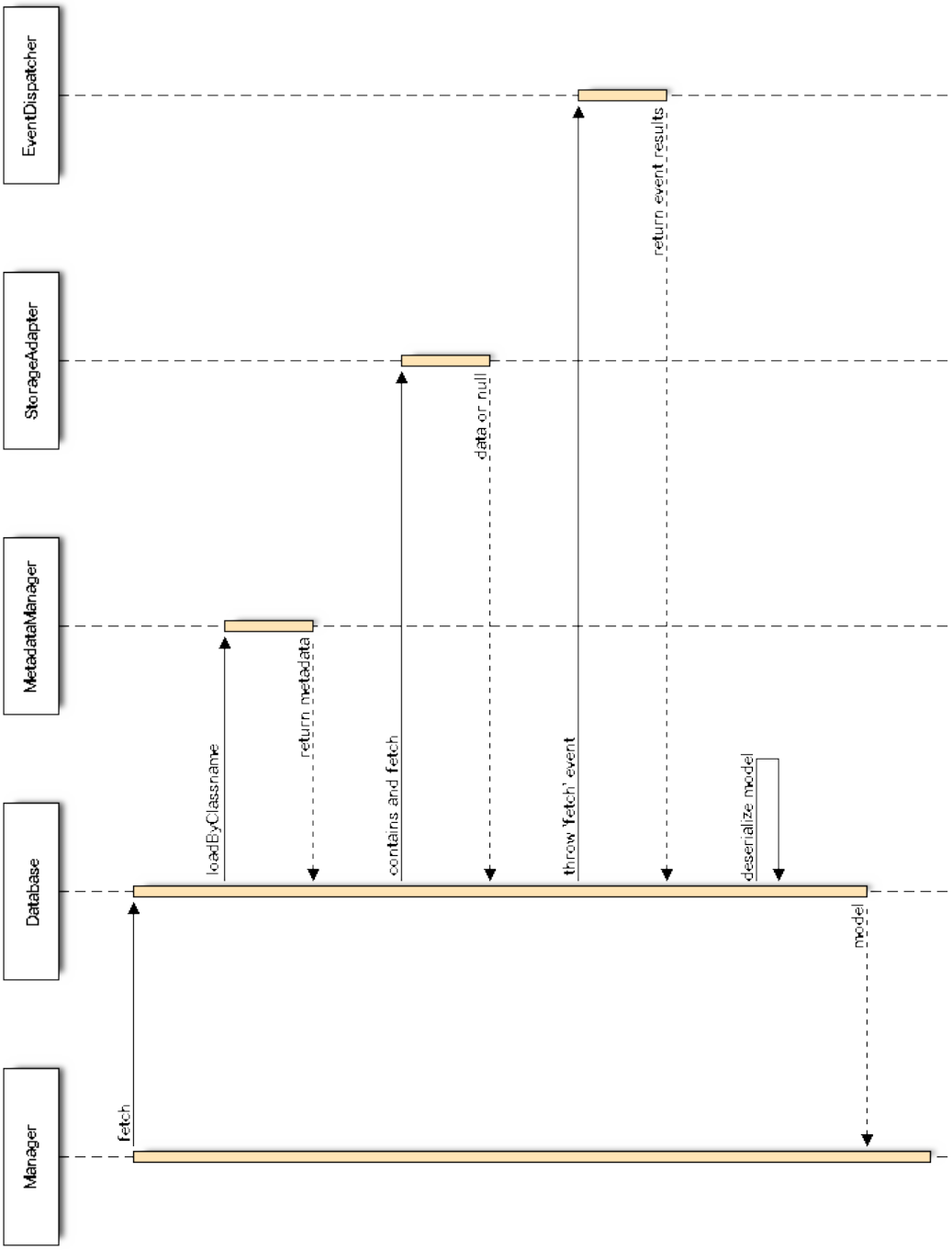

Abbildung 5.3: Objekte abrufen

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.

5.1.3 Replikator

Wie schon erwähnt, verwendet der Replikator 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 Typen (“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 primäry 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 den Daten persistiert wird und später wiederverwendet werden kann. Dieser Prozess wird in der Abbildung 5.4 visualisiert.

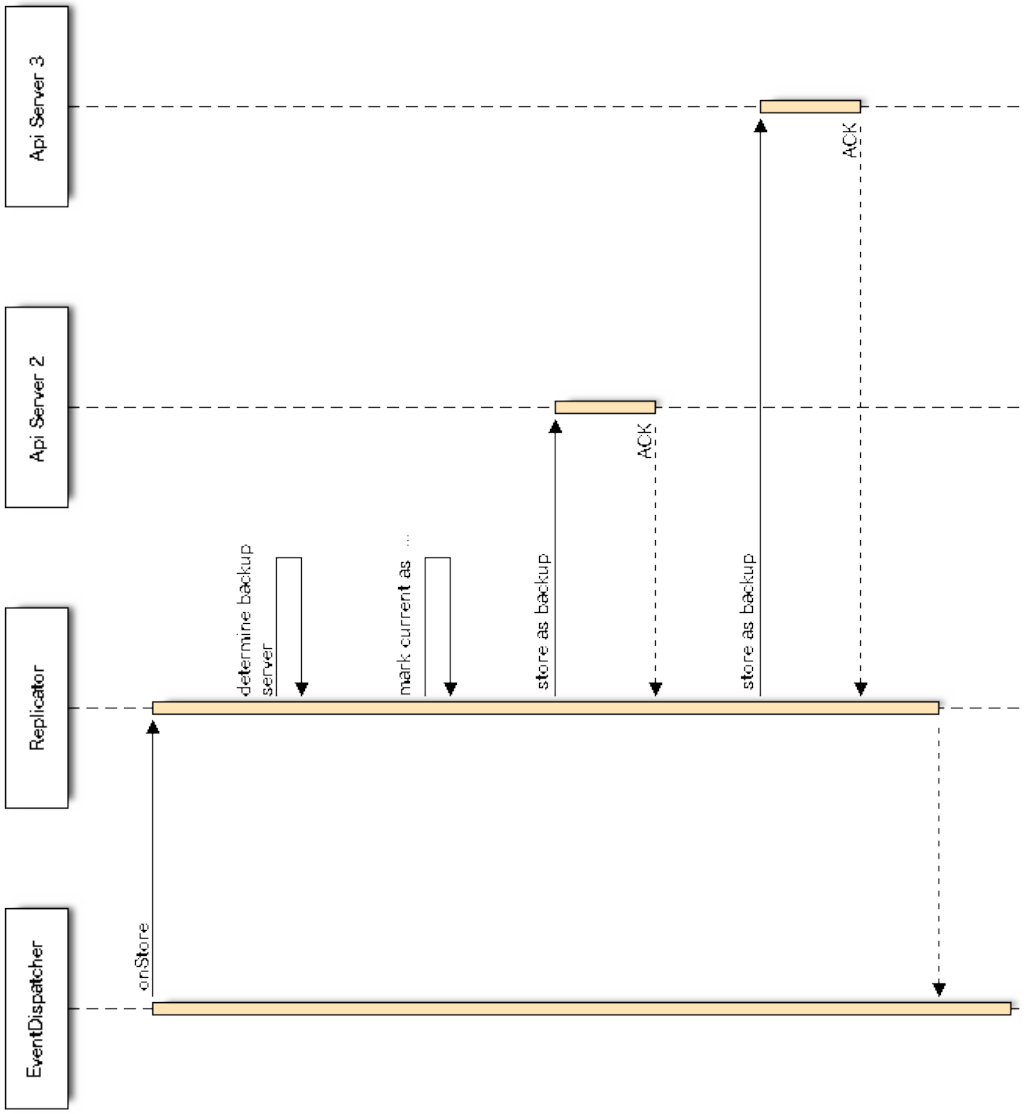

Abbildung 5.4: Replikationtyp “Full”

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-Status-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 Zugangsberechtigung, weil diese aber nicht gegeben ist, 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, aber der Benutzer ist 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.

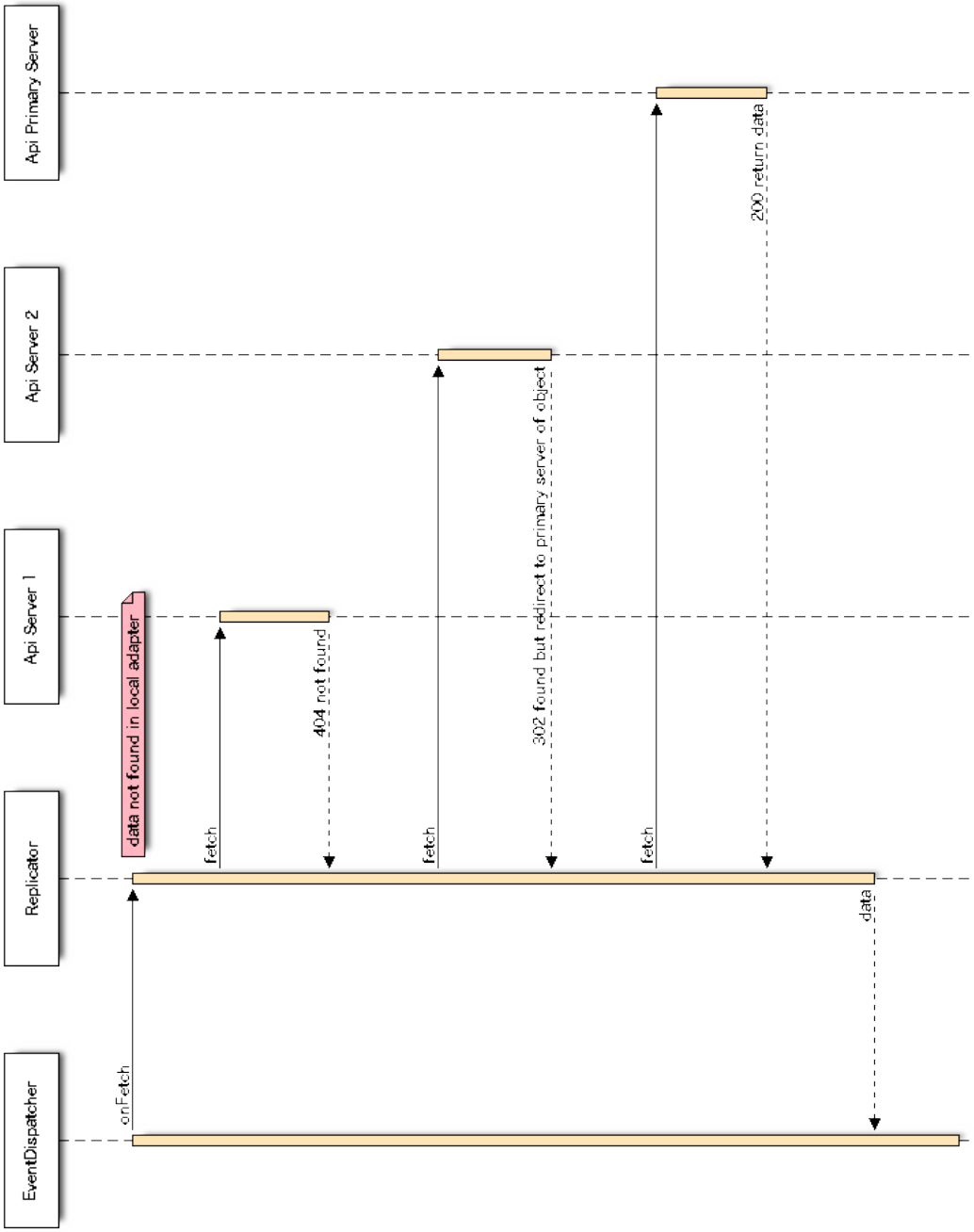

Abbildung 5.5: Replikator "Lazy"-Nachladen

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. Dieser Prozess kann daher bei einem großen Netzwerk sehr lange dauern. Dieser Fall sollte allerdings aufgrund des Datenmodells 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.

TODO Beispiel

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: `index`, `search` 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 Prototypen wurde die Bibliothek Zend-Search-Lucene[4] 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 Adapter implementiert den Befehlssatz: `fetch` und `store`.

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" bieten 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 Zusammenfassung

Die Bibliothek "Distributed-Storage" bietet eine einfache und effiziente Implementierung des in Kapitel 4 beschriebenen Konzeptes. Es baut auf eine erweiterbare Hash-Value

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

Datenbank auf. Diese Datenbank wird mittels eines Eventhandlers (Replikator) zu einer verteilten Datenbank. Dabei ist es für die Datenbank irrelevant, welcher Transportlayer 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 Kapitel 7.5 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[5] aufzubereiten. Für eine verteilte Installation implementiert die Plattform den "ApiAdapter", um die Rest-API für die Bibliothek, zu abstrahieren.

5.2.3 Benutzeroberfläche

Die Benutzeroberfläche ... **TODO was wurde/wird implementiert**

5.2.4 Zusammenfassung

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 späteren Zeitpunkt[6] diesen Prototypen 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

[5] http://restcookbook.com/
[6] Dieses Feature wurde in der Implementierung, die während dieser Arbeit entstanden ist, nicht umgesetzt.

Symfony Konsole-Komponente[7] umgesetzt. Diese Komponente ermöglicht eine schnelle und unkomplizierte Entwicklung solcher Konsolen-Programme.

Ausgeliefert wird das Programm in einem sogenannten PHAR-Container[8]. Dieser Container enthält alle benötigten Source-Code- und Konfigurationsdateien. Das Format ist vergleichbar mit dem JAVA-Container JAR. PHAR-Container werden in der PHP-Gemeinschaft oft verwendet, um komplexe Applikationen, wie zum Beispiel PHPUnit[9] (ein Test Framework für PHP) auszuliefern.

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 5.1: 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 5.1 ist die Ausführung des "Konfigurieren"-Kommandos dargestellt. Argumente können sowohl an den Befehl angehängt werden oder durch den Befehl abgefragt werden. Eine Validierung von 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

[7] http://symfony.com/doc/current/components/console/introduction.html
[8] http://php.net/manual/de/intro.phar.php
[9] https://phpunit.de/

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

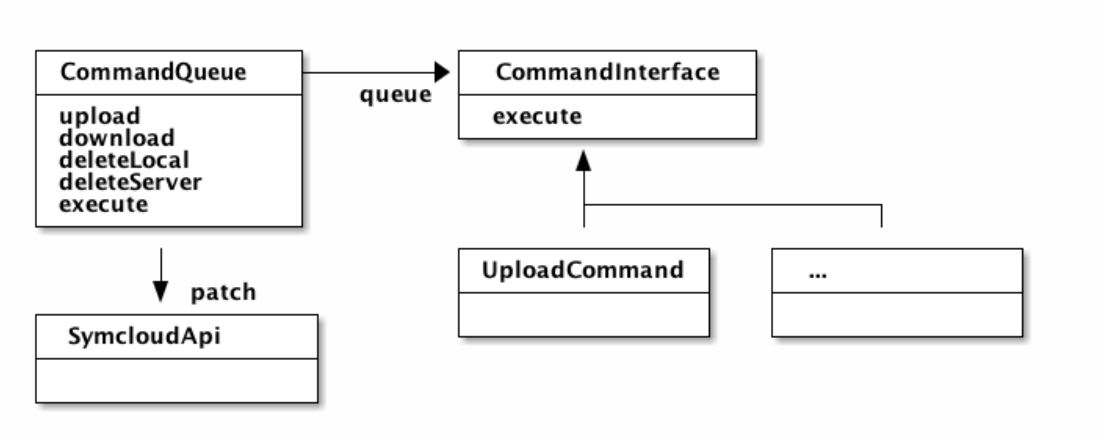

Abbildung 5.6: Architektur von Jibe

Der zentrale Bestandteil von Jibe ist eine “CommandQueue” (siehe Abbildung 5.6). Sie sammelt alle nötigen Kommandos ein und führt sie nacheinander aus. Diese “Queue” ist 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 an 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 können sehr einfach ermittelt werden. 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	Delete local	Delete server	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

Nicht angeführte Werte in der Tabelle, sind zu dem Zeitpunkt nicht verfügbar. Was zum Beispiel bedeutet, dass wenn der Lokale Hash nicht angeführt ist, die Datei nicht vorhaden 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.
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 im Kapitel 7.1 kurz behandelt.

5.3.3 Zusammenfassung

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 verschiedene 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

TODO Zusammenfassung Kapitel Implementierung

6 Ergebnisse und Ausblick

7 Ausblick

Welche Teile des Konzeptes konnten umgesetzt werden und wie gut funktionieren diese?

7.1 Konfliktbehandlung

7.2 Verteilung von Blobs

Besseres Verfahren als Zufall verwenden, das den freien Speicher als Grundlage für die Auswahl stellt. Eventuell könnte der Primary Server ebenfalls (zumindest für FULL - also Blobs) Aufgrund des freien Speicherplatzes ermittelt werden (falls der erstellende Server schon sehr viele Objekte besitzt oder wenig Speicherplatz besitzt).

TODO

- Was wird sonst verwendet?
- Literaturrecherche
- Papers

7.3 Konsistenz

Wenn ein Server nicht erreichbar ist, bedeutet das potenziell, dass dieser nicht mehr Konsistent ist. Dieser sollte keine Changes mehr annehmen. Sobald er wieder Online ist,

muss er bei allen Servern den OPLog abholen und diesen ausführen.

Dieser OPlog beinhaltet alle Operationen die ausgeführt werden. Genauer beschrieben hier: http://docs.mongodb.org/manual/core/replica-set-oplog/

7.4 Datei chunking

Theoretisch ist es möglich, dass Dateien, nach bestimmten Chunks durchsucht werden, die bereits im Storagesystem abgelegt sind. Dazu könnte ein ähnliches Verfahren wie bei rsync verwendet werden (Rolling-Checksum-Algorithm).

7.5 Lock-Mechanismen

TODO kurze Beschreibung und Ansätze

Anhang

Amazon S3 System-spezifische Metadaten

Tabelle 7.1: Objekt Metadaten [„Object Key and Metadata“ 2015]

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,

Name	Description
	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-aws-kms-key-id	If the x-amz-server-side-encryption is present and has the value of aws:kms, this indicates 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.

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 2015].

Der Benutzer 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 2015].

Begriffe

In OAuth2 werden folgende vier Rollen definiert:

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

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

Authorization server Der Server, der die Zugriffs-Tokens nach der erfolgreichen Authentifizierung des Ressourceneigentümers bereitstellt [Hardt 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 den Benutzer bestätigt wurde [Hardt 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 [Hardt 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 [Hardt 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 [Hardt 2012, S. 5].

Protokoll Ablauf

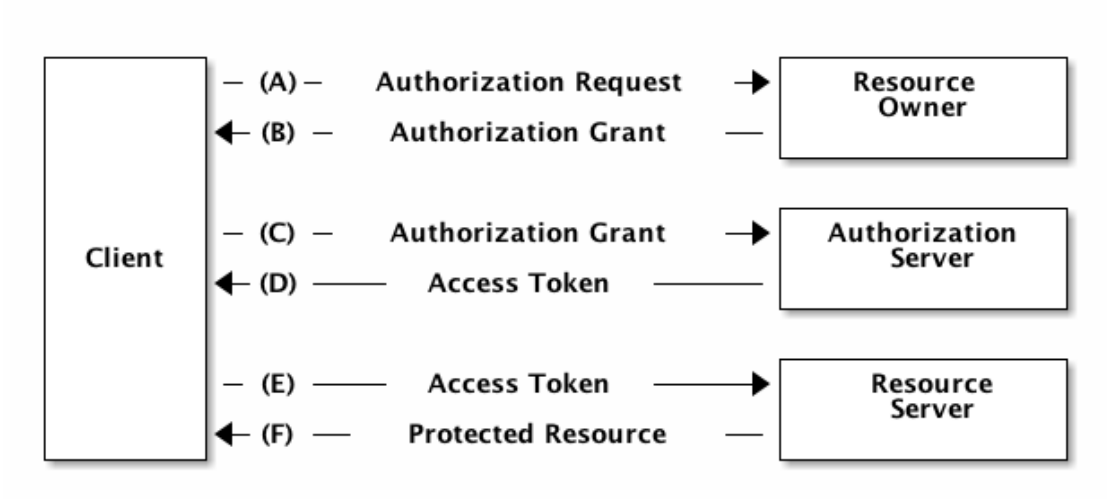

Abbildung 7.1: Ablaufdiagramm des OAuth [Hardt 2012, S. 7]

Der Ablauf einer Autorisierung [Hardt 2012, S. 7 ff] mittels Oauth2, der in der Abbildung 7.1 abgebildet ist, enthält folgende Schritte:

A) Der Client fordert die Genehmigung des Ressourcenbesitzers. Diese Anfrage kann direkt an den Benutzer 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.

Zusammenfassung

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

Installation

Dieses Kapitel enthält eine kurze Dokumentation wie Symcloud (inklusive JIBE) installiert und deployed werden kann. Es umfasst eine einfache Methode auf einem System und ein verteiltes Setup.

Lokal

Verteilt

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

„Amazon S3“ [2015]: Amazon S3. . Online im Internet: http://aws.amazon.com/de/s3/

„Amazon Web Services: We'll go to court to fight gov't requests for data | ITworld“ [o. J.]: Amazon Web Services: We'll go to court to fight gov't requests for data | ITworld. http://www.itworld.com/article/2705826/cloud-computing/amazon-web-services–we-ll-go-to-court-to-fight-gov-t-requests-for-data.html.

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 http://blog.codinghorror.com/the-xanadu-dream/.

Basho Technologies, Inc. [2015]: Riak CS http://docs.basho.com/riakcs/latest/.

Birrell, Andrew ; Needham, Roger [1980]: „A Universal File Server“ In: IEEE Transactions on Software Engineering, 6 [1980], 5, S. 450–453. Online im Internet: https://birrell.org/andrew/papers/UniversalFileServer.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 http://schacon.github.io/gitbook/1$_{\text{t}}$he$_{\text{g}}$it$_{\text{o}}$bject$_{\text{m}}$odel.

„Cloud-Dienste für Startups: „Automatisierung ist Pflicht“ [Interview] | t3n“ [o. J.]: Cloud-Dienste für Startups: „Automatisierung ist Pflicht“ [Interview] | t3n. http://t3n.de/news/cloud-dienste-startups-amazon-web-services-486480/.

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

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

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/

„Federation protocol overview“ [2015]: Federation protocol overview. . Online im Internet: https://wiki.diasporafoundation.org/Federation_protocol_overview [Zugriff am: 26.05.2015].

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

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

Hammer, Eran [2015]: OAuth Core 1.0. . Online im Internet: http://hueniverse.com/oauth/guide/history/ [Zugriff am: 15.05.2010].

Hardt, Dick [2012]: „The OAuth 2.0 authorization framework“In: https://tools.ietf.org/html/rfc6749. [2012]

„Introduction to Amazon S3“ [2015]: Introduction to Amazon S3. . Online im Internet: http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html

Jones, P. [2013]: „WebFinger“In: https://tools.ietf.org/html/rfc7033. [2013]

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

Nelson, T. [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 Proceedings of the Eighteenth Conference on Hypertext and Hypermedia New York, NY, USA: ACM, S. 227–228. Online im Internet: http://doi.acm.org/10.1145/1286240.1286303

„Object Key and Metadata“ [2015]: Object Key and Metadata. . Online im Internet: http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingMetadata.html

„Open Source Private Cloud Software | AWS-Compatible | HP Helion Eucalyptus“ [o. J.]: Open Source Private Cloud Software | AWS-Compatible | HP Helion Eucalyptus. https://www.eucalyptus.com/eucalyptus-cloud/iaas.

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].

Schütte, Prof. Dr. Alois [o. J.]: Verteilte Dateisysteme http://www.fbi.h-da.de/ a.schuette/Vorlesungen/VerteilteSysteme/Skript/6_VerteilteDateisysteme/VerteilteDateisysteme.pdf.

Seidel, Udo [2013]: Dateisystem-Ueberblick Linux Magazin .

„Server2Server - Sharing“ [2015]: Server2Server - Sharing. . Online im Internet: https://www.bitblokes.de/2014/07/server-2-server-sharing-mit-der-owncloud-7-schritt-fuer-schritt [Zugriff am: 27.03.2015].

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

„The Wizbit Open Source Project on Open Hub“ [o. J.]: The Wizbit Open Source Project on Open Hub. https://www.openhub.net/p/wizbit.

„Under the Hood: File Replication“ [o. J.]: Under the Hood: File Replication. http://xtreemfs.org/how_replication_works.php.

„Using Versioning“ [2015]: Using Versioning. . Online im Internet: http://docs.aws.amazon.com/AmazonS3/latest/dev/Versioning.html

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

„Wizbit: a Linux filesystem with distributed version control | Ars Technica“ [2008]: Wizbit: a Linux filesystem with distributed version control | Ars

Technica. http://arstechnica.com/information-technology/2008/10/wizbit-a-linux-filesystem-with-distributed-version-control/.

„XtreemFS - architecture, internals and developer's documentation“ [o. J.]: XtreemFS - architecture, internals and developer's documentation. http://www.xtreemfs.org/arch.php.

„XtreemFS Installation and User Guide“ [o. J.]: „XtreemFS Installation and User Guide“In: http://www.xtreemfs.org/xtfs-guide-1.5/index.html.