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

Wachter Johannes

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Seit den Abhörskandalen durch die NSA und andere Geheimdienste ist es immer mehr Menschen wichtig, die Kontrolle über die eigenen Daten zu behalten. Aufgrund dessen erregen Projekte wie Diaspora[1], ownCloud[2] und ähnliche Softwarelösungen immer mehr Aufmerksamkeit. Die beiden genannten Softwarelösungen decken zwei sehr wichtige Bereiche der persönlichen Datenkontrolle ab.

Diaspora ist ein dezentrales soziales Netzwerk. Die Benutzer von diesem Netzwerk sind durch die verteilte Infrastruktur nicht von einem Betreiber abhängig. Es ermöglicht, seinen Freunden bzw. der Familie, eine private social-media Plattform

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

anzubieten und diese nach seinen Wünschen zu gestalten. Das Interessante daran sind die sogenannten Pods (dezentrale Knoten), die sich beliebig untereinander vernetzen lassen. Damit baut Diaspora ein privates P2P Netzwerk auf. Pods können von jedem installiert und betrieben werden; dabei kann der Betreiber bestimmen, wer in sein Netzwerk eintreten darf und welche Server mit seinem verbunden sind. Die verbundenen Pods tauschen ohne einen zentralen Knoten, Daten aus und sind dadurch unabhängig. Dies garantiert die volle Kontrolle über seine Daten im Netzwerk [„Was ist Dezentralisierung“ 2015].

Das Projekt “ownCloud” ist eine Software, die es ermöglicht, Daten in einer privaten Cloud zu verwalten. Mittels Endgeräte-Clients können die Daten synchronisiert und über die Plattform auch geteilt werden. Insgesamt bietet die Software einen ähnlichen Funktionsumfang gängiger kommerzieller Lösungen an [„Owncloud Features“ 2015]. 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[3].

1.1 Projektbeschreibung

Symcloud ist eine private Cloud-Software, die es ermöglicht, über dezentrale Knoten (ähnlich wie Diaspora) Daten 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.

TODO genauere Beschreibung

[3] http://webodf.org/

1.2 Inspiration

TODO Noch einmal ownCloud - Diaspora und Ted Nelson mit dem Xanadu Projekt

1.3 Anforderungen

TODO Anforderungen an das Projekt (auch in Bezug auf xanadu)

- Sicherheit
- Datenschutz
- Effizienz
- Verteilbarkeit
- Zugriffberechtigungen
- Versionierung
- Shares

TODO genauere Ausformulierung

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. Sie 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 Dropbox

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

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].

In der Abbildung 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

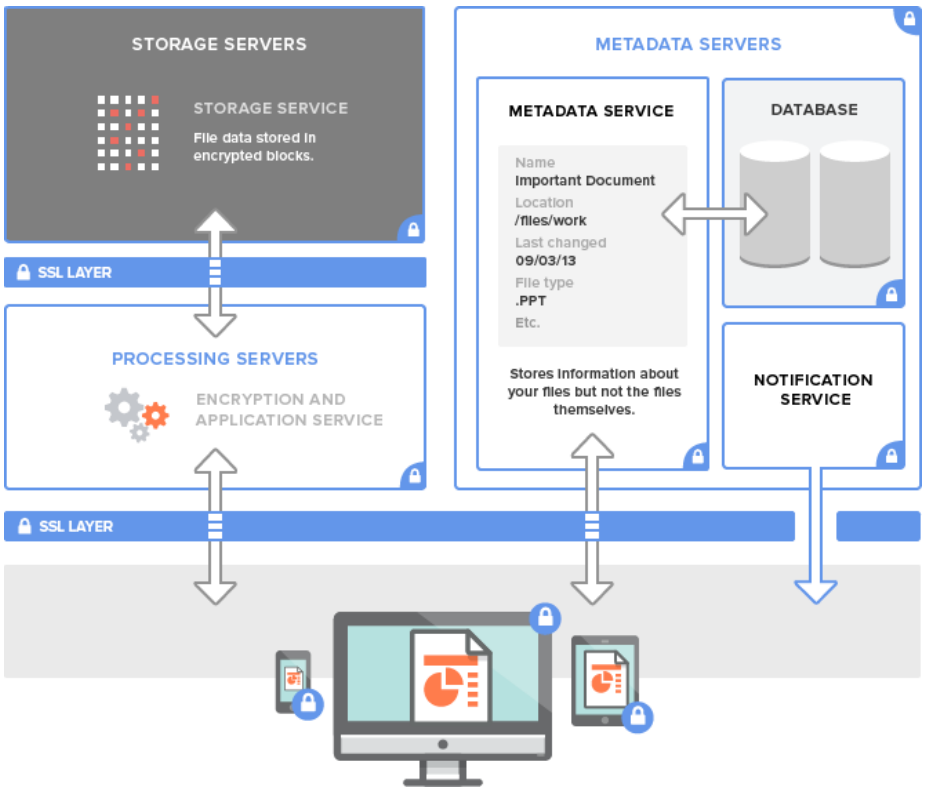

Abbildung 1: Blockdiagramm der Dropbox Services [„Wie funktioniert der Dropbox-Service“ 2015]

auch Dritt-Hersteller Anwendungen zu entwickeln, die mit Daten aus der Dropbox arbeiten. Für diesen Zweck gibt es im Authentifizierungsprotokoll OAuth2 sogenannte Scopes (siehe Kapitel 5.3). Es ermöglicht Anwendungen den Zugriff Teilbereiche der API zu autorisieren. Eine weitere Aufgabe, die diese Schicht erledigt, ist die Verschlüsselung der Anwendungsdaten [„Wie funktioniert der Dropbox-Service“ 2015].

2.3 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 ist abgebildet, wie ownCloud als eine Art Verbindungsschicht zwischen verschiedenen Lokalen- und Cloud-Speichersystemen dienen soll [ownCloud 2015, p. 1].

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 2015, p. 2]

Für einen produktiven Einsatz wird eine skalierbare Architektur, wie in Abbildung 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

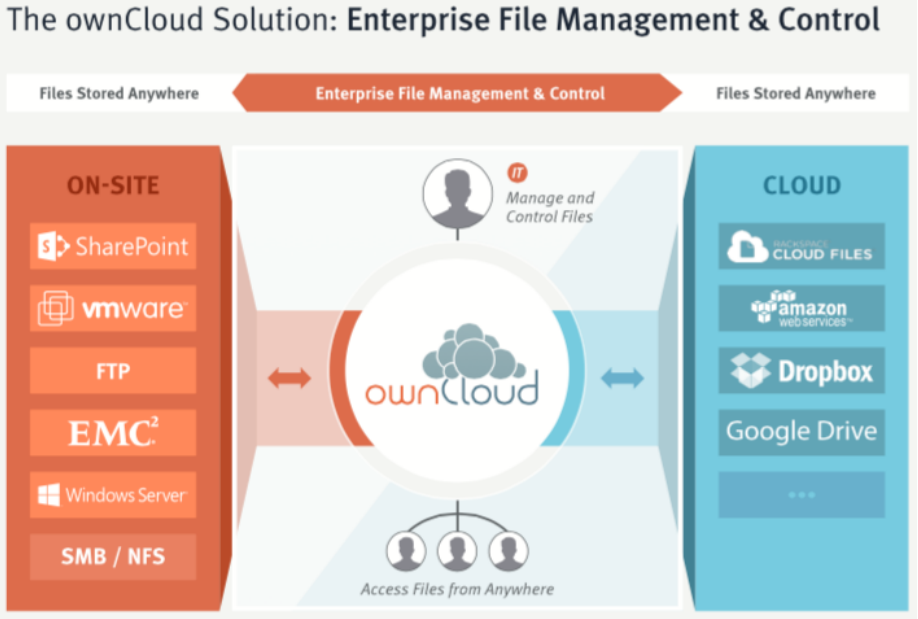

Abbildung 2: ownCloud Enterprise Architektur Übersicht [ownCloud 2015]

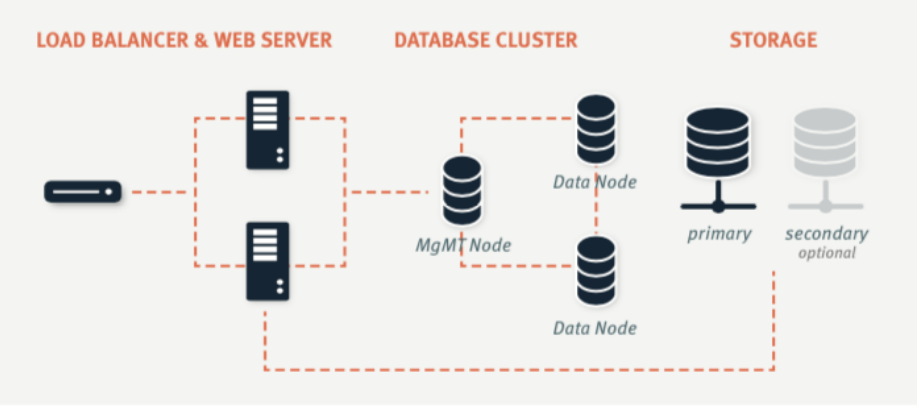

Abbildung 3: Bereitstellungsszenario von ownCloud [ownCloud 2015]

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 2015, p. 3-4]

2.4 Diaspora

Diaspora verwendet für die Kommunikation zwischen den Servern (Pods) ein eigenes Protokoll namens "Federation protocol". 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 werden 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] aus und erhält einen Link zum LRDD ("Link-based Resource Descriptor Document"[5]).

```
<Link rel="lrdd"
      template="https://bob.diaspora.example.com/?q={uri}"
      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:

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

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

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

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

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

2.5 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 betrachtet, die ähnliche Funktionen aufzuweisen haben, wie Symcloud.

Dropbox Kommerzielles Produkt mit dem gewünschten Funktionsumfang hinsichtlich der Dateisynchronisierung und Benutzerinteraktion.

ownCloud Open-Source Alternative zu Dropbox. Aufgrund aktueller Entwicklungen besitzt es zusätzliche Funktionen, die in Symcloud ebenfalls in den Anforderungen beschrieben wurden.

Diaspora Ein verteiltes Social-Media, welches einen ähnlichen Aufbau besitzt wie Symcloud.

Diese drei Systeme bieten, jeder für sich, Ansätze die für die Implementierung von Symcloud relevant sind.

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, Verlinkungen und Metadaten direkt in der Speicherlösung abzulegen. Dies erleichtert die Implementierung der eigentlichen Anwendung.

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

Performance 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, wie zum Beispiel Amazon S3[6], 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 fungieren als einfache Laufwerke und abstrahieren dadurch den komplexen Ablauf der darunter liegenden Services. Der Zugriff

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

auf diese Dateisysteme erfolgt meist über system-calls wie zum Beispiel `fopen` oder `fclose`. Dies ergibt sich aus der Transparenz Anforderung [Coulouris ; Dollimore ; Kindberg 2003, S. 369], die im Kapitel 3.3.1 beschrieben wird.

Datenbank gestützte Dateisysteme, wie zum Beispiel GridFS[7] von MondoDB, erweitern Datenbanken, um große Dateien effizient und sicher 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

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

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 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, in jedem Bucket aktiviert werden.

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

Abbildung 4: 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[8], 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.

[8] Amazons Zuständiger für die Zusammenarbeit zwischen den Partner

3.2.4 Alternativen zu Amazon S3

Es gibt einige Amazon S3 kompatible Anbieter, die einen ähnlichen Dienst 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.

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 Erstellung 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 5) kann die feste Transaktionszeit von ca. 140ms abgelesen werden.

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

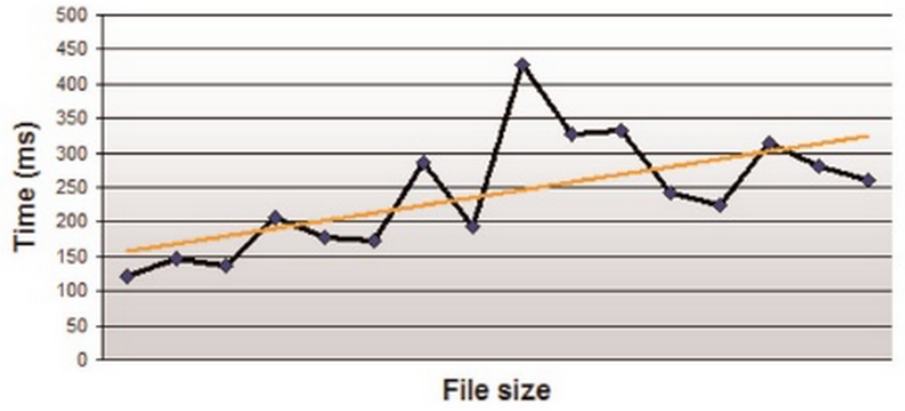

Abbildung 5: Upload Analyse zwischen EC2 und S3 [„Amazon S3 and EC2 Performance Report – How fast is S3“ 2009]

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 Ser-

ver verteilt, müssen die Operationen, an alle Server weitergeleitet werden. Die Verzögerung, die dabei auftritt, führt in dieser Zeit zu einem inkonsistenten Zustand des Systems [Coulouris ; Dollimore ; Kindberg 2003, S. 369ff].

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

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

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

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

Aufgrund der Tatsache, dass Festplatten heutzutage nicht mehr so teuer 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, dass von einem entfernten Server verwaltet wird. Dies ist vergleichbar mit RPC[9]. Der Client erhält den Zugriff auf eine Schnittstelle, um auf Dateien zuzugreifen, die ein entfernter Server implementiert [Tanenbaum ; Steen 2003, S. 647ff].

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 Datei-

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

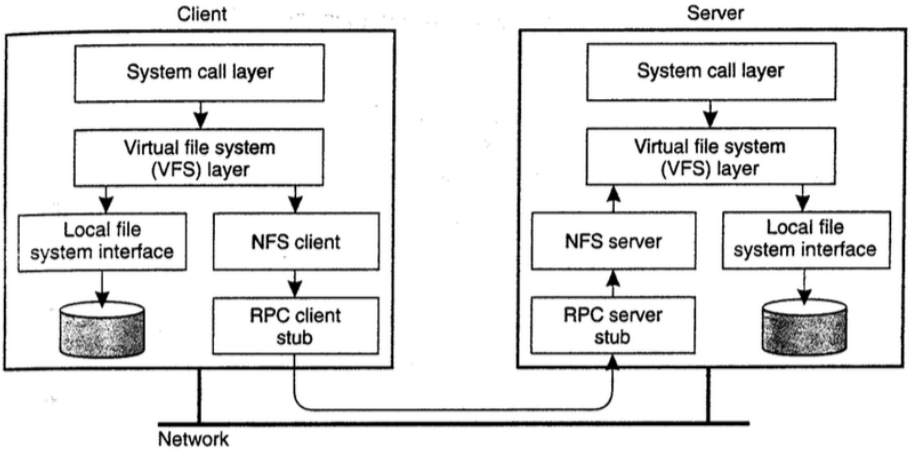

Abbildung 6: NFS Architektur[Tanenbaum ; Steen 2003, S. 647]

baum, ob die Operation an das lokale Dateisystem oder an den NFS-Client weitergegeben wird (siehe Abbildung 6). Der NFS-Client ist eine separate Komponente, die sich um den Zugriff auf entfernte Dateien kümmert. Dabei fungiert der Client als eine Art Stub-Implementierung der Schnittstelle 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].

Replication Die drei Hauptkomponenten von XtreemFS, Directory Service, Metadata Catalog und die Object Storage Devices (siehe Kapitel 7), können repliziert redundant verwendet werden, dies führt zu einem fehlertoleranten System. Die Replikationen zwischen diesen Systemen erfolgt 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 und Anzahl der Server 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[10] [„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 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:

DIR - Directory Service ist das zentrale Register für alle Services. Andere Services bzw. Client verwenden ihn um zum Beispiel alle Object-Storage-Devices zu finden. [„XtreemFS Installation and User Guide“ o. J., S. 2.4]

MRC - Metadata and Replica Catalog verwaltet die Metadaten der Datei, wie zum Beispiel Dateiname, Dateigröße oder Bearbeitungsdatum. 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 - 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]

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

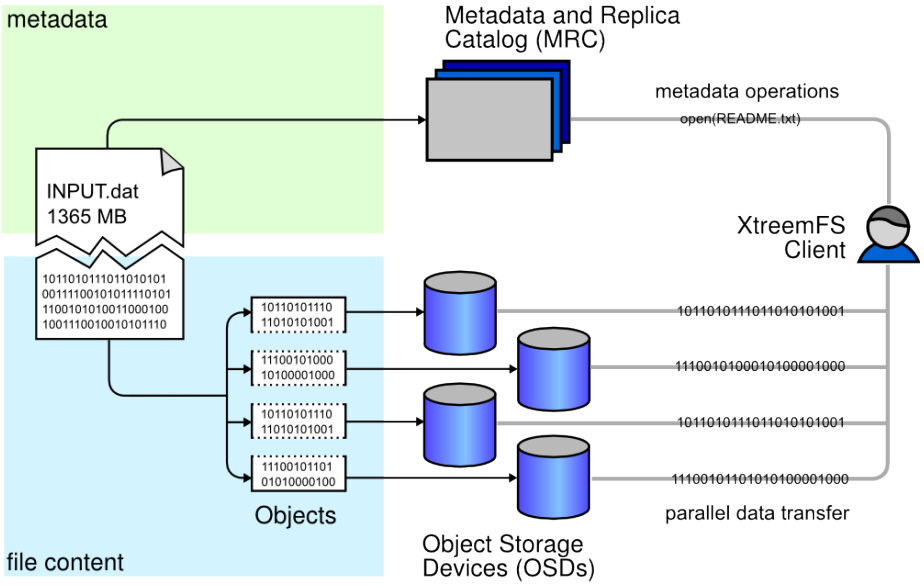

Abbildung 7: XtreemFS Architektur [„XtreemFS - architecture, internals and developer's documentation“ o. J.]

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 können 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-Schreiben

Es gibt auch hier zwei Arten zur Implementierung des Protokolls. Das eine ist ein nicht replizierendes Protokoll, bei dem alle Schreib- und Lesezugriffe auf

den 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].

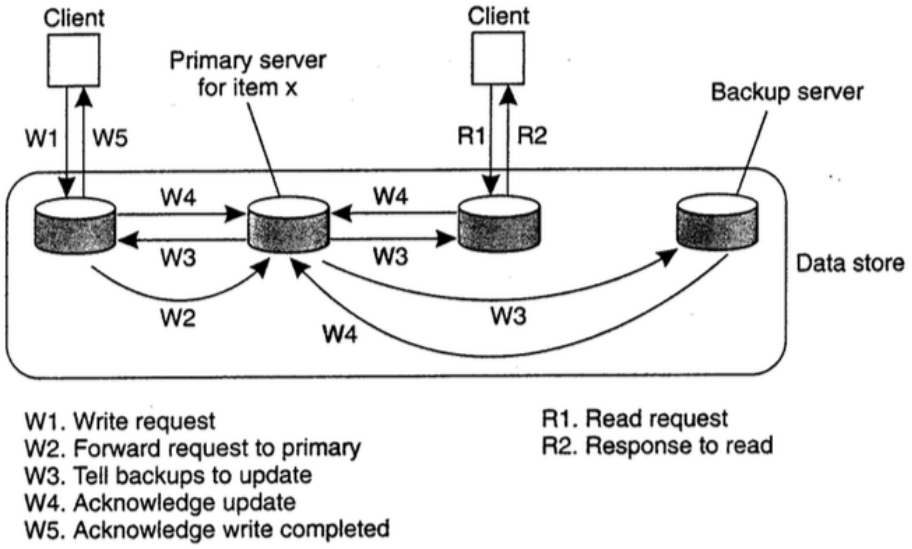

Abbildung 8: Primary-Backup-Protokoll: Entferntes-Schreiben[Tanenbaum ; Steen 2003, S. 385]

Der Prozess, der eine Schreiboperation (siehe 8) auf das Objekt ausführen will, gibt sie an den primären Server weiter. Dieser führt die Operation lokal an dem Objekt aus und gibt die Aktualisierungen an die Backup-Server weiter. Jeder dieser Server führt die Operation aus und gibt eine Bestätigung an den primären Server 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 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 in dieser Implementierung gibt es zwei Möglichkeiten es zu implementieren. 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 9), bei dem der primäre Server des Objektes zu dem ausführenden Server migriert wird [Tanenbaum ; Steen 2003, S. 386ff].

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

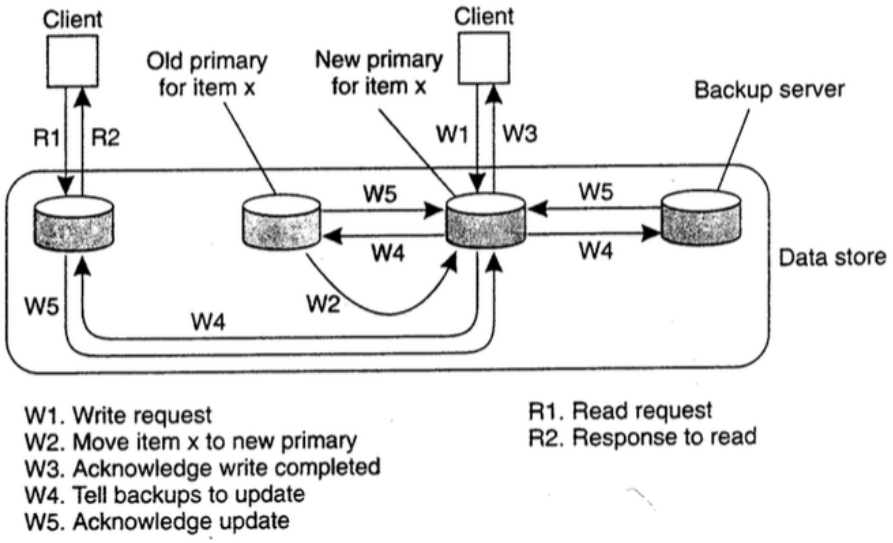

Abbildung 9: Primary-Backup-Protokoll: Lokales-Schreiben[Tanenbaum ; Steen 2003, S. 387]

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[11], 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 müsste dann die Anwendungen für jeden Anwendungsbenutzer einen Systembenutzer anlegen [„XtreemFS

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

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 Konsistenzprotokollen von XtreemFS gezogen werden und in ein Gesamtkonzept miteingebunden werden.

3.4 Datenbank gestützte Dateiverwaltungen

Einige Datenbanksysteme, wie zum Beispiel MongoDB[12], 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.

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.

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

Beispiel:

```
$mongo = new Mongo();                    // connect to database
$database = $mongo->selectDB("example"); // select mongo database

$gridFS = $database->getGridFS();        // use GridFS class for
                                         //handling files

$name = $_FILES['Filedata']['name'];     // optional - capture the
                                         // name of the uploaded file
$id = $gridFS->storeUpload('Filedata', $name);
                                         // load file into MongoDB
```

Bei der Verwendung von MongoDB ist es sehr einfach, Dateien in GridFS 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.

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 ist 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 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 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 **?? TODO evtl. Nummer anpassen**) 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 Konzept für 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 10 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 “StorageAdaper” bzw. über den

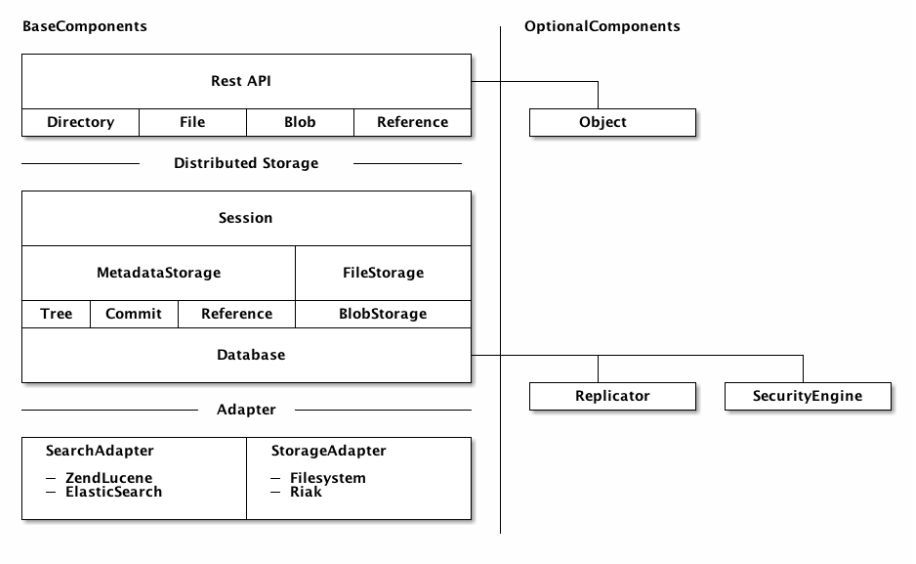

Abbildung 10: Architektur für “Symcloud-Distributed-Storage”

“SearchAdapter”, lassen sich die Speichermedien der Daten anpassen. Für eine einfache Installation reicht es die Daten direkt auf die Festplatte zu schreiben. Es ist allerdings auch denkbar die Daten in eine verteilte Datenbank wie Riak oder MongoDB zu schreiben, um die Datensicherheit zu erhöhen.

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

4.2 Datenmodell

Das Datenmodell wurde speziell für Symcloud entwickelt, um seine Anforderungen zu erfüllen. Dabei wurde großen Wert darauf gelegt, optimale und effiziente Datenhaltung zu gewährleisten. Abgeleitet wurde das Model (siehe Abbildung 11) aus dem Model, dass dem Versionskontrollsystem GIT zugrunde liegt. Dieses

Model unterstützt viele Anforderungen, welche Symcloud an seine Daten stellt.

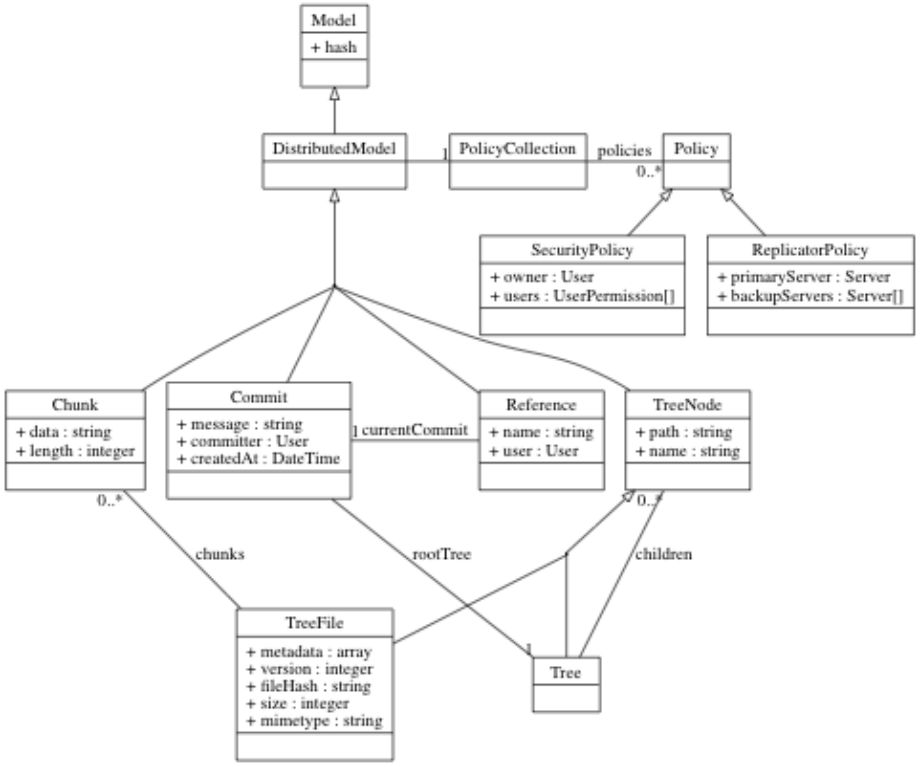

Abbildung 11: Datenmodel für “Symcloud-DistributedStorage”

4.2.1 Exkurs: GIT

GIT[13] ist ein verteilte Versionsverwaltung, das 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 dem 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

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

Abbildung 12: GIT-Logo

SHA ist ein mittels “Secure-Hash-Algorithm” berechneter Hashwert der Daten [Chacon 2009, S. 9.2]. Das Listing **??** zeigt, wie ein SHA in einem Terminal berechnet werden kann [Keepers 2012].

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

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

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

Die Objekte in GIT sind immutable, also nicht veränderbar. 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:

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].

Der TREE beschreibt ein Ordner im Repository. Ein TREE enthält andere TREE bzw. BLOB Objekte und definiert damit eine Ordnerstruktur. In einem TREE werden auch die Namen zu BLOB und TREE Objekten festgelegt [Chacon 2009, S. 9.2].

Der COMMIT ist ein Zeitstempel eines einzelnen TREE Objektes. Im folgenden Listing **??** wird der Inhalt eines COMMIT Objektes auf einem Terminal ausgegeben [Chacon 2009, S. 9.2].

```
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
```

Das Objekt enthält folgende Werte:

Tabelle 1: GIT commit Eigenschaften [Chacon 2009, S. 9.2]

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

Anmerkungen (zu der Tabelle 1):

- Ein COMMIT kann mehrere Vorgänger haben, wen sie zusammengeführt werden. Zum Beispiel würde dieser Mechanismus bei einem MERGE verwendet werden, um die beiden Vorgänger zu speichern.
- Der 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 einen bestimmte COMMIT Objekt. Diese Referenzen sind die Grundlage for das Branching-Model von GIT [Chacon 2015].

Diese Zusammenhänge werden in der Abbildung 13 visualisiert.

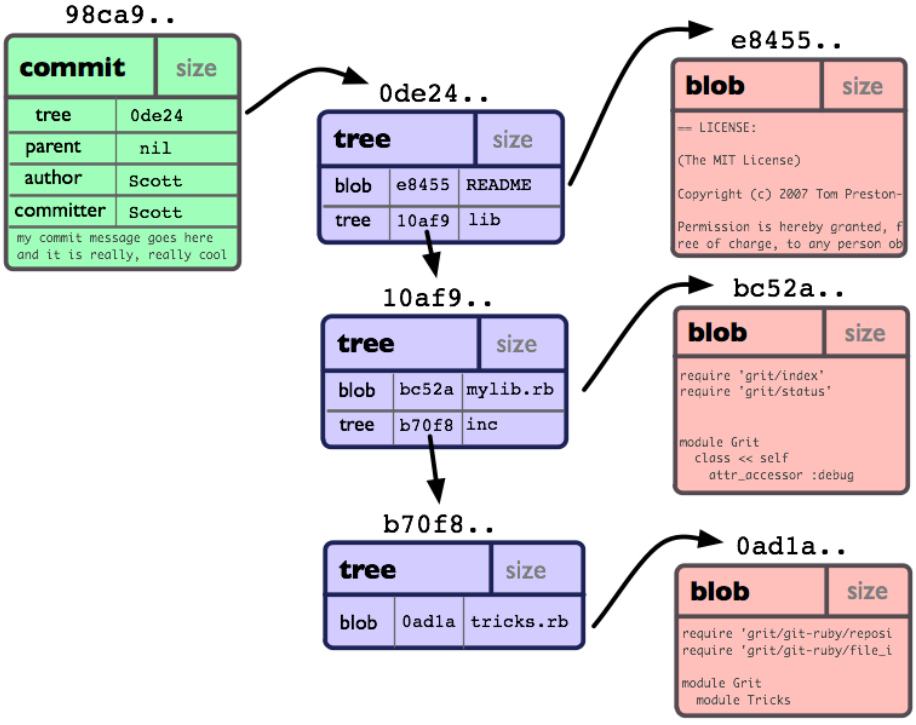

Abbildung 13: Beispiel eines Repositories [Chacon 2015]

Anforderungen Das Datenmodell von GIT erfüllt folgende Anforderungen von Symcloud:

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

Symlinks Ebenfalls mit den Referenzen, können sogenannte Symlinks erstellt werden. Diese Symlinks werden im System verwendet, um Shares an einer bestimmten Stelle des Dateibaums eines Benutzers zu platzieren[14].

Zusammenfassung Das Datenmodell von GIT ist aufgrund seiner Flexibilität eine gute Grundlage für ein Verteiltes Dateisystem. 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,

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

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) Dateien werden nicht komplett in einen chunk geschrieben sondern werden 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[15].

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).

Symlinks Die dritte Erweiterung ist die Verbindung zwischen Tree und Referenz. Diese Verbindung verwendet Symcloud um Symlinks (zu Referenzen) in einem Dateibaum zu modellieren und dadurch die Einbettung von Shares in den Dateibaum zu ermöglichen[16]. Diese Verbindung ist unabhängig

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

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

von dem aktuellen Commit der Referenz und dadurch ist die gemeinsame Verwendung der Dateien zwischen den Benutzern einfach umzusetzen.

Policies Die Policies oder Strategien 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, der mithilfe des "Replicators" zu einer verteilen Datenbank erweitert wird. Die Datenbank serialisiert die Objekte und speichert sie mithilfe eines Adapters auf einem bestimmten Speichermedium. Dieses Speichermedium kann mithilfe des Adapters verschiedene Ziele besitzen. 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 erzeugt wurde. Aus einem Pool an Servern werden die Backup-Server ermittelt. Dabei gibt es drei Arten diese zu ermitteln.

Full Die Backup-Server 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. Backup-Server zu ermitteln (siehe Kapitel 7.2).

Permissions Wen 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 Backup-Server. Dieses Verfahren, wird verwendet für kleinere Objekte, die zum Beispiel Datei- bzw. Ordnerstrukturen enthalten. Die Verteilung kann sofort ausgeführt werden oder die Objekte werden "Lazy" beim ersten Zugriff jedes Servers 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 nach die neuesten Daten verwendet, bevor er Änderungen durchführt. Wichtig ist bei diesem Verfahren, dass Änderungen der Zugriffsrechte Automatisch zu einer Änderung der Referenz führen, damit die Backup-Server diese Änderung mitbekommen. Um die Datensicherheit für diese Objekte zu erhöhen könnten aus dem Serverpool eine konfigurierbare Anzahl von Backup-Servern, wie bei dem Full Typen, ausgewählt werden. Allerdings müsste der Pool auf die Zugriffsberechtigten Server beschränkt werden.

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. Was 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 Backup-Server und nicht das komplette Netzwerk. Stubs können wie auch der vorherige Typ automatisch verteilt werden oder “Lazy” bei der ersten Verwendung nachgeladen werden.

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. Dies unterstützt die implementierte Datenbank dadurch, dass diese Objekte auf keinem Server gecached werden und die Backup-Server automatische Updates zu Änderungen erhalten.

Diese Objekte werden im Netzwerk mit unterschiedlichen Typen verteilt. Die Strukturdaten (Tree und Commit) verwendet den Typ “Permission”. Was 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 dann 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. Was 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 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. Wen 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, das mehrere Dateien den selben chunk teilen. Häufiger jedoch geschieht dies, wenn zum 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, besteht die neue Version 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[17] (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 wider aus dem Speicher gelöscht werden, bevor der nächste chunk aus der Datenbank geladen wird. Dies beschleunigt die Zeit um eine Antwort zu erzeugen. Moderne Video-Player machen sich dieses Verfahren zu Nutzen 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

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

berechnen und genau diese aus dem Storage laden [Fielding 2014].

Im Filestorage werden zwei Arten von Objekten beschrieben. Zum einen sind dies die abstrakten Dateien, die nicht direkt in die Datenbank geschrieben werden, sondern primär der Kommunikation dienen und in den Dateibaum eingebettet werden können. Zum anderen sind es die konkreten chunks die direkt in die Datenbank geschrieben werden. Um diese 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 Remote-Servern 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 und diese 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 Sulu 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 mit SULU 2 passiert ist. Die OAuth2 Schnittstelle ermöglicht es auch externe Applikationen mit Daten aus Symcloud zu versorgen.

Die Rest-API ist in drei 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 geändert und diese gesammelt versioniert werden.

Optional ist die Schnittstelle für die Datenbank-Objekte. Diese Schnittstelle verwendet der Replikator um die Objekte zwischen den Servern zu verteilen. Dabei

werden die HTTP-Befehle GET und POST verwendet um Daten anzufragen 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 die Verteilung der Daten innerhalb eines internen Netzwerkes auf. Dies ermöglicht eine Effiziente und Sichere Datenverwaltung. Allerdings kann die Software auch alleinstehend 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 Replikations-Typen lassen sich verschiedene Objekte auf verschiedenste weise im Netzwerk verteilen. Die einzelnen Server sind durch eine definierte Rest-API getrennt 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:

Die Bibliothek (distributed-storage) ist der Kern der Applikation und im-

plementiert große Teile des Konzeptes von Symcloud. Sie baut auf modernen Web-Technologien auf und verwendet einige Komponenten des PHP-Frameworks Symfony2[18]. Dieses Framework ist eines der beliebtesten Frameworks in der Open-Source Community von PHP.

Die Plattform (symcloud) bietet neben der REST-API auch ein einfaches UI an, mit dem es möglich ist, im Browser sein Dateien zu verwalten. Als Basis verwendet Symcloud die Content-Management-Plattform SULU[19] der Vorarlberger Firma MASSIVE ART WebServices[20] aus Dornbirn. Diese Plattform bietet ein erweiterbares Admin-UI, eine Benutzerverwaltung und ein Rechtesystem. Diese Features ermöglichen Symcloud eine schnelle Entwicklung der Oberfläche und deren zugrundeliegenden Services.

Der Client (jibe) 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

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

Der interne Aufbau der Bibliothek ist in vier Schichten (siehe Abbildung 14) aufgeteilt.

[18] http://symfony.com/
[19] http://www.sulu.io
[20] http://www.massiveart.com/de

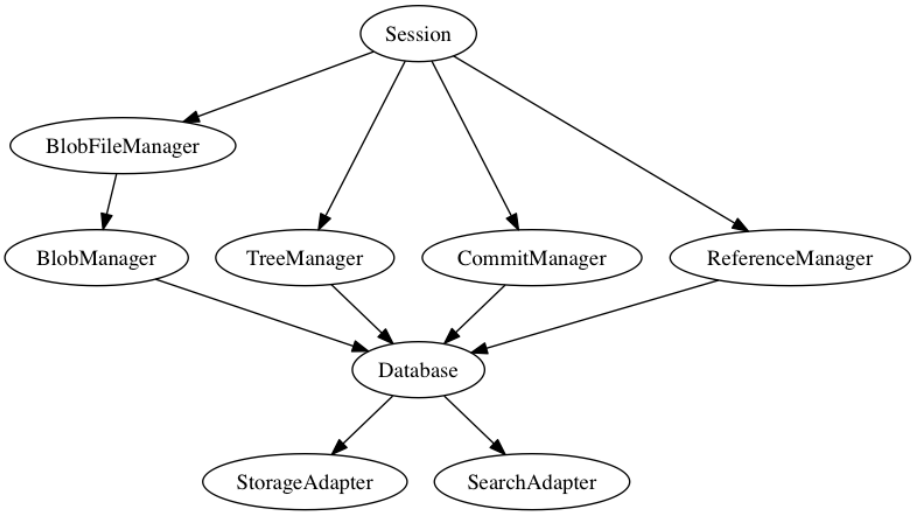

Abbildung 14: Schichten von “Distributed Storage”

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 einfache Mechanismen, 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 Backup-Server. Um die Einstellungen des Replikators zu persistieren fügt der Eventhandler eine “ReplicatorPolicy” an das Model an. Diese Strategie wird dann zusätzlich mit Model 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 SpeicherAdapter nicht vorhanden sind. Dies erkennt der Replikator und beginnt alle bekannten Servern anzufragen, ob sie dieses Objekt kennen. 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 15) ist die Serialisierung zu Beginn. Hierfür werden die Metadaten des Models anhand seiner Klasse aus dem “MetadataManager” geladen und anhand dieser Informationen serialisiert. Diese Daten werden mithilfe des “EventDispatcher”, aus dem Symfony2 Framework, in einem Event zugänglich gemacht. Die Eventhandler haben, die Möglichkeit die Daten zu bearbeiten und Strategien zu dem Model zu erstellen. Abschließend werden die Daten zuerst mithilfe des “StorageAdapter” persistiert und dann mithilfe des “SearchAdapter” in den Suchmaschinenindex aufgenommen.

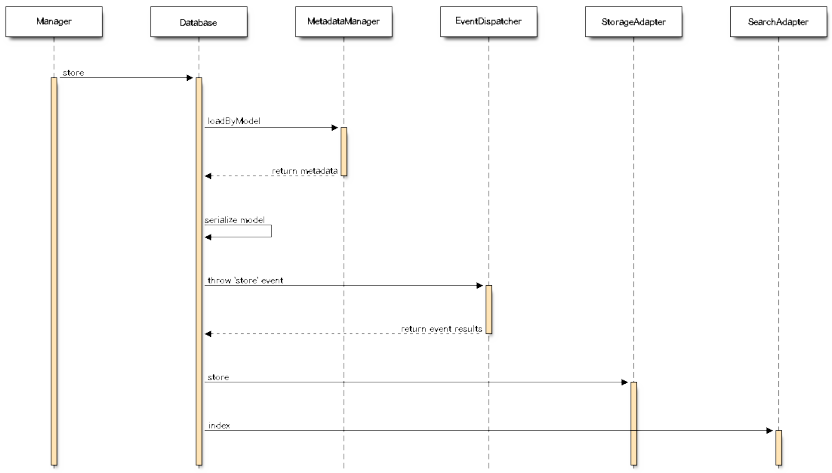

Abbildung 15: Objekte speichern

5.1.2 Objekte abrufen

Wie zu erwarten, ist der Abruf-Prozess von Daten, ein Spiegelbild des Speicher-Prozesses. Zuerst wird versucht mithilfe der Klassenmetadaten die Daten aus dem Storage zu laden. Diese Daten werden mithilfe des “EventDispatcher” den Eventhandler zur Verfügung gestellt. Diese haben dann die Möglichkeit, zum Beispiel fehlende Daten nachzuladen oder Änderungen an der Struktur durchzuführen. Diese veränderten Daten werden abschließend für den Deserialisierungs-Prozess herangezogen.

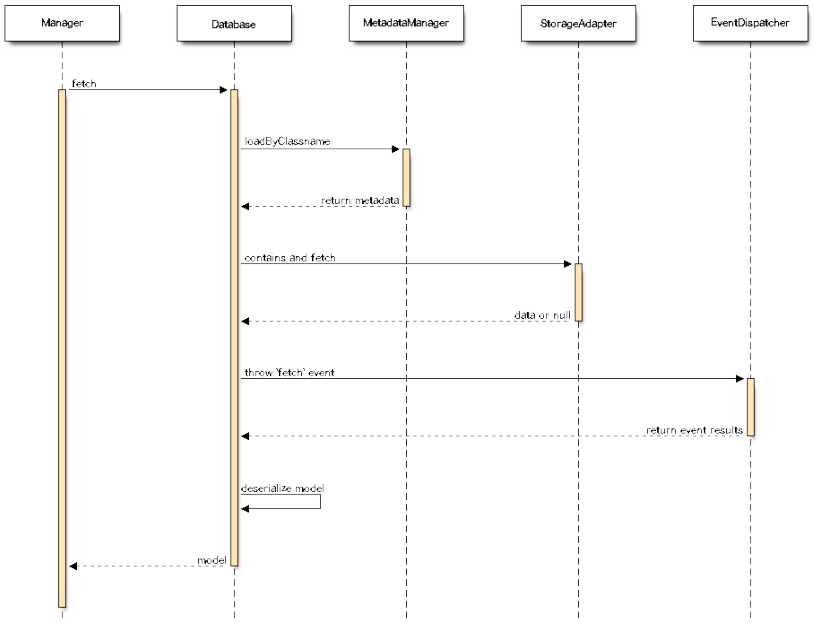

Abbildung 16: Objekte abrufen

Diese beiden Abläufe beschreiben eine lokale Datenbank, die die Möglichkeit bietet über Events die Daten zu verändern oder zu verwenden. Sie ist unabhängig zum 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 Verteilte 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 Backup-Server für die Objekte ermittelt.

Wie schon im Kapitel 4.3 erwähnt, gibt es verschiedene Arten die Backup-Server 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.

Full

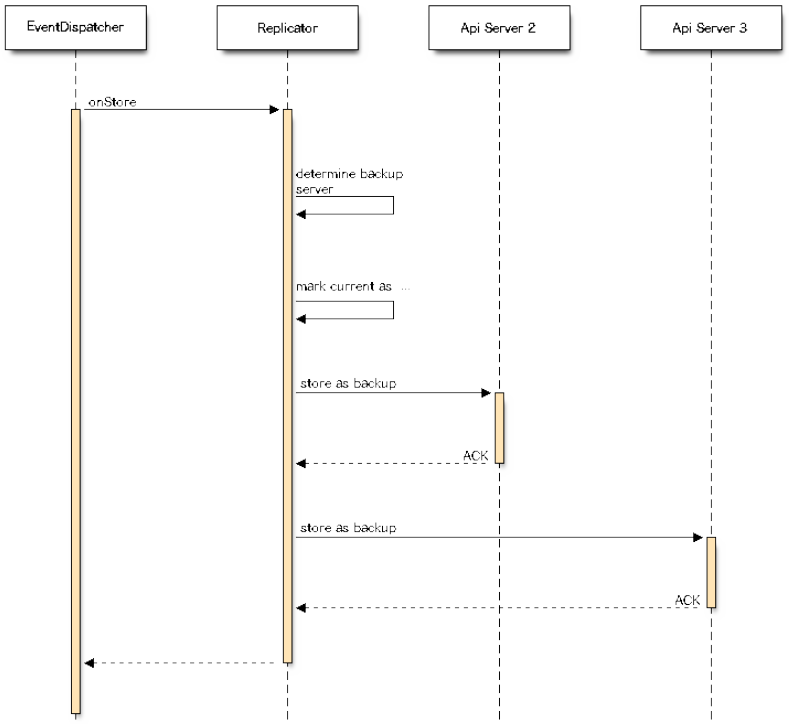

Abbildung 17: Replikationtyp “Full”

Bei einem “store” Event werden die Backup-Server per Zufall aus der Liste der vorhandenen Server ausgewählt und der aktuelle Server als Primäry-Server markiert. Anhand der Backup-Server Liste werden die Daten an die Server verteilt. Dazu wird der Reihe nach die Daten an die Server versendet und auf die Bestätigung gewartet. Damit wird der Konsistente Zustand der Datenbank

verifiziert. Abschließend wird die erstellte Strategie (policy) zu den Daten hinzugefügt, damit sie mit dem Daten persistiert wird und später wider verwendet werden kann. Dieser Prozess wird in der Abbildung 17 visualisiert.

Lazy

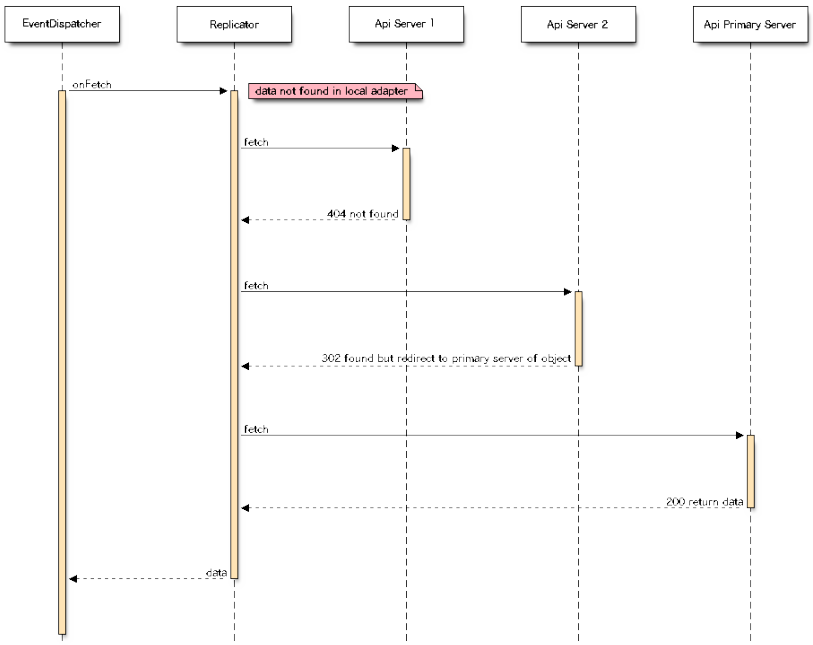

Abbildung 18: Replikator “Lazy”-Nachladen

Um fehlende Daten im lokalen Speicher nachzuladen, werden der Reihe nach alle bekannten Server abgefragt. Dabei gibt es vier mögliche Antworten (siehe Abbildung 18), 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 nur als Backup-Server markiert. Dieser Server kennt die genaue Adresse des Primary-Server und fügt diese in den Response

ein.

403 Das Objekt ist bekannt und der angefragte Server als Primary-Server für dieses Objekt markiert. Der Server überprüft die Zugangsberechtigung. Diese sind aber nicht gegeben und daher wird der Zugriff verweigert. Der Replicator erkennt daher, dass der Benutzer nicht Berechtigt ist die Daten zu lesen.

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

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. 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 dessen, kann jede Symcloud Installation sein eigenes Speichermedium verwenden. Dabei gibt es zwei Arten von Adaptern:

Storage Der “StorageAdapter” wird dazu verwendet, um serialisierte Objekte lokal zu speichern oder zu laden. Es implementiert im Grunde ein einfacher

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. Daher wird zum Beispiel für den Dateisystem-Adapter für jeden dieser Kontexte ein Ordner erstellt und für jeden Hash eine Datei. So kann schnell auf ein einzelnes Objekt zugegriffen werden.

Search Der "SearchAdapter" wird verwendet um die Metadaten zu den Objekten zu indexieren. Dies wird benötigt wenn die Daten durchsucht werden. Jeder Adapter implementiert folgende Befehle: `index`, `search` und `deindex`. Wobei auch hier mit Hash und Context gearbeitet wird. Über den Suchbefehl, können alle oder bestimmte Kontexte durchsucht werden. Als Prototypen Implementierung wurde die Bibliothek Zend-Search-Lucene[21] 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 Remote-Server kommunizieren kann. Dabei wird das Interface mit den Befehlen `fetch` und `store` implementiert.

Die Adapter sind also Klassen, die die Komplexität des Speichermediums bzw. der API von der restlichen Applikation trennt.

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 Models zu kommunizieren. Im Grunde genommen, sind dies meist Befehle um ein Objekt zu erstellen oder abzufragen.

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

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. 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 für eine einfache und effiziente Implementierung, des in Kapitel 4, beschriebenen Konzeptes. Es baut auf eine erweiterbare Hash-Orientierte Datenbank auf. Diese Datenbank wird mittels eines Replikator Eventhandlers zu einer verteilten Datenbank. Dabei hat die Datenbank keine Ahnung von dem verwendeten Protokoll. 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 beim Primary-Server angefragt werden müssen und nur für die Datensicherheit an Backup-Server verteilt werden.

5.2 Plattform

Die Plattform bzw. die Anwendung in die Bibliothek eingebettet wird, stellt dem Kern die Rest-API und die Authentifizierung zur Verfügung. Zusätzlich beinhaltet sie die Oberfläche um mit den Daten im 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 5.3 gegeben.

Eine Autorisierung zwischen den Servern, ist momentan nicht vorgesehen, es wäre allerdings möglich, über das OAuth2 Protokoll die Applikationen sogar dem richtigen Benutzer zuzuordnen. Dies wurde allerdings in der ersten Implementierungsphase nicht umgesetzt.

5.2.2 Rest-API

Die Rest-API ist wie schon im Kapitel 4.7 in vier verschiedene Schnittstellen aufgeteilt. Dabei werden die SULU internen Komponenten verwendet um die Daten für die Übertragung zu serialisieren und RESTful[22] aufzubereiten. Aufgrund dessen, dass Symcloud den Replikator verwendet, implementiert die Plattform den "ApiAdapter" um die Schnittstelle 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 konzipierte Konzept funktionieren kann. Es bietet in den Grundzügen, alle Funktionen an, um in einer späteren Implementierungsphase es zu einer vollständigen Plattform heranwachsen zu lassen.

[22] http://restcookbook.com/

5.3 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. [„OAuth – Wikipedia“ 2015]

Der Benutzer kann einer Applikation den Zugriff auf seine Daten autorisieren, die von einer andere Applikation zur Verfügung gestellt wird. Dabei werden nicht alle Details seiner Zugangsdaten preisgegeben. Typischerweise wird die Weitergabe eines Passwortes an Dritte vermieden. [„OAuth – Wikipedia“ 2015]

5.3.1 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 Geschützen 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 sesource server. Sie holt sich vorher die Genehmigung zu diesen geschützten Ressourcen [Hardt 2012, S. 5].

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

Neben diesen Rollen, spezifiziert OAuth2 diese Begriffe:

Access-Token Access-Tokens fungieren Zugangdsdaten zu geschützten Ressourcen. Es besteht aus einem string, der als Autorisierung für einen bestimmten Client ausgestellt wurde. Sie repräsentieren die Bereiche 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 der alte abgelaufen ist. Wenn der Autorisierungsserver diese Funktionalität zur Verfügung stellt, liefert er es mit dem Access-Token aus [Hardt 2012, S. 9].

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

Die Interaktion zwischen "Resource server" und "Authorization server" ist nicht spezifiziert. Der Autorisierungsserver und der Ressourcenserver können auf dem selben Server bzw. in der selben Applikation betrieben werden. Eine andere Möglichkeit wäre es, dass die beiden auf verschiedenen Server zu betreiben. Ein Autorisierungsserver kann auch Tokens für mehrere Ressourcenserver bereitstellen [Hardt 2012, S. 5].

5.3.2 Protokoll Ablauf

Der Ablauf einer Autorisierung [Hardt 2012, S. 6 ff] mittels Oauth2, der in der Abbildung 19 abgebildet ist, enthält folgende Schritte:

A) Der Client fordert die Genehmigung des "Resource owner". Diese Anfrage kann direkt an den Benutzer gemacht werden (wie in der Abbildung

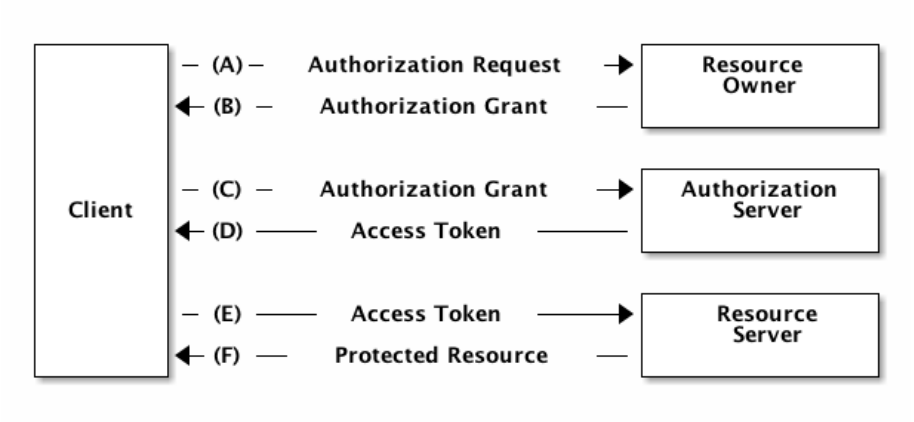

Abbildung 19: Ablaufdiagramm des OAuth

dargestellt) oder vorzugsweise indirekt über den "Authorization server" (wie zum Beispiel bei Facebook).

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

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

D) Der "Autorization server" 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 "Resource server" validiert den Token, validiert ihn und gibt die Ressource zurück.

5.3.3 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, die Dateien des Benutzers zu verwalten.

5.4 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 mit Hilfe der Symfony Konsole-Komponente[23] umgesetzt. Diese Komponente ermöglicht eine schnelle und unkomplizierte Entwicklung solcher Konsolen-Programme.

```
$ php jibe.phar
```

```
Token-Status: OK

  run jibe sync to start synchronization
```

[23] http://symfony.com/doc/current/components/console/introduction.html

Ein Konsolen-Programm besteht aus verschiedenen Kommandos, die über einen Namen aufgerufen werden können. Im diesem Beispiel wurde das Standard-Kommando des Tools aufgerufen. Über den Befehl `php jibe.phar sync` kann der Synchronisierungsvorgang gestartet werden. Alle Abhängigkeiten des Tools werden zusammen in einen PHAR-Container[24] geschrieben. Dieser ähnelt dem Format eines Java-JAR Archivs. Dieses Format wird in der PHP-Gemeinschaft oft verwendet um Komplexe Applikationen wie zum Beispiel PHPUnit[25] (ein Test Framework für PHP) auszuliefern.

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

Fehlende Argumente können vom Benutzer automatisch abgefragt werden. Eine Validierung, von zum Beispiel der URL, können direkt in einem Kommando implementiert werden.

Diese Kommandos stehen dem Benutzer zur Verfügung:

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

refresh-token Aktualisiert das Zugang-Token von OAuth2. Dies ist Notwendig, da diese über ein Ablaufzeitpunkt verfügen.

[24] http://php.net/manual/de/intro.phar.php
[25] https://phpunit.de/

status Gibt den aktuellen Status des Zugangs-Token aus. Wenn kein andere Kommando angegeben wurde, wird dieses aufgerufen.

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

5.4.1 Architektur

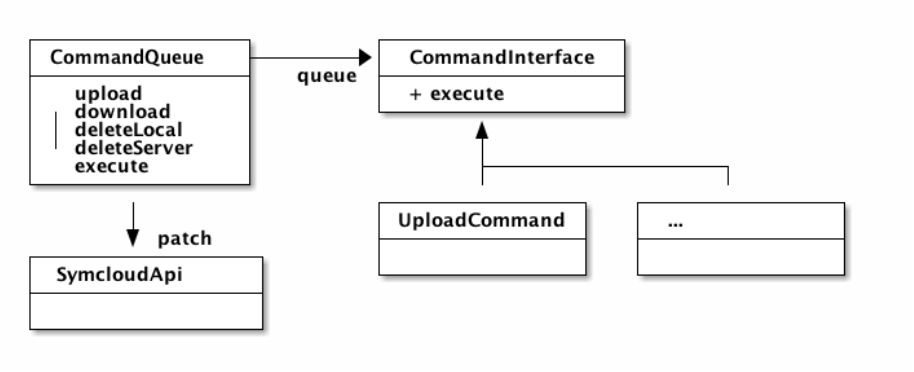

Abbildung 20: Architektur von Jibe

Der Zentrale Bestandteil von Jibe ist eine `CommandQueue` (siehe Abbildung 20). Sie sammelt alle nötigen Kommandos ein und führt sie dann nacheinander aus. Diese Queue ist nach den "Command Pattern" entworfen. Folgende Befehle können dadurch aufgerufen werden:

Upload Datei auf den Server hochladen

Download Datei wird vom Server heruntergeladen und lokal in die Datei geschrieben.

DeleteServer Datei auf dem Server wird gelöscht.

DeleteLocal Lokale Datei wird gelöscht.

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

5.4.2 Kommunikation

Aufgrund der Datenstruktur ist es notwendig, nicht nur die Daten hochzuladen oder zu löschen, sondern auch alle zusammengefassten Änderungen in einem Request an den Server zu senden. Daher retourniert jedes Kommando ein zusätzlicher Befehl, die am Ende des Synchronisierungsvorgans gesammelt an den Server gesendet werden. Diese Befehle weisen folgende Struktur auf:

```
{
    "command": "delete",
    "path": "/test-file.txt"
}
```

Dieses Kommando führt auf dem Server dazu, dass die angegebene Datei aus dem Baum des Benutzers entfernt wird.

Update

```
{
    "command": "update",
    "path": "/test-file.txt",
    "file": "<hashvalue>"
}
```

Dieses Kommando führt auf dem Server dazu, dass die angegebene Datei einen neuen Inhalt besitzt. Identifiziert wird der neue Inhalt, durch den Hashwert, der beim Upload im “Response” retourniert wird.

Commit

```
{
    "command": "commit",
    "message": "<message>"
}
```

Am Ende des PATCH-Requests[26] wird ein Commit ausgeführt. Dieser erstellt am Server einen neue Version des Trees. Aufgrund der Tatsache, dass dies in einem einzigen Request ausgeführt wird, kann es in Zukunft über eine Transaktion gesichert werden.

5.4.3 Abläufe

Für einen kompletten Synchronisierungsvorgang werden folgende Informationen benötigt:

Lokale Hashwerte werden aus den aktuellen Dateibeständen generiert.

Zustand der Dateibestände 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. Diese werden verwendet, um zu erkennen, dass Dateien auf dem Server verändert haben bzw. gelöscht wurden.

[26] http://tools.ietf.org/html/rfc5789#section-2.1

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 2 gibt Aufschluss über die Erkennung von Kommandos aus diesen Informationen.

Tabelle 2: Evaluierung der Zustände

hash old v.	hash version	Download	Upload	Delete local	Delete server	Conflict

1	X	1 1	X	1 2	Nothing to be	x	x	x	x	x
2	X	1 1	Y	1 2	done Server file	x	x			
3	Y	1 - -	X	2 -	changed,					
4	Y	1 1	Z	1 -	download new					
5	Y		Y	1	version Client file					
6	X		-		change, upload					
7	-		X		new version Client					
8 9	X		-		and Server file					
	-		X		changed, conflict					
					Server file					
					changed but					
					content is the					
					same New client					
					file, upload it New					
					server file,					
					download it					
					Server file deleted,					
					remove client					
					version Client file					
					deleted, remove					
					server version					

Folgende TODOs für diese Tabelle:

- Lesbarkeit verbessern
- Alter Dateihash hinzufügen
- Ändere X/Y und 1/2 zu Allgemein gültigen Werten (n/n+1)
- Muss aktuell gehalten werden

Beispiel der Auswertungen anhand des Falles Nummer vier:

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. Diese Konflikt Auflösung ist nicht Teil der Arbeit, wird allerdings im Kapitel 7.1 kurz behandelt.

5.4.4 Installation

Um nun Jibe mit einer aktiven Installation zu verbinden, müssen folgende Schritte ausgeführt werden.

TODO aktuell halten (evtl. in den Anhang?)

Server

- Erstellen eines OAuth2 Clients mit dem Grant-Type "password, refresh_token": `app/console symcloud:oauth2:create-client sync http://www.example.com -g password -g refresh_token`

Lokaler Rechner

- In dem Order, der synchronisiert werden soll, folgendes Kommando ausführen: `php jibe.phar configure` und die geforderten Eingaben durchführen.
- Um eine Synchronisierung durchzuführen reicht es folgendes Kommando auszuführen: `php jibe.phar sync`

5.4.5 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.

5.5 Zusammenfassung

TODO nur Notizen

Es kann pro Bucket festgelegt werden, welcher Benutzer Zugriff auf diesen hat bzw. ob er diese durchsuchen darf. Dies bestimmt die Einstellungen des Replikators, der die Daten anhand dieser Einstellungen über die verbundenen Instanzen verteilt.

Beispiel:

- Bucket 1 hat folgende Policies:
- SC1 User1 gehört der Bucket
- SC2 User2 hat Leserechte
- SC3 User3 hat Lese- und Schreibrechte

Der Replikator wird nun folgendermaßen vorgehen.

1. Die Metadaten des Buckets werden auf die Server SC2 und SC3 repliciert.
2. Die Nutzdaten (aktuellste Version) des Buckets werden auf den Server SC3 repliciert und aktuell gehalten.

3. Beides wird automatisch bei Änderungen durchgeführt.
4. Beim lesen der Datei wird SC2 bei SC1 oder SC3 (je nach Verfügbarkeit) die Daten holen und bei sich persistieren. Diese Kopie wird nicht automatisiert von SC3 upgedated, sie wird nur bei Bedarf aktualisiert.
5. Bei Änderung einer Datei des Buckets auf SC3 werden die Änderungen automatisch auf den Server S1 gespielt.

Die Suchschnittstelle wird bei der Suche nach Dateien für den User2 oder User3 auf das Bucket durchsuchen. Jedoch wird der User3 die Daten in seinem eigenen Server suchen und nicht bei S1 nachfragen. Da S2 nicht immer aktuelle Daten besitzt, setzt er bei der Schnittstelle S1 eine Anfrage ab, um die Suche bei sich zu Vervollständigen.

6 Ergebnisse

7 Ausblick

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

7.1 Konfliktbehandlung

7.2 Verteilung von Blobs

Besseres Verfahren wie Zufall verwenden, dass 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 viel Objekte besitzt oder wenig Speicherplatz besitzt).

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 wider 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 3: 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, 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

Name	Description
	encryption key that was used for the object.
x-amz-server-	Indicates whether server-side encryption
side-encryption-	with customer-provided encryption keys (SSE-C) is enabled.
customer-algorithm	

Installation

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

Lokal

Verteilt

Literaturverzeichnis

„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

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$_{t}$he$_{g}$it$_{o}$bject$_{m}$odel.htm

„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

„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].

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

„OAuth – Wikipedia“ [2015]: OAuth – Wikipedia http://de.wikipedia.org/wiki/OAuth.

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

„Owncloud Features“ [2015]: 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].

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$_{r}$eplication$_{w}$orks.php.

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

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

„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.