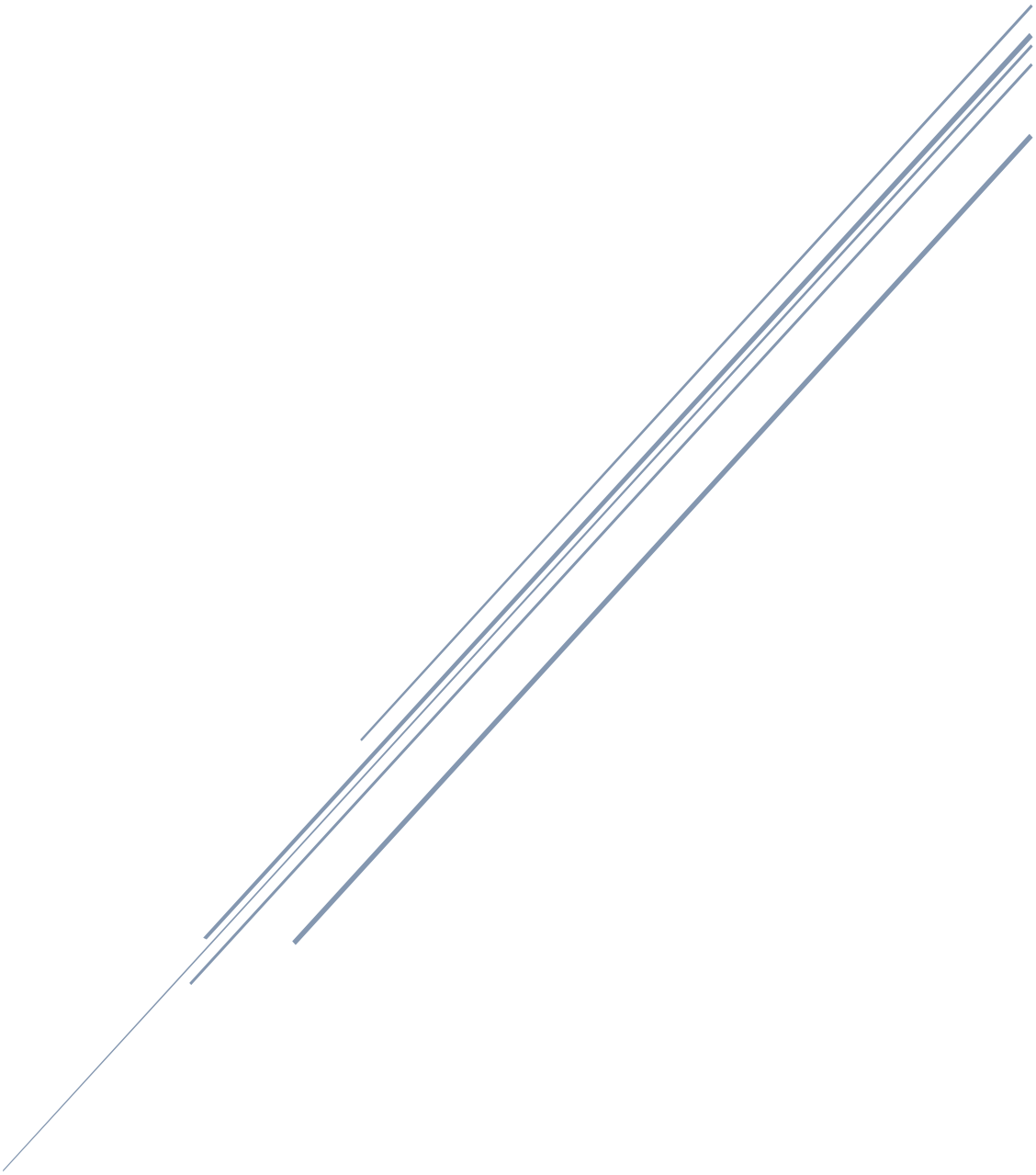
VERTEILTE DATEISYSTEME

Thomas Stedronsky



TGM

Dezentrale Systeme

Inhalt

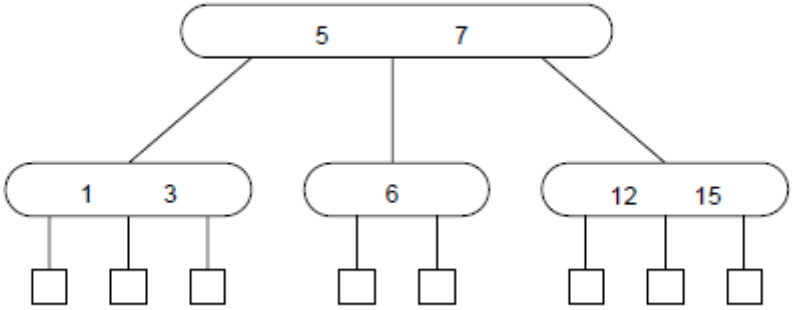
1. Einleitung 2
1.1. Grundlagen 2
1.1.1. B-Baum 2
1.1.2. B+ Baum 2
1.1.3. B* Baum 2
2. Eigenschaften von modernen Dateisystemen 3
2.1. ZFS 3
2.2. BTRFS 3
2.3. ReFS 4
2.4. VFS 4
3. Journaling 4
3.1. Problematik 4
3.2. Funktion von Journaling 5
4. Snapshoting 5
4.1. Motivation 5
4.2. Versionierung 6
4.3. Snapshot-Algorithmus 6
4.4. Umsetzung des Snapshoting-Algorithmus 7
5. Volumenmanagement 8
5.1. Vorteile von Volumenmanagement 8
5.2. Wie funktioniert Volumenmanagement? 8
5.3. LVM 9
5.4. Storage Spaces 10
5.5. ZFS 11
6. Implementierungen von Verteilten Dateisystemen 11
6.1. OCFS2 11
6.2. GlusterFS 12
7. Zusammenfassung 12
8. Abbildungsverzeichnis 14
9. Quellen 14

1. Einleitung

Ein Dateisystem ist eine Ablageorganisation auf einem Datenträger wie einer Festplatte. Den Aufbau kann man sich in Blöcken vorstellen, wobei ein Block mehrere Unterblöcke enthalten kann. Das Ziel eines Dateisystems ist Daten möglichst Ressourcenschonend zu persistieren.

1.1. Grundlagen

1.1.1. B-Baum



[1]

Abbildung 1 B-Baum

Ein B Baum besteht aus Knoten wobei jeder Knoten eine Abspaltung nach links und rechts hat, bei dieser Entscheidung handelt es sich um ein < als der Knoten und > als der Knoten. Ein Baum kann beliebig viele Wurzeln („Kinder“) haben. Jedes Kind hat dieselben Eigenschaften wie ein Eltern Knoten. [2]

1.1.2. B+ Baum

„Das entscheidende Merkmal eines B+-Baums (den man auch „hohlen Baum“ nennt) ist, dass alle Datensätze in den Blättern liegen, die ursprünglichen Charakteristika des normalen B-Baums bleiben jedoch erhalten; die Wege von der Wurzel zu den Blättern stets gleich lang. Da die Blattknoten miteinander verknüpft sind und einer verketteten Liste ähneln, eignen sich B+-Bäume gut zur sequentiellen Abarbeitung.
In den inneren Knoten eines B+-Baums liegen nur noch Referenzschlüssel.
Um einen gesuchten Wert zu finden, muss man nun durch die
Indizes bis zur Blattebene vordringen, wobei die Werte im Indexbereich keine Rolle spielen, da man den vollen Datensatz nur auf der Blattebene findet.“[3]

1.1.3. B* Baum

„Die wesentliche Veränderung gegenüber dem normalen B-Baum ist, dass die Knoten nunmehr zu 2/3 gefüllt sein müssen; beim B-Baum ist bereits eine Belegung von 50% ausreichend.

Um zu einer 2/3 Belegung der Knoten zu gelangen, bedarf es einiger Änderungen an der Splitting-Regelung beim Einfügen; es soll vermieden werden, dass Knoten allzu oft aufgespaltet werden. Der Grundgedanke ist, dass man ein lokales Umverteilungsschema verwendet, um somit die Spaltung so lange zu vermeiden, bis zwei Nachfolgerknoten voll sind. Man teilt dann diese zwei Knoten in drei auf, wobei dann jeder von ihnen zu 66% voll ist. Die Veränderungen am Algorithmus zur Einhaltung der B-Baum Kriterien halten sich in Grenzen; der Vorteil ist aber, dass die Höhe des Baums geringer wird (häufiges Spalten der Knoten das bis zur Wurzel vordringt vergrößert ja die Höhe des Baums) und somit eine schnellere Suche möglich ist.“[3]

2. Eigenschaften von modernen Dateisystemen

2.1. ZFS

Das Zettabyte File System implementiert das Dateisystems des ZFS POSIX Layers. Für die Speicherung von Daten verwendet ZFS Objekte im zpool (ZFS Plattenpool), diese werden von der Data Management Unit zur Verfügung gestellt. Wird nur eine Schreiboperation ausgelöst, wird auch nur eine Transaktion benötigt.

Mit dem ZFS-Dateisystem ist es möglich Snapshots anzulegen. Außerdem ist in diesem Dateisystem ein Volume Manager integriert. Um die Ausfallsicherheit zu steigern bietet ZFS die Möglichkeit von softwarebasiertem RAID. [4]

2.2. BTRFS

BTRFS steht für „B-tree Filesystem“ welches von Oracle implementiert wurde. Seit dem Kernel 2.6.29 zählt es zu den Linux unterstützen Dateisystemen. Wie der Name schon sagt verwendet BTRFS klassische B-Bäume für die Datenspeicherung. Mit dem B-Baum Filesystem können bis zu 16 Millionen Terrabyte persistiert werden. BTRFS weist sehr viele Merkmale von ZFS auf. Unter anderem verwenden beide Copy on Write[1]. [5]

„Btrfs lässt sich ohne zusätzliches Volume-Management oder Software-RAID über mehrere Datenträger hinweg als RAID 0, RAID 1 oder RAID 10 nutzen.“ [5]

„Mit Btrfs bringt der Linux-Kernel ein natives Linux-Dateisystem mit, das dank innovativer Features und Support von bis zu 16 Exabyte Dateisystemgröße gute Chancen hat, zum Linux-Standarddateisystem zu avancieren. Zurzeit eignet es sich allerdings nur für Tests und Spielereien – Heimanwender warten besser, bis die eigene Distribution Btrfs unterstützt.“ [5]

[1] Copy on Write: Geänderte Blöcke werden nicht überschrieben sondern vorerst nur auf einen freien Platz geschrieben. Die Metadaten werden anschließend aktualisiert.

2.3. ReFS

Ist ein Dateisystem welches von Windows als Weiterentwicklung von NTFS für Windows ab Version 8 implementiert wurde. ReFS steht für Resilent File System(„zuverlässiges Dateisystem“). Maßgebliche Features sind unter anderem Metadatenintegrität mit den erforderlichen Prüfsummen, Integritätsdatenströme, Zuverlässigkeit bei Fehlern. [6]

„Zusätzlich übernimmt ReFS die Features und die Semantik von NTFS, einschließlich BitLocker-Verschlüsselung, Zugriffssteuerungslisten für die Sicherheit, USN-Journal, Änderungsbenachrichtigungen, symbolische Links, Abzweigungspunkte, Bereitstellungspunkte, Analysepunkte, Volumesnapshots, Datei-IDs und Oplocks.

Der Zugriff auf ReFS-Daten erfolgt selbstverständlich über die gleichen Client-APIs für den Dateizugriff, mit denen auch der Zugriff auf die heutigen NTFS-Volumes möglich ist.“ [6]

2.4. VFS

Ist eine Abstraktionsschicht, welche sich oberhalb konkreter Dateisysteme befindet. Das File System kann als Layer angesehen werden, dieser Layer stellt eine einheitliche API zur Verfügung um auf verschiedene Dateisysteme zuzugreifen. Es dient sozusagen als allgemeine Schnittstelle und überdeckt lokale Dateisysteme und Zugriffe auf entfernte Dateien und Verzeichnisse. [7]

3. Journaling

Ist eine der wichtigsten Eigenschaften eines modernen Dateisystems. Alle Änderungen werden vor dem eigentlichen Speichervorgang in einen reservierten Speicherbereich zwischengespeichert. Dadurch wird ermöglicht, dass bei einem plötzlichen Ausfall des Systems die Daten in einem konsistenten Zustand erhalten bleiben.

Man unterscheidet Grundsätzlich zwischen Metadaten-Journaling und Full-Journaling. Beim Metadaten-Journaling wird nur die Konsistenz des Dateisystems gewährleistet. Beim Full-Journaling werden alle Dateiinhalte konsistent gehalten. [8]

3.1. Problematik

Ein Dateisystem speichert Informationen, indem es Namen Daten zuordnet. Bei jedem Dateisystem erfordert eine Veränderung der Daten(Erstellen, Umbenennen, Verschieben oder Löschen) Schreibvorgange an mehreren Stellen auf dem Speichermedium. Wird eine Schreiboperation ausgeführt so befindet sich das gesamte Dateisystem nicht mehr in einem konsistenten Zustand. Es wird davon ausgegangen das eine Schreiboperation von einen widerspruchsfreien Zustand in einen neuen konsistenten Zustand übergeht. Wird dieser letzte Schreibvorgang ordnungsgemäß abgeschlossen trifft dies auch zu, allerdings kommt es während des Schreibvorgangs zu einem Systemausfall kann es zu Fehlern kommen, diese dann bei einen Neustart erst überprüft werden müssen. Dieser Test kann übersprungen werden allerdings kann es dann zu Datenverlusten kommen. [8]

3.2. Funktion von Journaling

Mittels Journaling wirkt man dieser Problematik entgegen. Will man Beispielweise eine Datei von Verzeichnis A ins Verzeichnis B verschieben so erfordert das 2 Schreibvorgänge. Einerseits das Löschen der Datei im Verzeichnis A und andererseits muss die Datei in Verzeichnis B hinzugefügt werden. Aber durch Journaling werden diese Änderungen nicht an den Stellen durchgeführt wo sie eigentlich hingehören sondern es gibt einen speziellen Bereich wo diese Änderungen zwischen gespeichert werden. Diesen Zwischenspeicher nennt man Journal. Allerdings durch dieses Journal ist nicht automatisch die Konsistenz des Dateisystems gesichert. Es gibt eine Prüfsumme wo überprüft wird ob alle Vorgänge korrekt durchgeführt wurden. In dem konkreten Fall von oben muss die Prüfsumme 2 ergeben.

Zu den Journaling Dateisystemen zählen unter anderem die bereits erwähnten NTFS und BTRFS. [8]

4. Snapshoting

Ein Snapshot ist eine Momentaufnahme eines Zustands zu einen gewissen Zeitpunkt. Beim Snapshoting unterscheidet man zwischen folgenden Technologien Copy on Write und Redirect on Write. [9] [12]

Bei Copy on Write wird nur eine neue Kopie erzeugt, wenn ein Benutzer eine Kopie geändert hat. Durch dieses Verfahren wird die Kopien Anzahl verringert. Beim ersten Snapshot werden außerdem die Metadaten mit gespeichert. [10][12]

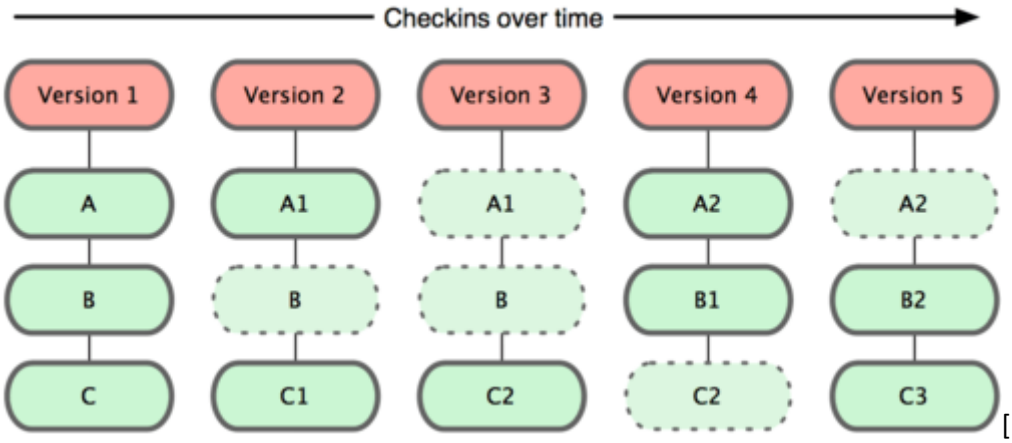
Redirect on Write hat gegenüber zu Copy On Write eine bessere Performance und ist speziell für langfristige Snapshot Speicherungen geeignet. Bei diesem Verfahren werden keine kompletten Kopien gespeichert, sondern lediglich die einzelnen Änderungen in den Datenblöcken. Änderungen werden automatisch in den Speicherbereich des Snaphots umgeleitet. [11][12]

4.1. Motivation

Die Motivation um Snapshoting zu betreiben ist diese einzelne Momente festzuhalten. Beispielsweise vor einem Update kann ein Snapshot durchgeführt werden um bei eventuell auftretenden Fehlern den Zustand vor dem Update wiederherstellen zu können. Allgemein werden Snapshots vorwiegend zur Datensicherung eingesetzt.

4.2. Versionierung

Mit Snapshoting kann auch Versionierung betrieben werden. Wenn man zum Beispiel Git hernimmt. Git betrachtet die gespeicherten Daten als Mini Dateisystem. Git speichert bei jedem Commit den momentanen Zustand sämtlicher Dateien und speichert eine Referenz auf diese Daten. Git legt aber lediglich Verknüpfungen mit Änderungen auf vorherige Versionen an Zwecks Performance. [13]



[13]

Abbildung 2 Git example

4.3. Snapshot-Algorithmus

Dieser wurde von Leslie Lamport und K. Mani Chandy entwickelt und ist auch nach den beiden Entwicklern benannt, deswegen ist der Snapshot-Algorithmus auch unter Chandy-Lamport-Algorithmus bekannt. *„Laut Chandy und Maport sollte ein solcher Snapshot-Algorithmus zwar in der Lage sein, Nachrichten zu versenden und lokale Aktionen in jedem Prozess anzustoßen, jedoch sollte er dabei niemals die darunter liegenden Berechnungn beeinflussen (bis auf wenige Ausnahmen)“* [14]

Der Algorithmus läuft wie folgt ab:

(a) „Ein Prozess entscheidet sich, den Algorithmus zu starten und
   a. Schreibt den lokalen Zustand nieder
   b. Sendet einen „Marker“ auf allen ausgehenden Kanälen (nachdem der Zustand aufgenommen wurde und bevor weitere Nachrichten gesendet werden)

(b) Empfängt ein Prozess einen „Marker“ und hat er seinen lokalen Zustand noch nicht niedergeschrieben
   a. Schreibt er den lokalen Zustand nieder
   b. Sendet einen „Marker“ auf allen ausgehenden Kanälen (nachdem der Zustand aufgenommen wurde und bevor weitere Nachrichten gesendet werden)
   c. Startet die Aufnahme für alle eingehenden Kanäle, auf denen noch kein „Marker“ empfangen wurden

(c) Empfängt ein Prozess einen weiteren „Marker“
   a. Stoppt er die Aufnahme für den entsprechenden Kanal und schreibt die aufgenommenen Nachrichten nieder.“[13]

4.4. Umsetzung des Snapshoting-Algorithmus

Es gibt noch eine weitere Variante des Snapshot-Algorithmus, diese Variante wurde von Shah und Toueg entwickelt und sollte dazu dienen auch im Falle eines Ausfalls die Konsistenz der Daten zu sichern.

Bei dieser Variante wurden wieder unidirektionale Kommunikationskanäle verwendet. Chandy und Lamport nahmen an, dass alle Kanäle fehlerfrei arbeiten und es keine Totalausfälle gibt. Mit dieser Weiterentwicklung war es also möglich kleine Ausfälle zu kompensieren indem sie den Betrieb komplett einstellten. Außerdem konnten Kanäle wiederbelebt werden. [13]

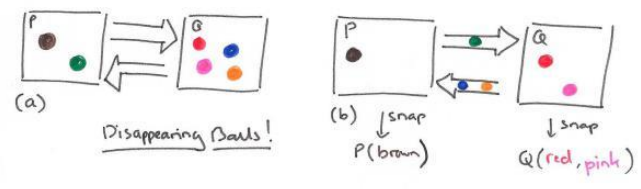
Beispiel:

„Initiator i:

```
inst_i = inst_i + 1;
e_i = das letzte Ereignis ;
Prozessstatus = aktueller Status ;
Sende ( signal_i , inst_i ) an alle anderen Prozesse ;
Starte fuer jeden Prozess einen Timer ;
```

Wird eine Nachricht m empfangen:

```
if(m == ( signal_k , inst_k )) or (m == ( msg_k , inst_k )):
      if inst_k > inst_j :
            inst_j = inst_k
            e_j = das letzte Ereignis ;
            Prozessstatus = aktueller Status ;
            Status von Kanal (k,j) = leere Sequenz ;
            Sende ( signal_j , inst_j ) an alle anderen
                  Prozesse ;
            Starte fuer jeden Prozess einen Timer ;
      else if inst_k == inst_k :
            Status von Kanal (k,j) = Sequenz von
                  Nachrichten die nach e_j und
                  vor m empfangen wurden ;
            Breche Timer fuer k ab;
      else if inst_k < inst_j :
            noop ;
      fi;
else if m == (time - out_k , inst_j ):
      Status von Kanal (k,j) = Sequenz von Nachrichten
      die nach e_j und vor m empfangen wurden ;
      Wirf alle Nachrichten von k weg , bis
            ( signal_k , inst_k ) oder ( msg_k , inst_k )
            empfangen wird mit inst_k >= inst_j ;
fi;“[13]
```



[15]

Abbildung 3 snapshot example

5. Volumenmanagement

Die höchste Ebene im Dateisystem ist das Volumen, das Dateisystem befindet sich auf dem Volumen, ein Volumen enthält mindestens eine Partition (logische Unterteilung des Speichermediums). Es wird unterschieden zwischen Daten die sich nur auf einer Partition befinden, diese nennen sich „simple volume“ und Daten die auf mehr als einer Partition vorhanden sind, diese werden „multipartition volume“ genannt. [16]

Volumen werden von sogenannten Volume Managern implementiert.

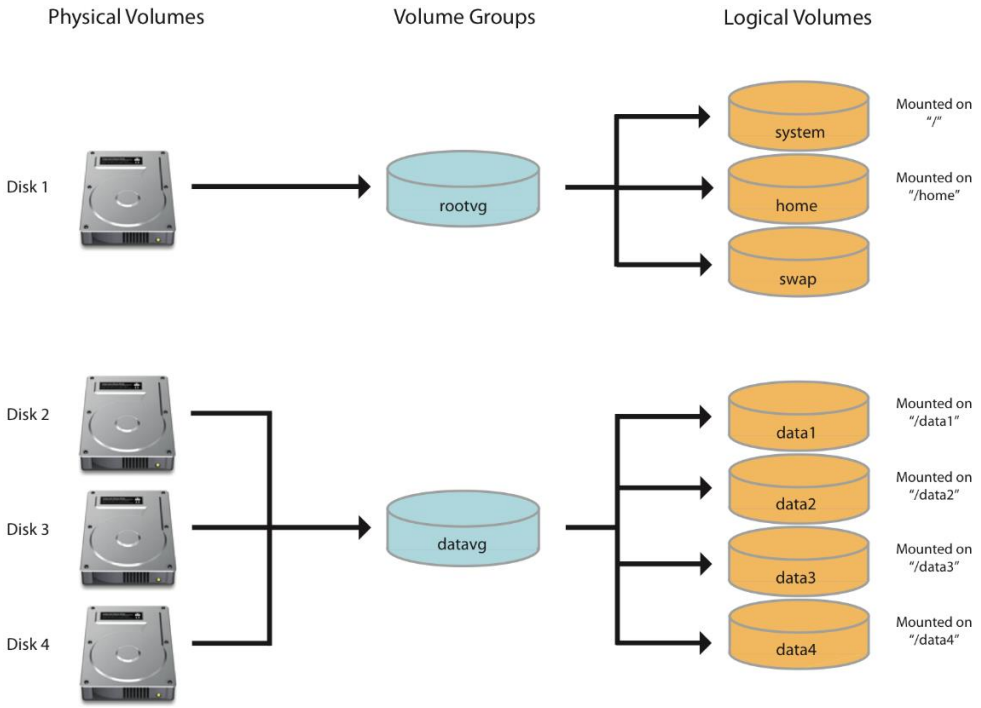
5.1. Vorteile von Volumenmanagement

Einer der größten Vorteile ist es, dass im laufenden Betrieb Volumen (Speicherplatz) angelegt, verkleinert und vergrößert werden können. Außerdem wird durch ein richtiges Volumenmanagement vermieden, dass falsche Partitionierungen auftreten. Mehrere Festplatten können zu einem Volume zusammengesetzt werden und somit als ein Medium agieren.

Mit einem Volume Management System ist man unabhängiger von den zu Verfügung gestellten Speicher, da er beliebig erweitert werden kann. [16]

5.2. Wie funktioniert Volumenmanagement?

Ein Volumenmanagement arbeitet mit sogenannten Volume Groups in solchen Volume Groups befindet sich mindestens ein physikalischer Speicher. Aus diesem Volume Groups können dann weiterte Logical Volumes erstellt werden. Erklärt anhand von LVM siehe 5.3.



[17]

Abbildung 4 volume management

5.3. LVM

LVM steht für Logical Volume Manager, mit diesem Volume Manager können wie bereits erwähnt und in Abbildung 4 beschrieben physikalische Speicher zu einer Volume Group zusammengefasst werden und diese können dann wieder in Logical Volumes unterteilt werden.

Es wurden bereits die Begriffe Physical Volume, Volume Group und Logical Volume genannt.

„Physical Extent

Das sog. Physical Extent (PE) ist die kleinste mögliche Speichergröße in einem LVM. Das PE ist vergleichbar mit den Sektoren von normalen Festplatten bzw. Partitionen. Es ist nicht möglich, die Größe der PEs nachträglich zu verändern. Dies ist aber auch nicht nötig, da die Anzahl der PEs seit LVM2 unbegrenzt ist, die Standardgröße beträgt 4 MByte.

Physical Volume

Ein Physical Volume (PV) ist der eigentliche Datenspeicher eines LVMs. Ein PV kann eine Partition oder ein ganzes Laufwerk sein. Es ist auch möglich, sogenannte "Loop-Dateien" zu verwenden.

Volume Group

Eine Volume Group (VG) ist eine Art Container für mindestens ein PV.

Logical Volume

Ein Logical Volume (LV oder auch Volume) ist im LVM Kontext für den Anwender eine Art Partition innerhalb einer Volume Group. Es ist nicht möglich Logical Volumes außerhalb einer VG zu erstellen. Das LV kann wie eine normale Partition verwendet werden, d.h. es können Dateisysteme darauf angelegt werden und es kann gemounted werden.“

Das Arbeiten mit LVM unterscheidet sich recht grundlegend von der Arbeit mit normalen Partionen. Versuchen Sie sich also etwas von Ihren vorhanden Vorstellungen vom Umgang mit Festplatten zu lösen.“[18]

„Am Anfang steht natürlich auch beim LVM die real existierende Festplatte oder die Festplatten Partition. Sie wird mit Physical Volume (PV) bezeichnet. Diese wird zu Beginn in eine Volume Group (VG) aufgenommen, dies ist ein Pool des gesamten zur Verfügung stehenden Speicherplatzes. Aus diesem Pool werden nun logische Volumes (LV) nach Bedarf erzeugt. Das Betriebsystem greift nun auf diese logischen Partitionen (LV) anstelle der reellen Partitionen (PV) zu. Der LVM hat also eine zusätzliche Ebene zwischen der Festplatte und der Ein-/Ausgabe des Linux-Kernels geschaffen. Dies ermöglicht die Definition von logischen Volumes, (fast) unabhängig von den zu Grunde liegenden physikalischen Volumes. Mehrere Platten können zu einer grossen logischen Partition zusammengefasst werden.

Der LVM erlaubt wesentliche Eingriffe, wie hinzufügen von PVs und erweitern von LVs sogar im laufenden System.“[19]

Prinzipiell ist es im laufenden Betrieb unwichtig wo genau die Daten auf dem Physical Volume liegen, der Logical Volume Manager regelt dies automatisch. Es kann aber auch mit entsprechender Konfiguration bestimmt werden auf welchen physikalischen Speicher bevorzugt zugegriffen wird Zwecks Performance. [19]

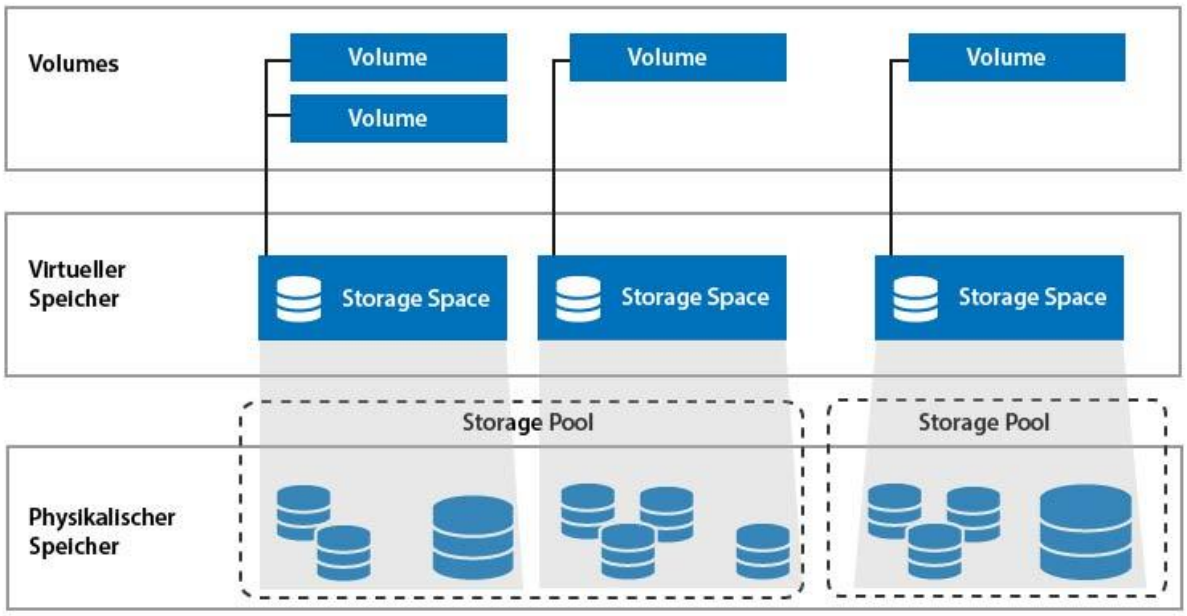
Es gibt eine Weiterentwicklung von LVM, dieses LVM2 unterstützt auch RAID 5 und thin-provisioning[1].

5.4. Storage Spaces

Ist ein von Windows entwickelter Volume Manager. Dieser arbeitet ähnlich wie LVM. Mit Speicherpools, wo physikalische Speicher zu einem virtuellen Speicher zusammengesetzt werden können. Das hinzufügen und entfernen von zusätzlichen Speicher ist voll automatisch, kann allerdings durch Power Shell Kommandos gesteuert werden. Ein Speicherpool ist mit 4096 TB begrenzt. Allerdings können beliebig viele Speicherpools erstellt werden.

Aus diesem Speicherpool kann ich dann meinen eigentlichen Storage Space erstellen, dieser Storage Space kann dann wieder in einzelne Volumes unterteilt werden. [20]

Auf diesen Volumes kann dann ein Dateisystem bzw. Betriebssystem festgelegt werden. Für die Ausfallsicherheit stehen RAID 1 und RAID 5 zu Verfügung. Es ist auch möglich verschiedenen Speicherarten im Storage Pool zu verwenden. [21]



[21]

Abbildung 5 storage space

[1] thin-provisioning: Gibt vor mehr Speicher zu besitzen als wirklich vorhanden.

5.5. ZFS

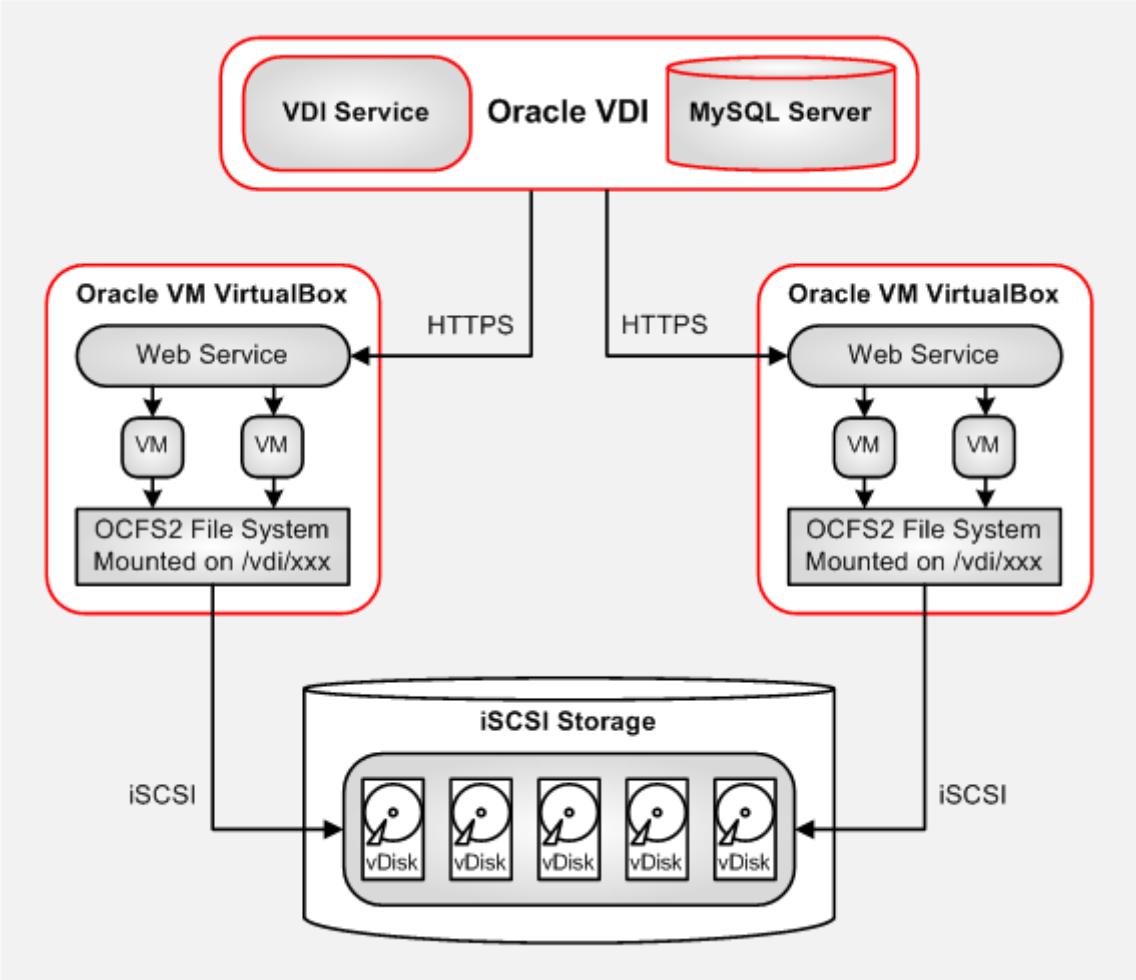
Im Bereits erwähnten Dateisystem ZFS ist bereits ein Volume Manager integriert. Dieser Manager funktioniert ähnlich wie LVM und Storage Spaces. Es werden wieder aus physikalischen Speicher Pools gebildet, die dynamisch verändert werden können. Außerdem ist es möglich dedizierte Block Devices einzeln zu Verfügung zu stellen.

Für die Ausfallsicherheit unterstützt ZFS alle verschiedenen RAID Systeme, die je nach Anforderungen verwendet werden können. [4]

6. Implementierungen von Verteilten Dateisystemen

6.1. OCFS2

Ist ein Open Source Cluster Dateisystem, welches Zugriff auf ein Shared Storage ermöglicht.



[22]

Abbildung 6 OCFS2

In Abbildung 6 befindet sich auf der Virtual Box VM ein OCFS2 Filesystem, das auf einen iSCSI Storage zugreift.

6.2. GlusterFS

Ist ein verteiltes Dateisystem, das mehrere Server mit Speicherelementen als einheitliches Dateisystem angesehen werden kann. Diese Server Nodes(können auch als Cluster Nodes bezeichnet werden) sind mittels einer Client-Server Architektur miteinander verbunden.

Es gibt verschiedene Betriebsmodi:

Standalone Storage (NFS Like)
ein einzelner Server.

Distributed Storage
Mehrere Server, die Daten werden verteilt gespeichert.

Replicated Storage
Die Daten werden untereinander auf mehreren Servern gespiegelt.

Distributed Replicated Storage
Daten werden auf mehreren Server verteilt und gespiegelt.

Striped Storage
Daten werden gestripped um Performance zu steigern(RAID 0).

Cloud/HPC Storage
Wie Distributed Storage

NFS Like Standalone Storage Server-2
Wie Standalone allerdings werden mehr als nur ein Dateisystem bereitgestellt.

Aggregating Three Storage Servers with Unify
3 Server, die ein einheitliches Dateisystem mittels Unify haben. [23]

7. Zusammenfassung

Jedes Dateisystem hat besondere Eigenschaften, um das perfekte Dateisystem zu finden müssen die Anforderungen mit den Funktionen des Dateisystems abgeglichen werden. Die modernen Dateisysteme wie ZFS, BTRFS, ReFS und VFS bieten alle derzeit aktuellen Funktionen mit sich.

Eines der wichtigsten Eigenschaften jedes Dateisystems ist das Journaling. Die Problematik ist hierbei die Datenkonsistenz(Lost Update Problem). Mittels Journaling das Daten im sogenannten Journal zwischenspeichert wirkt diesem Problem entgegen.

Zu Datensicherung bzw. Momentaufnahmen Speicherung gibt es die Funktion Snapshoting. Mithilfe des Algorithmus von Lamport und Chandy ist es möglich eine Momentaufnahme von meinem System zu machen und jeder Zeit wieder abzurufen. So kann unter anderem Datensicherung betrieben werden.

Mit dem Volumenmanagement ist möglich seine Speicherkapazität variabel zu gestalten. Mit sogenannten virtuellen Speicher können mehrere Festplatten (physikalische Speicher) zu einem Volume zusammengesetzt werden. Der berühmteste Volume Manager ist LVM. Es gibt aber auch zahlreiche andere unter anderem Storage Spaces von Windows und der integrierte Volume Manager im Dateisystem ZFS.

Zwei wichtige Implementierungen von verteilten Dateisystem sind OCFS2 und GlusterFS.

8. Abbildungsverzeichnis

Abbildung 1 B-Baum 2
Abbildung 2 Git example 6
Abbildung 3 snapshot example 7
Abbildung 4 volume management 8
Abbildung 5 storage space 10
Abbildung 6 OCFS2 11

9. Quellen

[1] Prof. Th. Ottmann, Algorithmen und Datenstrukturen, https://electures.informatik.uni-freiburg.de/portal/download/59/6149/INFO2-19-B_Baeume_4up.pdf, zuletzt besucht am 01.03.2016

[2] Bayer, B-Bäume, http://wwwbayer.in.tum.de/lehre/WS2001/HSEM-bayer/BTreesAusarbeitung.pdf, zuletzt besucht am 02.03.2016

[3] Jan Ehrlich, B- Bäume, http://www.informatik-forum.at/attachment.php?attachmentid=948, zuletzt besucht am 02.03.2016

[4] Ulrich Gräf, Oliver Frommel, ZFS - Snapshots, RAID und Datensicherheit, http://www.admin-magazin.de/Online-Artikel/ZFS-Snapshots-RAID-und-Datensicherheit, zuletzt besucht am 01.03.2016

[5] Marcel Hilzinger, Btrfs-Grundlagen ADMIN-Magazin, http://www.admin-magazin.de/Das-Heft/2009/04/Das-neue-Linux-Dateisystem-Btrfs-im-Detail, zuletzt besucht am 01.03.2016

[6] Steven Sinofsky, Hier entsteht die neue Generation von Dateisystemen für Windows: ReFS, https://blogs.msdn.microsoft.com/b8_de/2012/01/20/hier-entsteht-die-neue-generation-von-dateisystemen-fr-windows-refs/, zuletzt besucht am 01.03.2016

[7] Alessandro Rubini, The "Virtual File System" in Linux, http://www.linux.it/~rubini/docs/vfs/vfs.html, zuletzt besucht am 01.03.2016

[8]Frank, Bruns, Hagedorn, Journaling Dateisystem, http://www.selflinux.org/selflinux/html/dateisysteme_journaling.html, zuletzt besucht am 02.03.2016

[9] snapshot, http://www.itwissen.info/definition/lexikon/snapshot-Momentaufnahme.html, zuletzt besucht am 02.03.2016

[10] Copy on Write, http://www.itwissen.info/definition/lexikon/Copy-on-Write-copy-on-write-COW.html, zuletzt besucht am 02.03.2016

[11] Redirect on Write, http://www.itwissen.info/definition/lexikon/Redirect-on-Write-redirect-on-write-ROW.html, zuletzt besucht am 02.03.2016

[12] Appuswamy, von moolenbroeck, Tannenbaum, Flexible, Modular File Volume Virtualization in Loris, May 2011

[13] Los geht's - Git Grundlagen, https://git-scm.com/book/de/v1/Los-geht%E2%80%99s-Git-Grundlagen, zuletzt besucht am 02.03.2016

[14] Stefan Kreidel, Snapshot in Scalaris Ein Algorithmus zum Speichern konsistenter Zustände in verteilten Hashtabellen, Kapitel 3.1.2

[15] Adrian Colyer, Distributed Snapshots: Determining Global States of Distributed Systems, http://blog.acolyer.org/2015/04/22/distributed-snapshots-determining-global-states-of-distributed-systems/, zuletzt besucht am 02.03.2016

[16] Volume Management, https://msdn.microsoft.com/de-de/library/windows/desktop/aa365728%28v=vs.85%29.aspx, zuletzt besucht am 03.03.2016

[17] Helios, Volume Management, http://www.helios.de/support/manuals/vsa-e/volume-management.html, zuletzt besucht am 03.03.2016

[18] Logical Volume Manager, https://wiki.ubuntuusers.de/Logical_Volume_Manager/, zuletzt besucht am 03.03.2016

[19] Richard Heider, Logical Volume Manager (LVM) Howto - deutsche Version: Grundlagen, http://www.linuxhaven.de/dlhp/HOWTO-test/DE-LVM-HOWTO-1.html, zuletzt besucht am 03.03.2016

[20] Storage Spaces: FAQ, http://windows.microsoft.com/en-us/windows-8/storage-spaces-pools, zuletzt besucht am 03.03.2016

[21] Andreas Scheidmeir,Storage Spaces Überblick, https://www.escde.net/storage-spaces-ueberblick/, zuletzt besucht am 03.03.2016

[22] Storage, https://docs.oracle.com/cd/E36500_01/E36503/html/storage.html, zuletzt besucht am 03.03.2016

[23] Florian Wiessner, Verteiltes Dateisystem GlusterFS, http://www.netz-guru.de/2009/01/15/clusterdateisystem-glusterfs/, zuletzt besucht am 03.03.2016

Inhalt

1. Implementierungen 2
1.1. Sun Network File System 2
1.1.1. Die Basics 2
1.1.2. Ein Blick ins Sun Network File System (NFS) 2
1.1.3. NFS Protocol (version 2 a.k.a. NFSv2) 2
1.1.4. Statelessness 2
1.2. Google File System 4
1.2.1. Anforderungen 4
1.2.2. Zugriff 4
1.2.3. Architektur 5
1.2.4. Chunkgröße 6
1.2.5. Master 6
1.3. Hadoop Distributed File System 8
1.3.1. Architektur 8
2. Stärken und Schwächen 9
2.1. GFS 9
2.1.1. Stärken 9
2.1.2. Schwächen 10
2.2. Hadoop Distributed File System 10
2.2.1. Indizierung der inodes 10
2.2.3. Datenvollständigkeit 10
2.3. Sun Network File System 11
2.3.1. Stärken 11
2.3.2. Schwächen 11
3. Einsatzgebiete 12
3.1. GFS 12
3.2. NFS 12
3.3. HDFS 12
Literaturverzeichnis 13
Primäre Quellen 13

1. Implementierungen

1.1. Sun Network File System

1.1.1. Die Basics

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. [8]

1.1.2. Ein Blick ins Sun Network File System (NFS)

Anstatt ein fertiges System oder ein Konzept zur Implementierung eines verteilten Dateisystems, ist das Sun Network File System (NFS) nur ein Protokoll, welches beschreibt wie Client und Server miteinander zu kommunizieren haben. Das NFS Protocol. Von diesem gibt es mittlerweile mehrere Versionen, wobei im nächsten Abschnitt (1.1.3) Version 2 näher erläutert wird. [8]

1.1.3. NFS Protocol (version 2 a.k.a. NFSv2)

Der Fokus in diesem Protokoll liegt beim *„simple and fast server crash recovery“*. Bei einer multiple-Client und single-server Architektur, spielt der Server eine wichtige Rolle. Jede Minute die der Server nicht erreichbar ist, macht die Clients unglücklich und vor allem unproduktiv. Das ganze System ist also von einem Server abhängig und somit macht das *„simple and fast server crash recovery“* in dieser Architektur sehr viel Sinn. [8]

1.1.4. Statelessness

Der zuvor erwähnte Fokus auf *„simple and hast server crash recovery“*, wird im NFSv2 dadurch erreicht, mit stateless zu arbeiten d.h. in diesem Fall, dass der Server keinerlei Aktivitäten verfolgt oder vereinfacht gesagt der Server nichts über seine Clients zwischenspeichert. In einem Request, müssen daher *alle Informationen* enthalten sein, um einen Request erfolgreich abzuschließen.

Zur vereinfachten Darstellung, stellt man sich ein **stateful** Protokoll vor, bei welchem der system call `open()` aufgerufen wird. Übergibt man dieser Funktion einen Pfadnamen, so liefert sie einen so genannten *file descriptor* (ein Integer) zurück, wie im Folgenden Codebeispiel zu sehen ist:

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);          // read MAX bytes from foo
...
read(fd, buffer, MAX);          // read MAX bytes from foo
close(fd);                      // close file
```

Abbildung 1 Client Code: Lesen einer Datei [8]

Der client sendet nun eine Nachricht an den Server mit dem Inhalt „öffne die Datei ‚foo' und gib mir den *file descriptor* zurück". Nun öffnet der Server diese Datei und gibt dem Client, den angeforderten *file descriptor* zurück. Nun verwendet der Client diesen um von dieser Datei lesen zu können. Hierzu schickt er eine Nachricht an den Server mit dem Inhalt „lies n bytes von der Datei zu der dieser *file descriptor* passt".

Der *file descriptor* ist in diesem Fall ein so genannter **shared state** zwischen dem Client und dem Server und das verkompliziert das *crash recovery*. Stellt man sich nun vor, dass der Server abstürzt, nachdem der erste Lesevorgang abgeschlossen aber bevor der zweite beendet ist, so kommt man zu einem Problem. Schickt der Client nun seinen zweiten read Request an den Server, nachdem dieser wieder hochgefahren ist, so weiß der Server nicht mehr zu welcher Datei dieser file descriptor passt. Diese Information hat der Server in seinem Arbeitsspeicher zwischengespeichert und beim Absturz verloren. Um diese Situation zu vermeiden müsste der Client, alle benötigten Daten die der Server braucht, bei sich zwischenspeichern.

Das Ganze wird sogar noch schlimmer, wenn man es von der Server Seite aus betrachtet. Wenn z.B. ein client eine Datei mit `open()` öffnet und dann abstürzt weiß der Server nicht wann er die Datei wieder schließen darf. Normalerweise ruft der Client die Funktion `close()` auf, um eine Datei nach Gebrauch wieder zu schließen. Wenn also der Client abstürzt, würde er einen derartigen Close-Befehl niemals erhalten.

Aus diesen Gründen hat man sich entschieden, stateless vorzugehen d.h. jeder Request eines Clients enthält alle benötigten Informationen, um diesen abzuschließen. [8]

1.2. Google File System

1.2.1. Anforderungen

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. [6]

1.2.2. Zugriff

Dateien werden hierarchisch in Verzeichnissen angeordnet, sowie mit einem eindeutigen Pfad identifiziert. Das GFS stellt ein gewöhnliches Filesystem-Interface zur Verfügung, welches Operationen wie *create*, *delete*, *open*, *close*, *read* und *write* zur Verfügung stellt.

Zusätzlich dazu, werden noch 2 zusätzliche Operationen *snapshot* und *record append* angeboten. Die Operation *snapshot* kopiert eine Datei oder ein Verzeichnis, ohne das System großartig zu belasten. *Record append* erlaubt es mehreren Clients, gleichzeitig Daten an dieselbe Datei anzuhängen, während die Atomarität bei jedem Schreibzugriff gewährleistet wird. [6]

1.2.3. Architektur

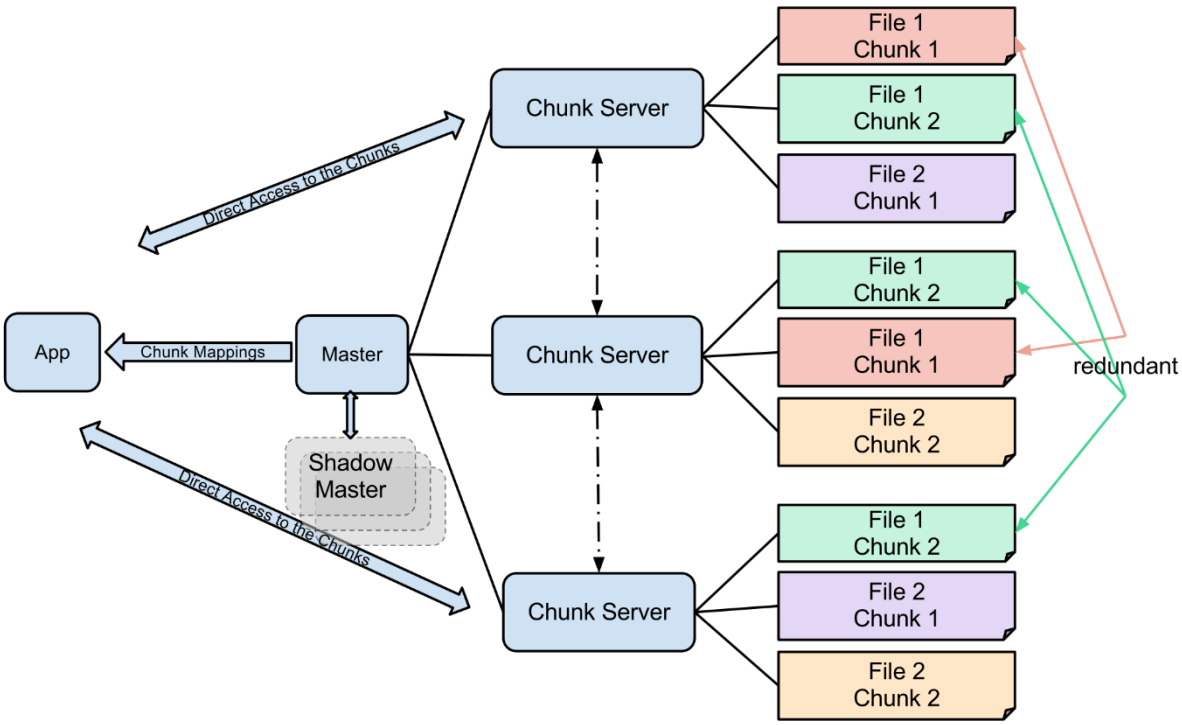


Abbildung 2 GFS Architecture [7]

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]

1.2.4. Chunkgröße

Die Größe eines Chunks liegt bei 64MB, was einige Vor- als auch Nachteile mit sich bringt. Einerseits ist sie ausschlaggebend, wie oft ein Client mit dem Master kommunizieren muss, denn das mehrmalige Lesen und Schreiben auf denselben Chunk, benötigt nur einen Request beim Master.

Ein weiterer Vorteil, ist die geringere Speicherung von Metadaten auf dem Master. Das erlaubt es dem Master nämlich, alle Metadaten im Arbeitsspeicher zu belassen und damit schnellere Zugriffszeiten zu erreichen.

Andererseits bringt ein größerer Chunk, unter den vielen Vorteilen, auch einen Nachteil mit sich. Besteht eine Datei beispielsweise aus nur wenigen oder sogar einem Chunk und mehrere Clients greifen gleichzeitig auf diese Datei zu, so könnte es dazu kommen, dass ein Server zu einem so genannten „Hot Spot“ wird. Greifen viele Clients gleichzeitig auf eine Datei zu, die aus nur einem Chunk besteht, welcher auf nur einem Chunkserver persistiert ist, so könnte der Server auf dem der Chunk liegt, überlastet werden. Dieses Problem kann jedoch gelöst werden, durch eine vermehrte Replikation solcher Dateien auf mehreren Chunkservern. [6]

1.2.5. Master

Der Master speichert 3 Typen von Metadaten, die Datei- und Chunk Namen, das Mapping von Dateien zu Chunks und den Standort der Chunk Replikationen. Alle Metadaten werden auf dem Master im Arbeitsspeicher festgehalten wobei die ersten beiden, die Datei- und Chunk Namen und das Datei-zu-Chunk Mapping, zusätzlich persistiert werden.

Die Standorte der Chunks, werden nicht persistiert, sondern jedes Mal abgefragt entweder beim Start des Masters oder wenn ein neuer Chunkserver zum Cluster hinzugefügt wird. [6]

1.2.6. Lese- und Schreibzugriffe auf einen Chunk

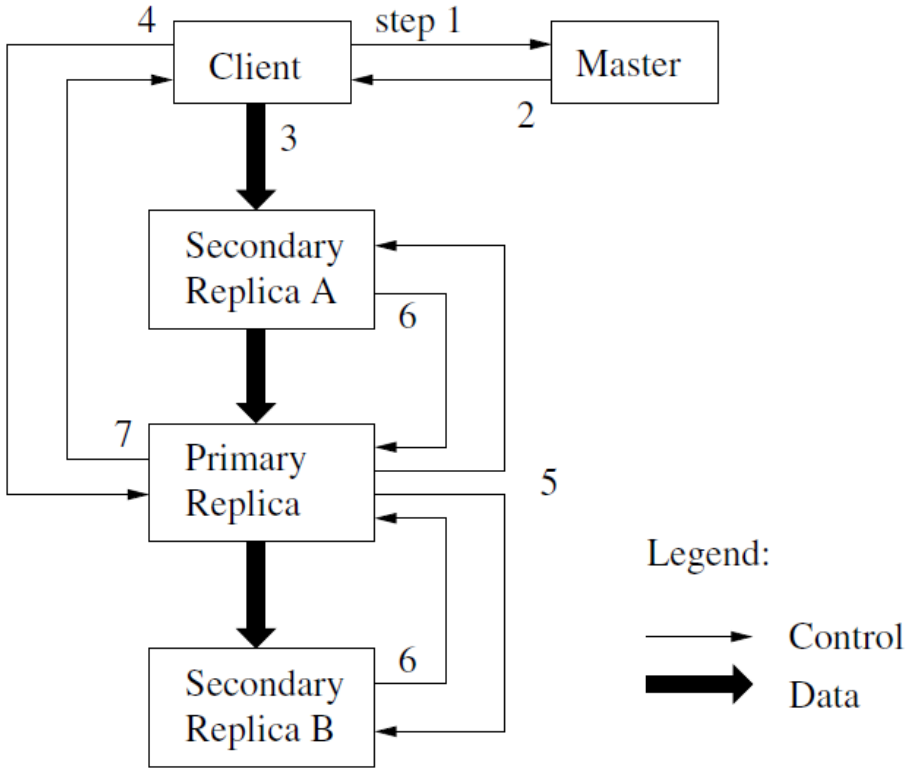


Abbildung 3 Schreibzugriffe und Datenstrom [6]

Obige Abbildung zeigt ein Beispiel für einen Schreibzugriff auf einen beliebigen Chunk. Alle in der Abbilung nummerierten Schritte von 1-6, werden im nachfolgenden genauer erklärt.

1. Der Client fragt den Master, auf welchen Chunkservern der Chunk und seine Replikationen gespeichert sind.
2. Der Master antwortet mit einem so genannten Lease (Lese- und Schreibrecht) für den Chunk (Primary) und den Standorten seiner Replikationen (Secondary). Auf dem Client werden diese Informationen zwischengespeichert für zukünftige Lese- oder Schreibvorgänge, um ständige Anfragen an den Master zu verhindern. Eine erneute Anfrage an den Master ist nur notwendig, wenn der Primary nicht mehr erreichbar ist oder der Lease abgelaufen ist.
3. Nun werden die eigentlichen Daten vom Client auf alle Replica gepusht und dort in einem so genannten LRU Buffer cache solange gespeichert bis diese benutzt werden oder veralten. Interessant ist hier, dass der Data Flow vom Control Flow getrennt ist und der Data Flow somit optimal an die verwendete Netzwerktopologie angepasst werden kann.
4. Haben alle Replica bestätigt, dass sie alle Daten erhalten haben, sendet der Client einen Write Request an den Primary. In so einem Request, werden alle Daten identifiziert die zuvor auf alle Replikationen gepusht worden sind. Der Primary weist diesen Write Requests (können auch mehrere von mehreren Clients sein) nun eine Reihenfolge zu und schreibt diese auf sich selbst.
5. Der Primary leitet diese Write Requests nun an alle Replica weiter, welche diese nun in der Reihenfolge schreiben, die der Master zugewiesen hat.
6. Die Secondaries antworten dem Primary nun, dass der Schreibvorgang erfolgreich ausgeführt worden ist oder ob ein Fehler aufgetreten ist.

7. Der Primary antwortet nun dem Client und berichtet ihm ob der Schreibvorgang auf allen Replicas erfolgreich war oder Fehler aufgetreten sind (Client erfährt auch welche Fehler das waren und wo sie aufgetreten sind).

[6]

1.3. Hadoop Distributed File System

Das Hadoop Distributed File System (HDFS) ist dem Google File System sehr ähnlich, jedoch gibt es ein paar kleine Unterschiede und auch der Fokus liegt woanders. Das HDFS wurde ursprünglich von 4 Yahoo! Mitarbeitern, *Konstantin Shvachko*, *Hairong Kuang*, *Sanjay Radia* und *Robert Chansler* entwickelt und wurde vom GFS inspiriert. [5]

1.3.1. Architektur

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. [5]

1.3.2. Namespace Images und Journals

Namespace Images sind im HDFS Dateien, die bestimmte Metadaten zum Dateisystem beinhalten, wie zum Beispiel die Struktur wie bestimmte Applikationen ihre Daten anordnen (Dateien und Ordner). Persistierte Versionen von solchen Images nennt man, wie schon in Sektion 1.3.1 erwähnt, Checkpoints. Diese werden niemals verändert, sondern nur gänzlich durch den CheckpointNode ersetzt, wie in der nächsten Sektion genauer beschrieben wird. Ein Journal beinhaltet alle Änderungen an einem Image, welches persistiert werden muss. Bei jeder Client Transaktion, werden nun alle Änderungen die der Client an einem Image vornehmen will, im Journal geloggt und synchronisiert, bevor eine Transaktion committet wird.

Fehlt ein Checkpoint oder ein dazugehöriges Journal, ist die Information zum Namespace zum Teil oder gänzlich verloren. Um das zu vermeiden gibt es die Möglichkeit Checkpoints und Journals mehrfach auf unterschiedlichen Ordnern, die auf unterschiedlichen Volumes liegen, zu speichern wobei einer dieser Ordner auf einem NFS Server liegt. Dies verhindert einen Datenverlust nicht nur beim Ausfall bestimmter Volumes, sondern auch beim Ausfall des ganzen Nodes. [5]

1.3.3. CheckpointNode

Zusätzlich zur primären Rolle des NameNodes, Client Requests zu verarbeiten, kann diesem auch noch eine der beiden Rollen *CheckpointNode* oder *BackupNode* zugewiesen werden, wobei eine Rolle beim Start des NameNodes festgelegt wird. Ein CheckpointNode läuft üblicherweise auf einem anderen Host, wie der NameNode.

Ein CheckpointNode lädt den aktuellen Checkpoint und die dazugehörigen Journals herunter, merged diese zusammen, leert das Journal und schickt sie zurück zum NameNode. Periodisch Checkpoints zu erzeugen, verhindert die Korruption eines Journals, da das Risiko, dass ein Journal korrupt wird, mit zunehmender Größe des Journals steigt. Zusätzlich verlangsamt ein großes Journal den Start des NameNodes. Bei einem großen Cluster kann es durchaus Stunden dauern, ein Journal, das Daten einer ganzen Woche beinhaltet, abzuarbeiten.

1.3.4. BackupNode

TODO wenn zu wenig Seiten

2. Stärken und Schwächen

2.1. GFS

2.1.1. Stärken

<u>Günstig</u>

Beim GFS kann durchschnittliche Hardware verwendet werden und es können trotzdem noch schnelle Datenraten garantiert werden.

<u>Fehlertoleranz und hohe Verfügbarkeit</u>

Dadurch das im GFS keine teure Hardware eingesetzt werden muss, kann es sehr leicht zu Ausfällen ganzer Nodes oder Storages kommen. Durch, wie bereits im Abschitt 1.2.1 erwähnt, ständige Überwachung, Fehlererkennung, Fehlertoleranz und automatische Wiederherstellung nach Fehlern wird jedoch eine Gegenmaßnahme getroffen.

<u>Trennung des Data Flow vom Control Flow</u>

Durch die Trennung des eigentlichen Datenstroms und dem Control Flow, kann die verwendete Netzwerktopologie optimal an den Datenstrom angepasst und somit effizient genutzt werden.

<u>Garbage Collection</u>

Chunks die keine dazugehörige Datei mehr haben, werden vom GFS automatisch erkannt und gelöscht.

[9]

2.1.2. Schwächen

Kleine Dateien

Werden im GFS kleine Dateien gespeichert, die im worst case in nur einen Chunk unterteilt werden und sich nur auf einem Server befinden, so kann es dazu kommen, dass ein Server zu einem Hotspot wird, wenn viele Clients gleichzeitig auf diese Datei zugreifen.

Spezielles Design

Da das GFS speziell an den Anforderungen der Google Search Engine angepasst wurde, ist es schwer dieses auf anderen Anforderungen anzupassen.

Master Memory Limitation

Der Master speichert Metadaten zu den Dateien wie Datei- und Chunk Namen, das Mapping von Dateien zu Chunks und den Standort der Chunk Replikationen, d.h. der Arbeitsspeicher des Masters könnte eine mögliche Limitierung darstellen.

[9]

2.2. Hadoop Distributed File System

Da das HDFS durch das GFS inspiriert wurde, teilen sie sich viele Eigenschaften, jedoch gibt es 3 Merkmale in denen sich die beiden signifikant voneinander unterscheiden.

- Indizierung der inodes
- Handling des Chunk Status und Standort
- Datenvollständigkeit

[10]

2.2.1. Indizierung der inodes

Beim HDFS wird der *file index state* und das Mapping von Dateien zu Chunks, auf dem NameNode im Arbeitsspeicher gehalten und regelmäßig auf die Hard Disk geschrieben. D.h. der Client selbst muss sich nicht darum kümmern den Chunk zu indizieren bzw. er muss relativ wenig über einen Chunk wissen.

Beim GFS, verwendet der Client die Größe eines Chunks und den Dateinamen und bildet daraus einen *chunk index*, welchen er dann zusammen mit dem Dateinamen an den Master schickt.

[10]

2.2.2. Chunk Status und Standort

Beim HDFS, wird der Standort und der Status eines Chunks, vom NameNode gemanagt.

Beim GFS, werden diese auf Anfrage eines Clients, vom Master an den Client gesendet, welcher diese Informationen dann zwischenspeichert.

[10]

2.2.3. Datenvollständigkeit

Beim HDFS, werden Daten die vom Client geschrieben werden, auf einer Pipeline von DataNode zu DataNode gesendet wobei der letzte DataNode in der Pipeline die Prüfsumme verifiziert.

Beim GFS, übernimmt jeder einzelne Chunkserver diese Aufgabe, wobei man alternativ auch die Replicas vergleichen könnte.

[10]

2.3. Sun Network File System

2.3.1. Stärken

Zugriffstransparenz

Da sich die API zum Zugriff auf Dateien auf einem NFS Server und im lokalen Dateisystem nicht unterscheiden, muss der Client auch nicht zwischen solchen Zugriffen unterscheiden. Es wird also für den Zugriff auf Dateien auf einem NFS Server die lokale API verwendet, d.h. Applikationen müssen nicht umgeschrieben werden, damit sie mit entfernten Dateien arbeiten können.

Ortstransparenz

Der Client bestimmt selbst, wo entfernte Dateisysteme auf dem lokalen Dateisystem gemountet werden. Der Knoten, auf dem sich das entfernte Dateisystem befindet, muss das Dateisystem exportieren und der Client muss es *entfernt mounten*. Der Client bestimmt dabei selbst, an welchem Punkt sich das entfernte Dateisystem, im lokalen Dateisystem befindet.

Heretogenität

NFS wird von sehr vielen, um nicht zu sagen fast allen, bekannten Betriebssystemen, Hardware-Plattformen und Dateisystemen unterstützt.

Sicherheit

Das NFS bietet eine Anbindung an den Authentifizierungsdienst Kerberos an, sodass die Sicherheit in einem NFS gewährleistet werden kann.

[2]

2.3.2. Schwächen

Mobilitätstransparenz

Wird ein Dateisystem, von einem Server auf einen anderen verschoben, so ist es notwendig die Mount-Tabellen des Clients, auf den neuen Standort des Dateisystems anzupassen.

[2]

3. Einsatzgebiete

3.1. GFS

Das GFS wurde speziell an die Anforderungen der *Google Search Engine* angepasst, d.h. große und schnell wachsende Datenmengen sind ein großes Berücksichtigungskriterium. Ein hoch Skalierbares und vor allem Effizientes Dateisystem ist gefragt.

3.2. NFS

Das NFS wurde entwickelt, um Dateien einfach und transparent zwischen einzelnen Clients zu teilen. Arbeitet man zum Beispiel gemeinsam an einem Projekt, das irgendwo auf einem entfernten Server liegt, so kann dieses beliebig auf dem lokalen Dateisystem gemountet werden. Die Anforderung an ein NFS sind also Clients, die sich untereinander Dateien teilen.

3.3. HDFS

Das HDFS ist eine Open-Source Java Implementierung, inspiriert vom GFS. Über das GFS, gibt es jedoch relativ wenig Informationen und Argumentierungen und wurde speziell an das von Google entwickelte MapReduce angepasst. Das HDFS ist also eine deutlich einfacher zu Implementierende Lösung.

Literaturverzeichnis

Primäre Quellen

[1] Adrew S. Tannenbaum, Maarten von Stehen. *Verteilte Systeme: Grundlage und Paradigmen*. 2003 Available at: http://catalogplus.tuwien.ac.at [zuletzt abgerufen am 06.11.2015]

[2] George Coulouris, Jean Dollimore, Tim Kindberg. *Verteilte Systeme: Konzepte und Design*. 2002 Available at: http://catalogplus.tuwien.ac.at [zuletzt abgerufen am 06.11.2015]

[3] Michael Weber. *Verteilte Systeme*. 1998.
Available at: http://catalogplus.tuwien.ac.at [zuletzt abgerufen am 06.11.2015]

[4] Hal Stern. NFS und NIS: Verwaltung von UNIX-Netzwerken. 1995
Available at: http://catalogplus.tuwien.ac.at [zuletzt abgerufen am 06.11.2015]

[5] Shvachko Konstantin, Kuang Hairong, Radia Sanjay, Chansler Robert. *The Hadoop Distributed File System*. Available at: http://catalogplus.tuwien.ac.at [zuletzt abgerufen am 16.03.2016]

[6] 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
[zuletzt abgerufen am 05.03.2016]

[7] Wikipedia.org. *Google File System Architecture*. Available at:
https://upload.wikimedia.org/wikipedia/commons/thumb/c/c3/GoogleFileSystemGFS.svg/2000px-GoogleFileSystemGFS.svg.png [zuletzt abgerufen am 05.03.2016]

[8] www.ostep.org. *Sun's Network File System*. Available at:
http://pages.cs.wisc.edu/~remzi/OSTEP/dist-nfs.pdf [zuletzt abgerufen am 16.03.2016]

[9] Kaushiki Nag, 25. Oktober 2012. *Google File System*. Available at: http://kaushiki-gfs.blogspot.co.at/ [zuletzt abgerufen am 17.03.2016]

[10] Ameya Daphalapurkar, Manali Shimpi, Priyal Newalkar 9 September, 2014. *MapReduce & Comparison of HDFS And GFS.* Available at: www.ijecs.in [zuletzt abgerufen am 17.03.2016]