JUST DO IT
HÖHERE ABTEILUNG FÜR INFORMATIONSTECHNOLOGIE

tgm
Die Schule der Technik

Matura Gliederung
Dezentrale Systeme

Systemtechnik Labor
5BHITT 2015/16

Manuel Reiländer

Version 1.0

Inhalt

Kompetenzen

1. können verteilte und redundante Dateisysteme einsetzen
2. können die in dokumentenbasierten, nachrichtenorientierten und serviceorientierte Systemen eingesetzten offenen Dokumentenformate und Auszeichnungssprachen erläutern
3. können ausfallsichere replizierte Datenbanksysteme und dezentrale Systeme installieren, warten und entwerfen
4. können den Datenbankentwurf in verteilten Systemen durchführen und zur dynamischen Generierung von Inhalten einsetzen

Themenbereiche

1 Clound Computing und IoT

1.1 Kompetenz Dateisysteme

Google File System

Wie auch andere verteilte Dateisysteme, verfolgt auch das von 3 Google Mitarbeitern, *Sanjay Ghemawat*, *Howard Gobioff* und *Shun-Tak Leung* entwickelte *Google File System (GFS)*, die wesentlichen Grundprinzipien von verteilten Datensystemen:

- Performance
- Erweiterbarkeit
- Zuverlässigkeit
- Verfügbarkeit

Zusätzlich zu diesen grundsätzlichen Anforderungen, werden noch 3 spezielle Anforderungen an das GFS gestellt. Erstens, einzelne Komponenten müssen nicht immer qualitativ hochwertig sein d.h. der Ausfall einzelner Komponenten ist sehr häufig und wird somit nicht als Seltenheit betrachtet. Es kann auch zu Fehlern kommen, nach denen einzelne Komponenten vielleicht nicht mehr funktionieren. Eine ständige Überwachung, Fehlererkennung, Fehlertoleranz und automatische Wiederherstellung nach Fehlern sind demnach eine wichtige Anforderung an das GFS.
Zweitens, Dateien in der heutigen Zeit werden immer größer, mehrere GB oder sogar TB sind hier keine Seltenheit mehr. Bei solchen Datenmengen, ist es unüblich diese in mehrere kleine Dateien abzuspeichern. Man entscheidet sich hierfür für eine große Datei, in der mehrere Applikationsdaten in Form von komprimierten Applikationsobjekten gespeichert werden.
Drittens, an Dateien werden eher Daten angehängt, als vorhandene überschrieben. Diese werden in den meisten Fällen nur einmal beschrieben und dann nur noch, meist sequentiell, gelesen. Überträgt man dieses Zugriffsmuster auf große Dateien, rückt der Fokus auf das Optimieren von Lesezugriffen sowie das Gewährleisten der Atomarität. [1]

Architektur

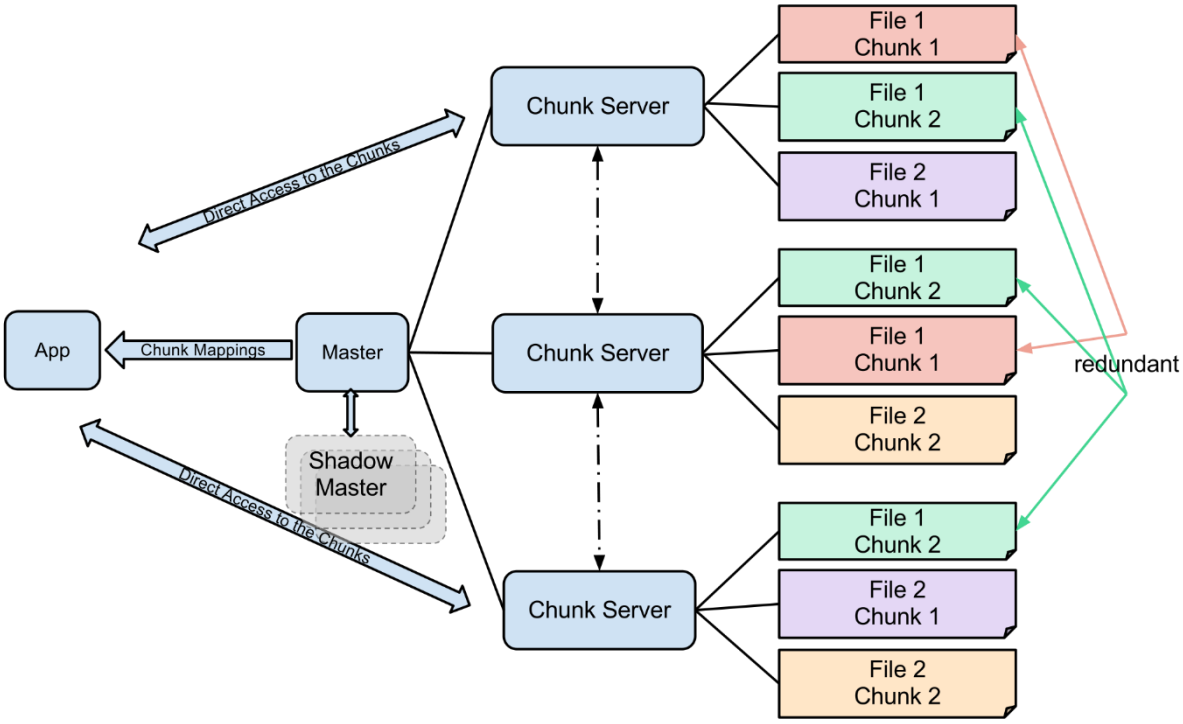

Abbildung 1 GFS Architektur [Abb1]

Wie in Abbildung 1 zu sehen ist, besteht das GFS aus einem Master und beliebig vielen Chunkservern, wobei die Chunkserver meist herkömmliche Linux Server sind. Werden Dateien abgespeichert, so werden diese in so genannte Chunks unterteilt. Diese Chunks werden nun auf mehrere Chunkserver aufgeteilt und dort redundant als Linux Dateien abgespeichert. Bei der Erstellung eines solchen, bekommt dieser einen so genannten *chunk handle* zugewiesen mit dem er eindeutig identifiziert werden kann. Dieser hat eine Größe von 64-Bit, ist unveränderbar und im ganzen System eindeutig. Der Master hat die Aufgabe, alle Chunkserver zu verwalten und zusätzlich mit sogenannten „*HeartBeat messages*", deren aktuellen Status abzufragen und Anweisungen zu geben. Diese Nachrichten werden regelmäßig in einem bestimmten Intervall gesendet.

Alle Metadaten zu einem Dateisystem bzw. Server, wie Name des Servers, Zugriffsinformationen, Mapping von Files zu Chunks und die aktuelle Position des Chunks, sind auf dem Master gespeichert. Möchte ein Client nun Metadaten eines bestimmten Chunkservers abrufen, so muss dieser mit dem Master kommunizieren. Sämtliche Dateioperationen, wie z.B. das Lesen und Schreiben von Dateien, erfolgen jedoch direkt über den Chunkserver. [6]

Hadoop File System

Wie auch beim GFS, werden beim HDFS Dateien, in so genannte Chunks unterteilt. Diese haben eine feste Größe von, anders als beim GFS (64 MB), 128 MB und werden auf so genannten DataNodes persistent gespeichert. Diese Chunks werden üblicherweise auf 3 verschiedenen DataNodes repliziert, jedoch kann die Anzahl an Replicas pro Datei eingestellt werden.

NameNode

Der NameNode managt alle Metadaten der einzelnen Chunks wie z.B. den Standort und das Mapping einer Datei zu mehreren Blocks und hält diese im RAM. Diese Metadaten werden im HDFS auch *Images* genannt. Diese Images werden jedoch auch persistent, auf dem lokalen Dateisystem des NameNodes gespeichert, wobei man diese *Checkpoints* nennt. Zusätzlich werden noch alle Änderungen an einem Image im so genannten *Journal* geloggt. Es ist auch noch möglich Journal und Checkpoints auf anderen Servern zu replizieren.
Auf dem NameNode, werden alle Dateien als so genannte *inodes* identifiziert. Dieser inode enthält Informationen über eine Datei wie Zugriffsrechte, Modifizierungs- und Zugriffszeit, namepsace und Größe der Datei.

DataNode

Wie vorher bereits erwähnt befinden sich die Chunks, in denen die eigentlichen Daten einer Datei enthalten sind, auf DataNodes. Diese Chunks werden nun in 2 Dateien unterteilt, die erste enthält die eigentlichen Daten eines Chunks und die andere enthält bestimmte Metadaten über den Block wie zum Beispiel Prüfsumme und einen *generation stamp*.
Die Größe eines Chunks ist flexibel, das heißt auch wenn ein Block (128MB) nur zur Hälfte voll ist, verbraucht er nicht den ganzen Speicherplatz (128MB) auf der Festplatte, wie es bei herkömmlichen Dateisysteme der Fall ist, sondern eben nur die Hälfte. [2]

Sun Network File System

Ein verteiltes Dateisystem besteht aus mehreren Komponenten. Auf der Seite des Clients, gibt es Applikationen, die auf Dateien und Verzeichnisse über das Client Dateisystem zugreifen. Diese rufen so genannte *system calls* auf dem Client Dateisystem auf wie z.B. `open()`, `read()`, `write()` usw. um Dateien die am Server gespeichert sind aufzurufen. Für die Client Applikation scheint es keinen sichtbaren Unterschied zwischen dem Lokalen Dateisystem und dem Dateisystem des Servers zu geben.

Die Aufgabe des Dateisystems auf dem Client ist es, die erforderlichen Aktionen auszuführen um diese *system calls* erfolgreich auszuführen. Wenn der Client beispielsweise einen `read()` request anfordert, muss sich das Client Dateisystem darum kümmern, einen bestimmten Block einer Datei vom Server Dateisystem zu lesen. Dieser liest dann diesen bestimmten Block von seiner lokalen Festplatte und gibt dem Client die angeforderten Daten zurück. Der Client kopiert diese Daten nun in seinen *user buffer*, damit sie vom system call `read()`, von dort aus abgerufen werden können.
Die beiden wichtigsten Komponenten in diesem simplen Beispiel, ist das Dateisystem des Clients und das des Servers, da diese letztendlich die eigentliche Arbeit verrichten müssen. [3]

Einsatzgebiete der verteilten Dateisysteme

Nun stellt sich natürlich die Frage, unter welchen Umständen/Umfelder die vorher erwähnten verteilten Dateisysteme am besten geeignet sind. Zuvor muss erwähnt werden, dass eine getrennte Ansicht des GFS und HDFS keinen Sinn macht, da das HDFS eine Open-Source Implementierung des GFS ist.

Es ist ganz offensichtlich, dass das HDFS besonders gut geeignet ist um große Datenmengen zu verarbeiten (man spricht von petabytes an Daten). Bei kleineren Datenmengen die im Giga- oder Megabyte Bereich liegen, würde das HDFS jedoch eher negativ punkten (Hotspot Problem). Wie bereits erwähnt, ist es eine Open-Source Implementierung des Google File Systems, geschrieben in Java. Dadurch, dass Java eine einfache Programmiersprache ist, wird er Umgang und die Verwaltung des HDFS deutlich vereinfacht.

Das Network File System wird dann verwendet, wenn man Clients hat die untereinander viele Daten teilen. Hat man zum Beispiel mehrere Benutzer die sich auf unterschiedlichen PCs einloggen müssen, so kann man mit dem NFS unter Linux das */home* Verzeichnis auf einen zentralen Server platzieren und es auf jedem Client mounten sodass ein Benutzer, wenn er von einem Computer zu einem anderen wechselt, mit denselben Daten weiterarbeiten kann.

Ein gemeinsamer Nachteil beider DFSs (Distributed File Systems), ist das Problem mit der Sicherheit (AAA). Sie sollten nur innerhalb eines sicheren Netzwerks hinter einer Firewall benutzt werden, d.h. Clients die auf die Systeme zugreifen sollten sich im gleichen sicheren Netzwerk befinden.

1.2 Kompetenz Dokumentenformate

Um Informationen zwischen zwei Endpunkten übertragen zu können muss definiert werden, wie Daten dieser Information bei der Übertragung strukturiert werden. Es muss ein Standard definiert werden, damit beide Endpunkte wissen mit welcher Struktur Daten gesendet werden müssen, damit sie vom zweiten Endpunkt verarbeitet werden können.

Die 2 am weitesten verbreitete Dokumentenformate die zur Übertragung von Informationen genutzt werden, sind XML und JSON. Beide haben ihre Vor- und Nachteile die im Folgenden erläutert werden.

XML

XML ist eine der ältesten Formate die von den meisten Systemen unterstützt wird. Wird JSON von einem System nicht unterstützt, so wird man am ehesten zu XML greifen. Zusätzlich bringt es noch den Vorteil XML-Dokumente zu validieren, um eine korrekt eingehaltene Syntax zu gewährleisten.

Es bietet zusätzlich noch die Möglichkeit, ein XML-Dokument in ein anderes Dokumentenformat zu transformieren (XSLT), was in einigen Situationen sehr hilfreich sein kann. Will man z.B. ein XML-Dokument als Tabelle in einem Browser anzeigen, so kann man durch Verwendung von XSTL das XML-Dokument in ein HTML-Dokument transformieren und an einen Browser weitergeben. Dieser kann dann das Dokument ohne Verwendung zusätzlicher Software anzeigen.

Ein weiterer Vorteil von XML ist XPath. Dies ist eine Query Language und ermöglicht es, bestimmte Stellen in einem XML-Dokument zu finden und zu verarbeiten. Dadurch ist es nicht mehr notwendig, selbst einen Parser zu schreiben der ein XML-Dokument nach bestimmten Stellen absucht.

JSON

JSON ist das gängigste Dokumentenformat für die Übertragung von Informationen, da sie sehr einfach zu lesen und zu verarbeiten ist. Bei gleicher Informationsmenge muss man, wenn man für die Übertragung JSON verwendet, im Gegensatz zu XML kleinere Datenmengen übertragen. Jedoch gibt es Systeme, die JSON noch nicht unterstützen wobei man in solch einem Fall zu XML greift.

Zwar gibt es für JSON auch diverse Softwarebibliotheken die eine Transformation in ein anderes Dokumentenformat erlauben, jedoch sind diese noch nicht so weit verbreitet wie XSLT. Das bedeutet, wenn eine Transformation notwendig ist, sollte eher XML als Format verwendet werden.

Weitere Formate

Sicher gibt es noch zahlreiche anderen Formate, die sich unter bestimmten Umständen besser eignen als die beiden vorher erwähnten, jedoch sind dies vergleichsweise wenige.

1.3 Kompetenz Datenbanksysteme

NoSQL vs RDBMSs

Das Internet der Dinge wird immer weiter verbreitet und die Menge an Daten die es zu verarbeiten gilt steigt dementsprechend rasant an. Viele, teilweise sehr verschiedene, Endgeräte liefern unterschiedliche Datenstrukturen die es zu speichern gilt. Da diese Daten so unterschiedlich sind und es in einem RDBMS vordefinierte Schemen zur Speicherung der Daten gibt, eignet sich für das Persistieren der Daten eher ein NoSQL System.

MongoDB

MongoDB ist ein dokumentbasiertes NoSQL System, mit dem es möglich ist jede beliebige Dokumentenstruktur in einer Datenbank abzuspeichern. Diese Dokumente werden im Hintergrund als BSON-Objekte abgespeichert um erstens mehr Datentypen zu unterstützen und zweitens ein effizienteres de- und encoden in vielen verschiedenen Programmiersprachen zu ermöglichen. In Verbindung mit dem CAP-Theorem setzt MongoDB auf Consistency und Partition Tolerance.

CouchDB

CouchDB ist ebenso ein dokumentenbasiertes NoSQL System, das jedoch im Gegensatz zu MongoDB den Fokus eher auf Availablility anstatt Consistency legt. Um Daten in der Datenbank zu speichern, muss man ein HTTP Request geschickt werden. Da intern Daten als JSON-Objekt abgespeichert werden, erleichtert das den Umgang mit der Datenbank, da man zum Speichern von Daten lediglich ein JSON-Objekt als HTTP Request senden muss.

Einsatz

Wann man welches Datenbanksystem verwendet, hängt davon was einem wichtiger ist: Consistency oder Partition Tolerance.

1.4 Kompetenz Datenbankentwurf

Um eine Tabelle in einer relationalen Datenbank auf mehreren Systemen aufzuteilen, gibt es 3 Ansätze.

Zum einen gibt es den Ansatz der horizontalen Fragmentierung, wobei die Aufteilung anhand der Zeilen erfolgt. Hat man zum Beispiel eine Tabelle *User* mit den Spalten *username*, *password* und *region*, so könnte man diese Tabelle nach den Regionen der User aufteilen. Das heißt gäbe es zum Beispiel 500 Benutzer in Region1, so könnte man diesen Teil der Tabelle auf einen Server in dieser Region platzieren, um die Performance dadurch deutlich zu steigern.

Auf der anderen Seite gibt es den Ansatz der vertikalen Fragmentierung, wobei die Aufteilung anhand der Spalten erfolgt. Hat man nun zum Beispiel wieder eine Tabelle *User* mit einer zusätzlichen Spalte *ID* als Primärschlüssel, so könnte man die Datensätze mit den Spalten *ID, password* und *region* auf einen Server legen und den Teil mit den restlichen Spalten *username* und *ID* auf einen anderen Server. Wichtig zu beachten ist hier, dass jedes Fragment den Primärschlüssel beinhaltet, um die Konsistenz der Daten beizubehalten.

Nun gibt es auch noch die Möglichkeit beide Varianten der Fragmentierung zu kombinieren, das heißt eine Fragmentierung nach Spalten und Zeilen gleichzeitig. Um bei dem Beispiel mit der Tabelle *User* zu bleiben, kann man hier alle Datensätze so aufteilen, dass die Datensätze aller User einer bestimmten Region zusätzlich noch nach bestimmten Spalten aufgeteilt werden.

2 Automatisierung, Regelung und Steuerung

n-1

3 Security, Safety, Availability

3.1 Kompetenz Dateisysteme

Encryption Zones

Beim HDFS ist es möglich so genannte *ecryption zones* zu erstellen, wobei diese nichts Anderes als Bereiche (Verzeichnisse) sind, in denen alle gespeicherten Daten bei einem Schreibvorgang verschlüsselt und bei einem Lesevorgang vom HDFS entschlüsselt werden. Die Verschlüsselung auf Applikationsebene auszulagern hat den Vorteil, dass bestehende Anwendungen ohne jegliche Änderungen auf die verschlüsselten Daten zugreifen können.

Die Verschlüsselung funktioniert so, dass jeder Verschlüsselungszone einen eigener Verschlüsselungskey (*encryption zone key*) zugewiesen wird. Jede Datei innerhalb dieser Zone besitzt nun wieder einen eigenen einmaligen Key (DEKs, data encryption key). Diese DEKs werden nun mit dem encryption zone key verschlüsselt (sie heißen dann EDEKs, *encrypted data encryption keys*) und mit den Metadaten auf dem NameNode persistiert.

Um diese Funktionalitäten anbieten zu können wird im HDFS ein neuer Service benötigt. Diesen nennt man *Hadoop Key Management Server (KMS)*. Wird nun eine neue Datei in einer encryption zone erstellt, so bittet der NameNode den KMS einen neuen EDEK, verschlüsselt mit dem encryption zone key, zu generieren welche dann wiederrum benutzt werden um die Datei zu ver- oder entschlüsseln.

Wird eine Datei in einer encryption zone gelesen, so gibt der NameNode dem Client den EDEK und die Version des encryption zone keys, die benutzt wurde um den EDEK zu verschlüsseln. Der Client kann nun den KMS bitten, den EDEK zu entschlüsseln wobei hier überprüft werden kann, ob der Client ausreichende Berechtigungen hat, um auf die Version des encryption zone keys zuzugreifen. Wurden alles erfolgreich abgeschlossen, kann der Client den entschlüsselten EDEK (=DEK) dazu verwenden die Datei zu entschlüsseln und deren Inhalt zu verarbeiten. [4]

3.2 Kompetenz Dokumentenformate

XML Encryption

Der heutige Standard zur sicheren Übertragung von Daten über mehrere Netzwerke heißt *Transport Security Layer (TLS)*, welches dem *Secure Socket Layer (SSL)* folgt. Jedoch werden von diesem Protokoll 2 wesentliche Dinge nicht abgedeckt:

- Die Verschlüsselung der zu übertragenen Daten
- Eine sichere Verbindung zwischen mehr als nur 2 Endpunkten

Mit der XML Verschlüsselung können sensible Daten, als auch unsensible Daten im selben Dokument gesendet werden. Dazu gibt es 3 verschiedene Möglichkeiten, die im Folgenden näher beschrieben werden. [5]

Alle nachfolgenden Erklärungen, gehen von diesem XML-Dokument aus.

```
<purchaseOrder>
        <Order>
                <Item>book</Item>
                <Id>123-958-74598</Id>
                <Quantity>12</Quantity>
        </Order>
        <Payment>
                <CardId>123654-8988889-9996874</CardId>
                <CardName>visa</CardName>
                <ValidDate>12-10-2004</ValidDate>
        </Payment>
</purchaseOrder>
```

Code 1 Beispiel XML-File [Abb3]

Verschlüsselung des gesamten Dokuments

Die folgende Abbildung zeigt die verschlüsselte Variante des in **Fehler! Verweisquelle konnte nicht gefunden werden.** gezeigten XML-Files.

```
<?xml version='1.0' ?>
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
                Type='http://www.isi.edu/in-notes/iana/assignments/media-types/text/xml'>
        <CipherData>
                <CipherValue>A23B45C56</CipherValue>
        </CipherData>
</EncryptedData>
```

Code 2 Verschlüsseln der gesamten Datei [Abb2]

Wie man in **Fehler! Verweisquelle konnte nicht gefunden werden.** erkennen kann, gibt es einen Tag mit dem Namen *CipherValue*, welcher den eigentlichen verschlüsselten Inhalt darstellt. Dieser befindet sich in einem *CipherData* Tag, welcher sich wiederrum in einem *EncryptedData* Tag befindet. Der *EncryptedData* Tag beinhaltet zusätzliche Informationen zur verwendeten Verschlüsselungsmethode, als Attribute. [5]

Verschlüsselung eines einzelnen Elements

```
<?xml version='1.0' ?>
<PurchaseOrder>
        <Order>
                <Item>book</Item>
                <Id>123-958-74598</Id>
                <Quantity>12</Quantity>
        </Order>
        <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'

        xmlns='http://www.w3.org/2001/04/xmlenc#'>
                <CipherData>
                        <CipherValue>A23B45C564587</CipherValue>
                </CipherData>
        </EncryptedData>
</PurchaseOrder>
```

Code 3 Verschlüsseln eines einzelnen Elements [Abb4]

Wie in *Code 3* zu sehen, wurde diesmal nur der *Payment* Tag verschlüsselt, wobei sich jedoch der Type in *EncryptedData* geändert hat.

Verschlüsselung des Inhalts eines Elements

```
<?xml version='1.0' ?>
<PurchaseOrder>
        <Order>
                <Item>book</Item>
                <Id>123-958-74598</Id>
                <Quantity>12</Quantity>
        </Order>
        <Payment>
                <CardId>
                        <EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Content'

        xmlns='http://www.w3.org/2001/04/xmlenc#'>
                                <CipherData>
                                        <CipherValue>A23B45C564587</CipherValue>
                                </CipherData>
                        </EncryptedData></CardId>
                <CardName>visa</CardName>
                <ValidDate>12-10-2004</ValidDate>
        </Payment>
</PurchaseOrder>
```

Code 4 Verschlüsseln des Inhalts eines Elements [Abb5]

In *Code 4* sieht man, dass lediglich der Inhalt des Elements *CardId* verschlüsselt wurde, wobei sich wieder der Type in *EncryptedData* geändert hat.

3.3 Kompetenz Datenbanksysteme

MongoDB

MongoDB bietet 2 Arten der Verschlüsselung an. Es kann der zu übertragende Inhalt verschlüsselt werden und die Datenbank selbst. Bei der Verschlüsselung der Datenbank gibt es 2 Möglichkeiten, die aber auch gemeinsam genutzt werden können. Einerseits ist das natürlich die Verschlüsselung auf Applikations- und andererseits auf Datenbankebene. [6]

Transport Encryption

Es wird sowohl TLS als auch SSL zum Verschlüsseln der Übertragung unterstützt. Zusätzlich gibt es auch noch ein Authentication System, um die Identität des Clients sicherzustellen, wobei zahlreiche Authentication Systeme unterstützt werden.

- SCRAM-SHA-1
- MongoDB Challenge and Response
- X.509 Certificate Authentication

Die folgenden 2 werden nur von der kommerziellen Version der MongoDB unterstützt.

- LDAP proxy authentication
- Kerberos authentication

[6]

Encryption at Rest

MongoDB Enterprise bietet ab Version 3.2 eine eingebaute Verschlüsselungsoption an, bei der nur Endpunkte die den Verschlüsselungskey besitzen, die verschlüsselten Daten lesen können.

Ist die Verschlüsselung aktiviert, so wird folgendes durchgeführt:

- Erzeugen eines Master Schlüssel
- Erzeugen der Schlüssel für jede Datenbank
- Daten in einer Datenbank mit dem entsprechenden Datenbankschlüssel verschlüsseln
- Datenbankschlüssel mit dem Master Schlüssel verschlüsseln

Die Verschlüsselung basiert auf Dateisystemebene und liegen nur unverschlüsselt im Arbeitsspeicher oder während einer Übertragung.

Die Datenbankschlüssel befinden sich auf dem gleichen Server auf dem auch die Datenbank liegt, werden jedoch erst persistiert, wenn sie mit dem Master Schlüssel verschlüsselt worden sind.

Der Master Schlüssel ist der einzige Schlüssel der extern verwaltet werden muss, um „volle" Sicherheit zu gewährleisten. Dazu bietet MongoDB folgende 2 Möglichkeiten an:

- Einsatz einer Third-Party Software durch das KMIP
- Lokale Schlüsselverwaltung durch ein Key-File (weniger sicher)

[6]

3.4 Kompetenz Datenbankentwurf

-

4 Authentication, Authorization, Accounting

4.1 Kompetenz Dateisysteme

HDFS Authentication and Authorization

Zuerst einmal muss man klarstellen wie Berechtigungen in Hadoop gehandhabt werden. Diese sind denen in Linux sehr ähnlich. Die Ausgabe eines *ls -l* in einem Verzeichnis in der Linux Konsole sieht meist wie folgt aus.

```
manuel@linux ~/test % ls -l
total 8
-rw-r--r-- 1 manuel manuel    0 Jun  1 16:21 test.txt
drwxr-xr-x 2 manuel manuel 4096 Jun  1 16:21 test1
drwxr-xr-x 2 root   root   4096 Jun  1 16:21 test2
```

Zwischen Vezeichnissen und Dateien wird durch ein Symbol am Anfang der ersten Zeichenkette unterschieden (d für directory). Danach kommen 3 Gruppen, *User*, *Group* und *Others*, für die es jeweils die Bereichtigungen *r (read), w (write)* und *x (execute)* gibt. Die beiden Zeichenketten nach den Berechtigungen, geben die Besitzer an. Links steht der User und rechts die Gruppe.

Wird nun zum Beispiel eine Operation auf eine Datei ausgeführt, so muss der User der auf die Datei zugreifen will folgende Kriterien erfüllen.

- Mitglied der Gruppe
- Oder Besitzer der Datei

Sind diese Berechtigungen überprüft, so müssen zusätzlich noch die richtigen Rechte für den Vorgang (z.B. Schreibvorgang) vorhanden sein. In Hadoop gibt es zum Ändern dieser Berechtigungen dieselben Kommandos wie in Linux (chmod, chgrp, chown).

Nun bringt Hadoop, bei nicht konfigurierter Authorisierungseinstellungen, 2 Probleme mit sich. Einerseits kann man sich sehr leicht als jemand anderer ausgeben, als man tatsächlich ist, denn Hadoop prüft per default mit dem lokalen Usernamen. Ist an z.B. auf einem System, das Zugriff zu einem Hadoop Cluster hat, mit einem Usernamen eingeloggt der zufällig denselben Namen hat wie der Superuser auf dem Cluster, so kann man mit root Berechtigungen auf das Cluster zugreifen. Um diese Probleme zu verhindern bietet Hadoop die Möglichkeit, LDAP und Kerberos in das Cluster zu integrieren. [7]

4.2 Kompetenz Dokumentenformate

OAuth 2.0, (SOAP)

4.3 Kompetenz Datenbanksysteme

CouchDB, MongoDB mit Ldap

4.4 Kompetenz Datenbankentwurf

-

5 Desaster Recovery

5.1 Kompetenz Dateisysteme

HDFS, GFS Multi master

5.2 Kompetenz Dokumentenformate

-

5.3 Kompetenz Datenbanksysteme

Datenbank Backups, wie oft, wann etc.

5.4 Kompetenz Datenbankentwurf

-

6 Algorithmen und Protokolle

6.1 Kompetenz Dateisysteme

HDFS, GFS API

6.2 Kompetenz Dokumentenformate

Heartbeat Messages Format (JSON, XML)

6.3 Kompetenz Datenbanksysteme

-

6.4 Kompetenz Datenbankentwurf

-

7 Konsistenz und Datenhaltung

7.1 Kompetenz Dateisysteme

Metadaten Replizierung im HDFS, GFS

7.2 Kompetenz Dokumentenformate

-

7.3 Kompetenz Datenbanksysteme

-

7.4 Kompetenz Datenbankentwurf

-

Literaturverzeichnis

[1] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung - Google∗, The Google File System. SOSP 2003. Available at: http://static.googleusercontent.com/media/research.google.com/de//archive/gfs-sosp2003.pdf [last visited 2016-05-29]

[2] Shvachko Konstantin, Kuang Hairong, Radia Sanjay, Chansler Robert. *The Hadoop Distributed File System*. Available at: http://catalogplus.tuwien.ac.at [last visited 2016-05-29]

[3] www.ostep.org. *Sun's Network File System*. Available at: http://pages.cs.wisc.edu/~remzi/OSTEP/dist-nfs.pdf [last visited 2016-03-16]

[4] Apache. *Transparent Encryption in HDFS*. Available at: https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-hdfs/TransparentEncryption.html [last visited 2016-06-01]

[5] Bilal Siddiqui. *Exploring XML Encryption, Part 1*. Available at: https://www.ibm.com/developerworks/library/x-encrypt/ [last visited 2016-06-01]

[6] docs.mongodb.com. *Encryption.* Available at: https://docs.mongodb.com/manual/core/security-encryption/ [last visited 2016-06-01]

[7] Jon Natkins. *Authorization and Authentication in Hadoop.* Available at: http://blog.cloudera.com/blog/2012/03/authorization-and-authentication-in-hadoop/ [last visited 2016-06-01]

Abbildungsverzeichnis

[Abb1] Wikipedia.org. *Google File System Architecture*. Available at: https://upload.wikimedia.org/wikipedia/commons/thumb/c/c3/GoogleFileSystemGFS.svg/2000px-GoogleFileSystemGFS.svg.png [last visited 2016-05-29]

[Abb2] Bilal Siddiqui. *Encrypting the entire file*. Available at: https://www.ibm.com/developerworks/library/x-encrypt/listing2.html [last visited 2016-06-01]

[Abb3] Bilal Siddiqui. *Sample XML file to be encrypted*. Available at: https://www.ibm.com/developerworks/library/x-encrypt/#code1 [last visited 2016-06-01]

[Abb4] Bilal Siddiqui. *Encrypting only the <Payment> element*. Available at: https://www.ibm.com/developerworks/library/x-encrypt/listing3.html [last visited 2016-06-01]

[Abb5] Bilal Siddiqui. *Encrypting only the content in the CardId element*. Available at: https://www.ibm.com/developerworks/library/x-encrypt/listing4.html [last visited 2016-06-01]