Ausarbeitung zur Vorlesung “Dependable Systems” im Sommersemester 2010
Hasso-Plattner-Institut, Universität Potsdam

Distributed Fault-Tolerant File Systems

Marko Röder Martin Schütte

31. Juli 2010

Inhaltsverzeichnis

1 Begriffsbestimmung

In dieser Arbeit betrachten wir den Aufbau verteilter fehlertoleranter Dateisysteme. Eine Begriffsbestimmung beginnt mit der klassischen Definition eines verteilten Systems als Zusammenschluss mehrerer unabhängiger Rechner, die dem Benutzer aber als einheitliches System präsentiert bzw. zugänglich gemacht werden [TvS07]. Analog dazu ist ein verteiltes Dateisystem ein System, das Dateien auf verschiedenen Rechnern speichert, diese dem Benutzer aber einheitlich präsentiert, das heißt in einer Verzeichnishierarchie organisiert und über eine Schnittstelle (zum Beispiel unter Unix die VFS-Abstraktion) erreichbar macht.

Das zusätzliche Attribut „fehlertolerant“ bezeichnet die Möglichkeit den Dienst auch beim Eintreten von Fehlern zu erbringen. Ein fehlertolerantes verteiltes Dateisystem erlaubt also das Arbeiten mit den Dateien auch wenn ein beteiligter Rechner ausfällt.

2 Anforderungen

2.1 Fehlermodell

Während der Betrachtung der unterschiedlichen Ansätze, wie Fehlertoleranz von verschiedenen, verteilten Systemen umgesetzt wird, haben wir das im Folgenden genauer erläuterte Fehlermodell ermittelt.

Zu den behandelten Fehlern gehören:

Fail-Stop von Komponenten, was unterschiedliche Ursachen und Auswirkungen haben kann. Zu den Ursachen hierfür zählt z. B. der Stromausfall in Teilen des verteilten Systems. Mögliche Auswirkungen hingegen sind z. B. dass plötzlich ein Rechner ausfällt, aber es kann sich auch nur um den Ausfall einer Festplatten, eines Festplatten-Controller, o. ä. handeln.

Netzwerkpartition, die durch den Ausfall der Kommunikation zwischen den Rechnern entstehen. Diese sind ähnlich einem Ausfall mehrer Rechner zu sehen, da für die unterschiedlichen Teile der Kontakt zu der anderen Seite abbricht und nicht erkannt werden kann, dass es sich nur um eine Partition handelt.

Weil kein Gerät und keine Systemkomponente perfekt ist, muss in großen verteilten Systemen immer mit Komponentenausfällen gerechnet werden. Die Herausforderung besteht nun darin diese Komponentenausfälle so zu behandeln, dass sie die Funktionsfähigkeit des Gesamtsystems sowenig wie möglich beeinträchtigen. Aus diesem Grund ist es notwendig Komponentenfehler möglichst frühzeitig zu erkennen, elegant und unbemerkt zu überbrücken sowie schnell zu beheben. Dabei ist das oberste Ziel, dass mit dem Dateisystem während der ganzen Zeit weitergearbeitet werden kann.

Zu den Fehlerzuständen die in dieser Betrachtung nicht berücksichtig werden, zählen Softwarefehler in der Implementierung des verteilten Dateisystems selbst, Fehler im Betriebssystem und den Gerätetreibern (z. B. den Festplatten-Treibern) und Berechnungsfehler, die z. B. bei der Berechnung von Checksummen auftreten können. Ebenso deckt nicht jeder Ansatz den inkrementellen Datenverlust einer Festplatte durch Veraltung und „Abnutzung“ ab.

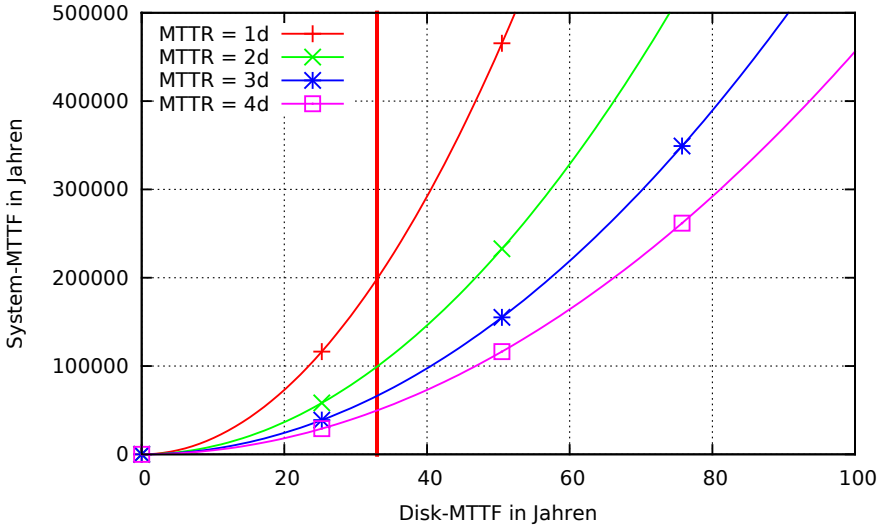

Abbildung 1: Einfache Replikation zweier Komponenten: Abhängigkeit der System-MTTF von den Komponenten-MTTF und -MTTR (markiert ist eine für Festplatten anzunehmende MTTF von 33 Jahren).

2.2 Hardware-Charakteristiken

Für den Ausfall von Komponenten, in diesem Fall besonders von Festplatten, gibt es vereinzelte empirische Studien, so dass sich Zuverlässigkeitswerte angeben lassen. Grundlage der Betrachtung ist die *Mean Time To Failure* (MTTF) als mittlere Zeit bis zum Ausfall einer Komponente, die *Mean Time To Repair* (MTTR) als mittlere Zeitspanne zwischen Ausfall und Reparatur, sowie die *Annualized Failure Rate* (AFR) als Umrechnung der MTTF auf ein Jahr[1].

Für Festplatten wurde eine mittlere AFR von etwa 3 % ermittelt [SG07], was einer MTTF von 33 Jahren entspricht (damit ist dieser Wert deutlich geringer als die von den Herstellern üblicherweise angenommene AFR von 0.88 % bzw. MTTF von 114 Jahren). Für die Fehlfunktion eines Rechners bzw. Clusterknotens sind Festplattenfehler nur eine von mehreren möglichen Ursachen; weitere häufige Fehlerquellen sind CPU, Arbeitsspeicher, Stromversorgung und auch die Software – die Zuverlässigkeit bzw. die Ausfallraten sind je nach System allerdings sehr unterschiedlich und lassen sich nicht sinnvoll allgemeingültig beziffern [SG06].

Die zweite wichtige Größe ist die MTTR nach einem Komponentenausfall. Je nach Art des Ausfalls, der Organisation und vorhandener Automatisierung kann diese von wenigen Stunden bis zu mehreren Tagen variieren. Schon die beispielhafte Betrachtung einer einfachen Replikation[2] (in diesem Kontext seien dies zwei Festplatten im RAID 1) zeigt, dass das Hauptziel im Vermeiden von Doppelfehlern (dem gleichzeitigen Ausfall beider Plat-

[1] $\mathrm{AFR} = 1/(\mathrm{MTTF_y} + \mathrm{MTTR_y}) \approx 1/\mathrm{MTTF_y}$
In der Praxis werden Platten auch schon vor einem erwarteten Ausfall gewechselt, so dass beim Erfassen des Datenmaterials von der *Annualized Replacement Rate* (ARR) gesprochen wird. Wir nehmen an dass AFR = ARR

[2] Die System-MTTF ergibt sich dabei aus MTTF und MTTR der zwei Komponenten mit $\mathrm{MTTF_{system}} = \frac{\mathrm{MTTF}}{2} \times \frac{\mathrm{MTTF}}{\mathrm{MTTR}}$

ten) liegt. Während die MTTF sich nur wenig beeinflussen lässt, kann eine Verbesserung der MTTR (durch organisatorische Maßnahmen und Optimierung der Systemarchitektur) leichter zu erreichen sein und führt ebenfalls zu einer größeren Zuverlässigkeit des Gesamtsystems (vgl. Abbildung 2.2).

An dieser Stelle zeigt sich außerdem dass Fehlertoleranz auf verschiedenen Ebenen eingefügt werden kann: einerseits müssen im verteilten System immer Knotenausfälle behandelt werden – andererseits können aber auch die Knoten selbst so fehlertolerant ausgelegt werden dass der Ausfall einer Komponente (besonders einer Festplatte) nicht sofort zum Ausfall des Knotens führt. Bei der Konzeption eines verteilten fehlertoleranten Dateisystems ist es eine Designentscheidung und Kosten/Nutzen-Abwägung ob die Zuverlässigkeit der Knoten erhöht wird (zum Beispiel falls eine Knotenausfall eine aufwendige Resynchronisation erfordert), oder ob stattdessen günstigere und weniger zuverlässige Knoten benutzt werden weil die Fehlerbehandlung eines Knotenausfalls optimiert wurde.

2.3 Dateisystem-Charakteristiken

Viele Designentscheidungen eines Dateisystems beruhen auf Annahmen über die zu speichernden Daten und erwarteten Zugriffsmuster der Nutzer; daher haben empirische Untersuchungen von Dateisystemen und Datenbeständen lange Tradition. Um einen Überblick über die Anforderungen zu bekommen betrachten wir zwei aktuelle Studien, einerseits für Workstations und andererseits für High Performance Computing (HPC) Systeme.

Zur Datenerfassung für Workstations wurde 2007 über drei Monate hinweg der CIFS-Netzwerkverkehr zweier großer Firmen-Dateiserver mit vorwiegend Windows-Clients erfasst [LPGM08]. Dabei wurde beobachtet dass auf weniger als 10 % der vorhandenen Daten auch zugegriffen wurde, wobei das Verhältnis von Lese- zu Schreiboperationen bei 2:1 lag und etwa die Hälfte der Operationen auf Metadaten entfällt. Die aktiv benutzten Dateien sind meist klein (bis 100 KB), allerdings wird die meiste Netzwerk-Bandbreite von Zugriffen auf größere Dateien beansprucht (ca. 75 % der Dateien sind kleiner als 20 KB, diese verursachen aber nur ca. 30 % des Netzwerkverkehrs). Der gleichzeitige Zugriff zweier Clients auf dieselbe Datei ist sehr selten (nur ca. 10 % der Dateien werden überhaupt von mehr als einem Client geöffnet) und dabei überwiegen die lesenden Zugriffe (ca. 90 %), so dass Konflikte beim Schreibzugriff noch seltener sind.

Im HPC-Bereich werden Dateien etwa gleichoft gelesen und geschrieben, allerdings dominiert das sequentielle Lesen/Schreiben einer ganzen Datei und der direkte Zugriff (*random access*) innerhalb von Dateien ist selten [SS07]. Die ungleiche Verteilung der Dateigrößen finden sich hier auch, allerdings sind die Dateien generell größer (ca. 50 % der Dateien sind kleiner als 20 KB, ca. 90 % kleiner als 2 MB) [Day08].

3 Techniken

3.1 Dateizugriffs-Semantiken

Das Verhalten verschiedener Dateisystem-Operationen muss für die Anwendungen dokumentiert sein.

Für UNIX-artige Betriebssysteme ist hierbei die POSIX-Spezifikation maßgeblich, die das Verhalten von `read`- und `write`-Systemaufrufen festlegt (daher bezeichnet als POSIX- oder UNIX-Semantik). Diese erfordert eine sofortige Sichtbarkeit (bzw. Auswirkung auf

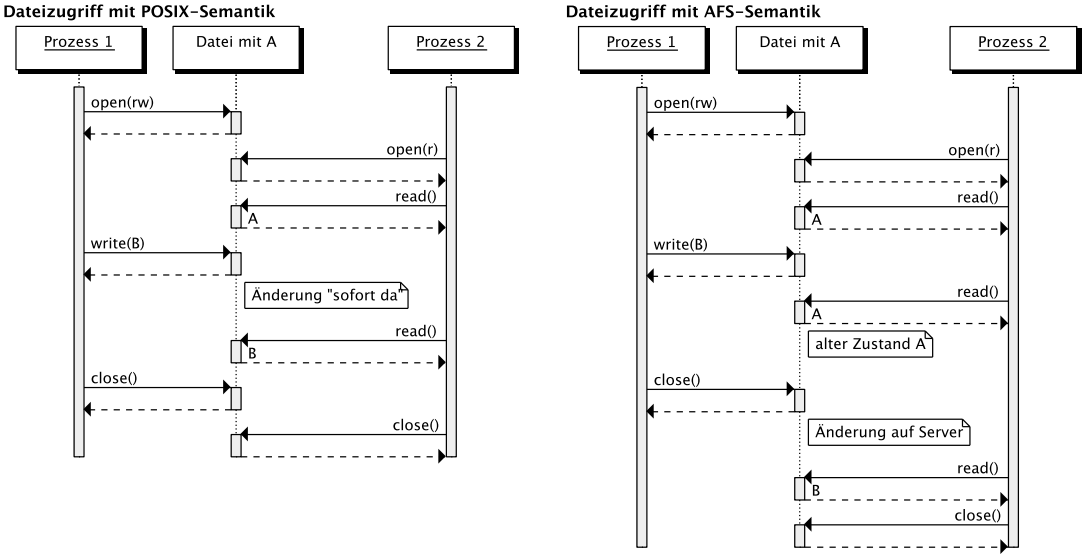

Abbildung 2: Gemeinsamer Dateizugriff mit POSIX- und AFS-Semantik

den Systemzustand) einer Änderung mittels `write`, so dass jedes folgende `read` schon die neuen Daten zurückgibt[3]. Selbst eine Optimierung durch Umsortieren von Zugriffen ist nur erlaubt wenn `write-read`-Folgen auf gleichen Positionen korrekt beibehalten werden[4].

Im verteilten System ist diese Spezifikation problematisch, denn sie beschränkt die Möglichkeit den Datenzugriff mit Caches zu beschleunigen. Sobald eine Datei von mehreren Clients benutzt wird und ein Client diese Datei auch zum Schreiben geöffnet hat, muss auf alle Caches verzichtet werden um alle Zugriffe auf dem Server in korrekter Reihenfolge auszuführen. Dies führt natürlich zu Performance-Einbußen und belastet das Netzwerk. Seit einiger Zeit wird daher über eine POSIX-Erweiterung für HPC-I/O diskutiert mit der Anwendungen auf bestimmte Garantien verzichten können [VLR+08]; eine Aufnahme dieser Erweiterungen in verbreitete Betriebssysteme ist allerdings noch nicht abzusehen.

Um dieses Problem zu umgehen ist es hilfreich verschiedene Clients und Prozesse stärker voneinander zu isolieren, so dass mehr Zugriffe unabhängig voneinander erfolgen können. Wenn die Auswirkung eines `write`s erst dann auf den Dateiserver übertragen und sichtbar wird, wenn der schreibende Prozess die Datei schließt, dann können alle Lese- und Schreibzugriffe der Clients auf lokalen Arbeitskopien erfolgen und belasten nicht das Netzwerk. Erst beim `close` (oder `fsync`) muss kommuniziert werden, um zum Beispiel alle aktiven Arbeitskopien anderer Clients zu invalidieren oder aktualisieren. Dieses Zugriffsverhalten wird als AFS- oder Session-Semantik bezeichnet. Da sie für die meisten Anwendungen

[3] "After a `write()` to a regular file has successfully returned: any successful `read()` from each byte position in the file that was modified by that write shall return the data specified by the `write()` for that position until such byte positions are again modified." [IG08, `write` description]

[4] "Writes can be serialized with respect to other reads and writes. If a `read()` of file data can be proven (by any means) to occur after a `write()` of the data, it must reflect that `write()`, even if the calls are made by different processes. A similar requirement applies to multiple write operations to the same file position. This is needed to guarantee the propagation of data from `write()` calls to subsequent `read()` calls. This requirement is particularly significant for networked file systems, where some caching schemes violate these semantics." [IG08, `write` rationale]

und Dateizugriffsmuster gut funktioniert[5] wird sie von fast allen verteilten Dateisystemen benutzt.

Darüberhinaus besteht auch immer die Möglichkeit ein Dateisystem noch stärker an eine bestimmte Anwendung anzupassen. Beispiele dafür sind das Google File System, das ohne `write`-Operation auskommt (vgl. Abschnitt 4.3), sowie parallele Dateisysteme für High Performance Computing, in denen die Koordination paralleler Zugriffe auf Anwendungsebene erfolgt.

3.2 Lock-Management

In Verbindung mit der Zugriffssemantik steht auch die Koordination konkurrierender Dateizugriffe von verschiedenen Clients. Um die Konsistenz zu gewährleisten sollen üblicherweise beliebig viele Clients eine Datei lesen, aber immer nur ein Client eine Datei verändern dürfen (die Anzahl und Art der Zugriffsmodi ist je nach System unterschiedlich).

Eine einfache Lösung ist die Benutzung eines Knotens als Locking-Dienst für das ganze Dateisystem. – Unter dem Aspekt der Fehlertoleranz ist dies jedoch keine ausreichende Lösung, denn dieser Knoten wird zum *Single Point of Failure* und sein Ausfall führt zum Ausfall des ganzen Dateisystems.

Daher ist ein verteiltes Lock-Management notwendig, das den Ausfall eines beteiligten Knotens überlebt. Die klassische Implementierung dieses Dienstes ist der *Distributed Lock Manager* eines VAXClusters [ST87]. Dabei wird vorausgesetzt dass alle beteiligten Knoten über eine konsistente Netzsicht und zuverlässige Gruppenkommunikation verfügen. Die Knoten verwalten nun eine gemeinsame Liste in der jede Datei einem Masterknoten zugeordnet wird; dieser Masterknoten verwaltet alle Zugriffsberechtigungen für „seine“ Dateien. Wenn sich die beteiligten Knoten ändern (also insbesondere wenn ein Knoten ausfällt) so starten die übrigen Knoten einen Wiederherstellungs-Prozess in dem sie die Zuordnungsliste neu erstellen und alle Zugriffsberechtigungen neu vergeben.

3.3 Striping

Die zu speichernden Dateien werden von vielen verteilten Dateisystemen (wie auch in lokalen Datei- und RAID-Systemen) in Blöcke fester Größe zerlegt und auf mehreren Speicherknoten verteilt.

Der wichtigste Vorteil (zum Beispiel gegenüber einer Verwaltung und Verteilung auf Dateiebene) ist dabei die Lastverteilung unter den Speicherknoten.

Besonders in Verbindung mit Replikation (vgl. Abschnitt 3.5) und bei Berücksichtigung der Netzwerk-Topologie erlaubt Striping es einerseits die verschiedene Blöcke großer Dateien so zu verteilen, dass Clients sie parallel lesen und schreiben können, und andererseits dass in großen Netzen verschiedene Kopien der Blöcke in verschiedenen Teilnetzen liegen und so auch nach Ausfällen noch erreichbar sind.

[5] Klassische Anwendungen, die nicht mit AFS-Semantik kompatibel sind, sind Mailserver und Datenbanken. In diesen Fällen werden sehr häufig Dateien geschrieben und von anderen Prozessen gelesen – dadurch entfallen die Vorteile der lokalen Arbeitskopien und ständige Cache-Aktualisierungen belasten das Netzwerk noch zusätzlich.

3.4 Separate Metadaten

Viele der betrachteten verteilten Dateisysteme trennen die Metadaten, die z. B. den Dateinamen, Zugriffszeiten und Rechte enthalten, von den Dateiinhalten. Dazu kommt in verteilten Dateisystemen noch, dass diese Metadaten auch enthalten wo eine Datei oder deren einzelne Dateifragmente gespeichert sind (vgl. Abschnitt 3.3) und wie oft sie ggf. repliziert wurden (vgl. Abschnitt 3.5). All diese Metadaten werden daher in den meisten Fällen zentral auf einem eigenen Master- oder Metadaten-Knoten gespeichert, der nur zur Metadatenverwaltung da ist.

Vorteil einer solchen Separierung ist ganz klar, dass die sicherlich häufiger angefragten Dateiinhalte von den eher weniger gebrauchten Daten zur Verwaltung der Dateien getrennt sind. Damit lassen sich die Zugriffe auf die Inhalte besser auf mehrere Speicherknoten verteilen, während ein zentraler Knoten für die Verwaltung zuständig ist.

Als Nachteil ergibt sich daraus, dass es einen zentralen Knoten, den Knoten mit den Metadaten gibt, der im Zweifelsfall sehr häufig für unterschiedliche Metadaten angefragt wird und diese auch allein nur besitzt. Somit wird er zum *Single Point of Failure* und was noch viel schlimmer ist, bei einem kompletten Verlust der Metadaten sind auch die verteilten Dateiinhalte nutzlos (da sie nicht mehr den Dateien zugeordnet werden können). Aus diesem Grund setzen fast alle Dateisysteme, die separate Metadaten besitzen, darauf diese zu replizieren und mindestens einen weiteren Knoten zu haben, der die Metadaten zusätzlich speichert. Eine der Ausnahmen hier ist das Hadoop File System (HDFS) [SKRC10], das die Metadaten nicht repliziert.

3.5 Replikation

Eine der wohl bekanntesten Arten von Replikation ist RAID (Redundant Array of Independent Disks). Hierunter werden verschiedene Kombinationen zusammengefasst wie Daten über mehrer Festplatte verteilt werden können, sodass eine gewisse Ausfallsicherheit bezüglich einer einzelnen Festplatte entsteht. So kann z. B. Striping (vgl. Abschnitt 3.3) zusammen mit Paritätsbits (einer Art Korrekturbits) eingesetzt werden, damit eine Festplatte problemlos ausfallen kann und es dabei trotzdem zu keinem Datenverlust kommt.

Auch verteilte Dateisysteme setzen RAID zum Absichern des Ausfalls einer Festplatte ein, nur reicht dies oft nicht aus. Neben dem Ausfall einer Festplatte kann nämlich auch ein kompletter Speicherknoten (mit mehreren Festplatten) ausfallen, was zumindest einen temporären Verlust der Daten zur Folge hat (je nachdem ob, sich der Schaden reparieren lässt). Aus diesem Grund gibt es einige verteilte und fehlertolerante Dateisysteme, die ein ähnliches Prinzip wie das des RAID auf die Ebene der Knoten überträgt. So werden einzelne Dateien und Dateifragmente nicht nur über mehrere Festplatten, sondern auch über mehrere Knoten verteilt (vgl. Abschnitt 3.3).

Besonders wichtig ist die Replikation von Daten bei den zuvor erwähnten separaten Metadaten, da sonst ein *Single Point of Failure* entsteht. Doch auch bei Speicherknoten, die „nur“ die Inhalte von Dateien enthalten ist dies sinnvoll und hat zusätzlich den Vorteil einer besseren Lastverteilung, da die selben Daten von unterschiedlichen Knoten gelesen werden können.

Einer der Nachteile von Replikation ist, dass sowohl das Original als auch die Replikate bei einem Schreibvorgang immer synchron gehalten werden müssen und so ein gewisser Aufwand zum Abgleich der Daten entsteht. Eine Möglichkeit diesen Aufwand zu

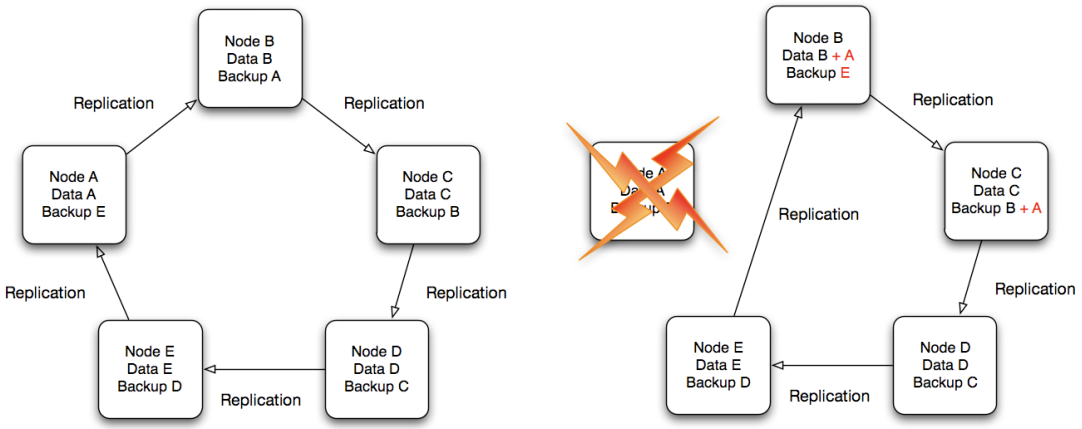

Abbildung 3: Ring-/Buddy-Replikation [Bon10]

verringern, ist die Verwendung von Schattenreplikaten, die dann zwar nicht mehr zum Lastausgleich verwendet werden können (da sie nur für den Notfall bereitstehen), aber eine Synchronisation so nur vom Original zur Kopie notwendig ist und im Hintergrund stattfinden kann.

Ein in verteilten Dateisystemen häufig anzutreffendes Replikationsmuster ist die sogenannte Ring-Replikation, auch *Buddy Replication* genannt. Hier werden alle Knoten gedanklich in einem Kreis angeordnet und jeder Knoten repliziert seine eigenen Daten zusätzlich auf die nächsten `n` Knoten (siehe Abbildung 3). Bei einem Ausfall wird dann der ausgefallene Knoten aus dem Ring entfernt und es werden neue Replikate auf den noch verbleibenden Knoten angelegt.

3.6 Journaling

Eine weitere Technik, die in großen, verteilten Dateisystemen hilfreich ist und daher eingesetzt wird, ist die des Journaling. Ausgangspunkt hierfür ist, dass es bei so großen Datenmengen, die in einem verteilten Dateisystem gespeichert werden, nicht möglich ist ein komplette Überprüfung der Daten durchzuführen (wie dies z. B. beim Starten eines Linux-Systems durch `fsck` passiert). Dennoch muss die Konsistenz des Dateisystems sichergestellt werden können. Dies geschieht normalerweise aber durch eine komplette Überprüfung der Daten und ggf. das Wiederherstellen bei der Abweichung von Soll- und Ist-Zustand. Die Lösung des Problems – Journaling – ist eine Lösung, die bereits bei vielen Unix-Dateisystemen zum Einsatz kommt.

Journaling funktioniert so, dass in einem Journal alle Änderungen an Dateien zuerst erfasst werden, bevor diese zur Ausführung kommen. Das Journal wird, nachdem die Änderungen eingetragen wurden, als erstes auf die Festplatte gespeichert, um dem Verlust von Einträgen vorzubeugen. So kann ein Ausfall einer Komponente, sei es ein Knoten oder nur eine Festplatte, durch vor- oder zurückspielen des Journals behoben werden.

Ist das Journal geschrieben, so kann die eigentliche Abarbeitung der Änderung durchgeführt werden, doch im Zweifelsfall kann schon nach dem Schreiben des Journals eine Rückmeldung an den Client gegeben werden, dass die Änderung vollzogen wurde (da sie

nachvollziehbar gespeichert ist).

Wenn ein Journaleintrag abgearbeitet ist und die Änderungen durchgeführt wurden, so kann dieser Teil des Journals problemlos vernichtet werden, was in den meisten Fällen durch ein Überschreiben der Daten bei zukünftigen Änderungen passiert (das Journal hat meist eine festgelegt Größe, damit es bei vielen Aktionen nicht unendlich groß wird).

3.7 Quorum & Fencing

Die Begriffe *Quorum* und *Fencing* sind in den meisten Fällen als zusammenhängend anzusehen. Beide Begriffe kommen zum Einsatz, wenn sich die Gruppe von Clients durch einen Ausfall des dazwischen liegenden Netzwerks in zwei Teile zerteilt (ein sogenanntes *Split-Brain*) und eine Kommunikation und Abstimmung beider Teile somit nicht mehr möglich ist.

Das Quorum wird genutzt, um zu bestimmen welche Hälfte weiterhin mit den mit den Speicherknoten arbeiten darf. Hierfür können verschiedene Algorithmen eingesetzt werden, wodurch nur durch Kommunikation beider Hälften mit dem Daten festgestellt werden kann, wer die Mehrheit besitzt und somit das Recht erhält weiterzuarbeiten.

Da nach der Ermittlung der Mehrheit durch das Quorum eine Hälfte die Minderheit ist, muss im zweiten Schritt dafür gesorgt werden, dass diese Minderheit keine Zugriffe mehr auf die Speicherknoten durchführt. Auch hierfür stehen verschiedene Techniken zur Verfügung und unterschiedliche verteilte Dateisysteme setzen hier verschiedene Techniken ein. Eine Möglichkeit ist z. B. dass den Speicherknoten mitgeteilt wird, wer das Quorum besitzt und diese danach die Kommunikation mit der anderen Hälfte verweigern.

Da Quorum und Fencing voraussetzt, dass erkannt wird, dass es eine Partitionierung der Clients gab, wird in den meisten Fällen *Monitoring*[6] eingesetzt, um diesen Fall zu erkennen.

4 Beispiele

Bevor wir drei aktuelle, verteilte, fehlertolerante Dateisysteme näher betrachten, sollen einige klassische und im Einsatz verbreitete Systeme kurz vorgestellt werden.

NFS Das *Network File System* von Sun Microsystems dürfte (in der Version 3 von 1995 [CPS95]) das unter Unix am weitesten verbreitete verteilte Dateisystem sein. Es erlaubt einem Server Teile seines lokalen Dateisystems für Clients bereitzustellen und zeichnet sich durch sein zustandsloses Design aus.

AFS Das *Andrew File System* [HKM+88] der Carnegie Mellon University (CMU) benutzt die Session-Semantik und ist die erste Implementierung eines verteilten Dateisystems mit persistentem Client-Cache für Daten und Metadaten. Dadurch kann auch ein einzelner Dateiserver vergleichsweise viele Clients bedienen.

[6] Unter Monitoring versteht man die Überwachung der Clients und Knoten z. B. durch Heartbeat-Nachrichten und die damit verbundene Prüfung, ob dieser noch aktiv ist.

Coda Coda [Sat90] ist eine Weiterentwicklung von AFS und fügt diesem die Möglichkeit hinzu, Dateiserver zu replizieren und Clients offline arbeiten zu lassen. Dazu werden alle Dateien mit Versionsnummern versehen und jegliche offline durchgeführten Dateiänderungen mitgeloggt. Bei einer Reintegration können so die Versionen schnell verglichen und die Änderungen in Reihenfolge eingespielt werden[7].

VAXCluster Das VAXCluster (später: VMSCluster [KLS86]) der Digital Equipment Corporation beinhaltete ein eigenes Dateisystem das in diesem Kontext von Bedeutung ist. Es unterscheidet sich von den anderen hier betrachteten Dateisystemen, da es auf *shared storage*, also von allen Knoten gemeinsam benutzten Festplatten, aufbaut. Damit entfällt der Dateizugriff über das Netzwerk ebenso wie die Unterscheidung der Knoten in Clients und Server. Die fehlertolerante Koordination der Clusterknoten mittels Quorum und Distributed Lock Manager waren zudem wegweisend und finden sich auch in aktuellen, fehlertoleranten, verteilten Dateisystemen wieder.

Lustre Lustre (Wortbildung aus "Linux" und "Cluster" [Bra02]) ist ein verteiltes Dateisystem das bereits Metadaten-Server und Speicherknoten trennt. Viele Konzepte, insbesondere das Locking, wurden dafür vom VAXCluster übernommen. Ursprünglich an der CMU entstanden, wurde die Entwicklung lange Zeit von Sun Microsystems vorangetrieben wodurch das System im Bereich des Supercomputing recht verbreitet ist.

4.1 Ceph

4.1.1 Allgemeines

Das Ceph Dateisystem wurde an der University of California Santa Cruz konzipiert [WBM+06, Wei07] und befindet sich als Open Source-Projekt in aktiver Entwicklung. Grundlegendes Designziel ist die Skalierbarkeit auf große Datenmengen und viele Knoten. Damit einhergehend ist die Fehlertoleranz ein weiteres Ziel, da ab einer gewissen Knotenmenge die Knotenausfälle vom Fehler- zum Normalzustand werden.

Die Implementierung von Ceph wird zur Zeit als Open Source Projekt entwickelt; das System ist noch nicht für produktiven Einsatz geeignet.

4.1.2 Architektur

Ceph besteht aus drei Gruppen von Servern: den Speicher-, den Metadaten- und den Monitorknoten. Die Monitorknoten müssen dabei keine separaten Maschinen sein, es reicht den entsprechenden Dienst auf einigen Speicher- und Metadatenservern mitlaufen zu lassen.

Aufgabe der Monitore ist das Erkennen von Knotenausfällen und die Verwaltung einer konsistenten Netzsicht (der *cluster map*). Diese Netzsicht wird an alle anderen Server und Clients verbreitet.

Die Speicherknoten (die *Object Storage Devices*, OSD) bilden als eigenes Teilsystem das *Reliable Autonomic Distributed Object Store* (RADOS), einen verteilten Objektspeicher,

[7] Es kann natürlich vorkommen dass so mehrere Knoten unabhängig voneinander dieselben Daten verändern – solche Konflikte lassen sich dann nicht immer automatisch auflösen.

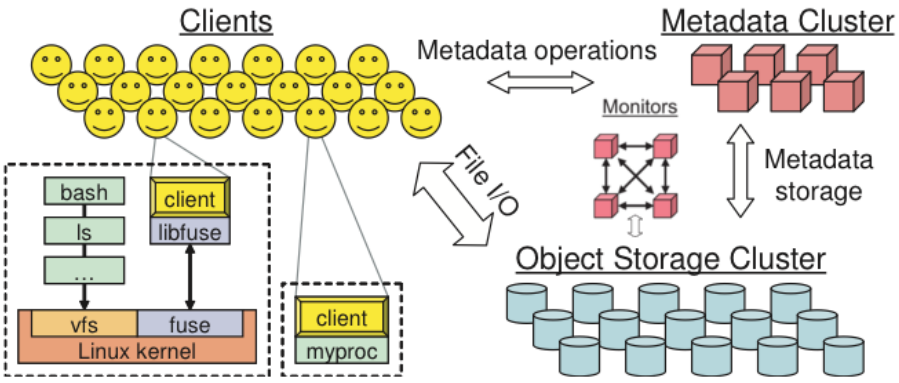

Abbildung 4: Architektur von Ceph

der möglichst autonom die verteilte und zuverlässige Speicherung von Datenobjekten übernimmt. Replikation, Zugriffskontrolle und die Fehlerbehandlung bei Ausfällen werden (auf Grundlage der *cluster map*) komplett von RADOS verwaltet.

Die Metadatenserver (MDS) verwalten alle Dateiattribute (Inodes, Namen, Verzeichnisse) und Locks. Dabei fungieren sie praktisch als Cache zum schnellen Zugriff auf die Metadaten, die in persistenter Form ebenfalls als Objekte im Dateisystem abgelegt, aber zur Laufzeit von den MDS im Speicher gehalten werden. Zur Lastverteilung wird die Zuständigkeit der MDS für Teilbäume der Verzeichnishierarchie dynamisch angepasst. Falls die Aktivität auf einem MDS unverhältnismäßig groß wird, so werden Verzeichnis-Teilbäume auf einen anderen MDS migriert um die Last wieder gleich zu verteilen.

4.1.3 Striping und Replikation

Die Organisation der Daten erfolgt über mehrere Ebenen: Dateien werden zunächst in (normalerweise 8 MB große) Objekte aufgeteilt. Diese Objekte können verschiedenen *placement groups* (PG) angehören (normalerweise gibt es nur zwei PGs für Daten und Metadaten). Über die PGs wird auch die Zahl der Replikate konfiguriert.

Eine Hashfunktion berechnet dann für jedes Objekt aus seiner ID, seiner PG sowie der aktuellen *cluster map* eine Liste von OSDs auf denen das Objekt repliziert gespeichert wird. Der Client muss dabei nur mit dem ersten OSD (dem *primary OSD*) kommunizieren, im Falle eines Schreibvorgangs initiiert dieser dann die Replikation auf die weiteren OSDs.

Falls OSDs ausfallen oder neu hinzukommen, so wird durch eine Aktualisierung der *cluster map* eine Überprüfung aller Objekte und ihrer jeweiligen OSD-Listen angestoßen. Falls ein OSD aus der Liste herausfällt (also in der alten, aber nicht mehr in der neuen Liste enthalten ist), so kontaktiert er den (möglicherweise neuen) *primary OSD* um ihm gegebenenfalls die aktuelle Version des Objekts zu senden. Falls ein OSD in eine Liste hinzukommt, so ist der *primary OSD* dafür zuständig ihm ein Replikat zu senden. Durch die Verteilung der Datenobjekte erfolgt der ganze Vorgang stark parallel mit Beteiligung aller Speicherknoten.

4.1.4 Metadaten und Journaling

Die Metadatenserver benutzen große Journaling-Dateien um alle Metadaten-Operationen schnell ins RADOS zu speichern. Dies erlaubt nach Ausfall eines MDS einen schnellen *failover* innerhalb einiger Sekunden: ein anderer Knoten wird das Journal einlesen, nicht abgeschlossene Transaktionen wiederholen, Client-Sessions wiederherstellen und dann als zuständiger MDS die Arbeit aufnehmen.

Die Verarbeitung der Journals, also das Anwenden der Operationen auf die persistenten Metadaten-Objekte im RADOS, erfolgt aus Performanzgründen so spät wie möglich. Einerseits können dann viele Änderungen ignoriert werden weil sie später wieder überschrieben werden oder sich auf später gelöschte Dateien beziehen; und andererseits können so viele einzelne Änderungen innerhalb eines Verzeichnisses zu einem einzigen Metadaten-Objektzugriff zusammengefasst werden.

4.1.5 Zugriffssemantik

Im Normalfall erfolgt der Dateizugriff (bzw. der Zugriff auf Datenobjekte im RADOS) mit POSIX-Semantik; es werden zusätzlich aber auch Teile der geplanten POSIX-Erweiterung für HPC-I/O unterstützt. So können Clients im Falle gemeinsamen Dateizugriffs auf die korrekte Schreib-Lese-Ordnung verzichten und dem Dateisystem erlauben bis zu einer expliziten Synchronisierung alle Operationen im Cache auszuführen.

4.1.6 Locking

Eine Designentscheidung von Ceph ist kein Locking für Metadaten zu implementieren, weil konkurrierende Änderungen hier nicht als hinreichend relevantes Problem gesehen werden.

Für den Datenzugriff verwalten die *primary OSDs* die Locks für ihre Objekte. Locks (üblicherweise mit einem Timeout versehen, also genaugenommen *leases*) sind Objekt-Attribute und werden als solche erst repliziert und dann erst dem Client bestätigt.

4.1.7 Quorum

Die Monitore benutzen den Paxos-Algorithmus, um ihren gemeinsamen Zustand (die *cluster map*) zu aktualisieren. Dieser erfordert ein einfaches Quorum, also die Teilnahme einer Mehrheit der Monitore, um Änderungen vorzunehmen. Falls, zum Beispiel während einer Netzpartition, das Quorum nicht erreicht wird, so können die verbleibenden Monitore ihre *cluster map* nicht aktualisieren und werden sie auch nicht länger an Clients oder MDS verbreiten; dadurch wird (nach Timeout der Caches etc.) ein implizites Fencing erreicht, weil ohne *cluster map* keine Dateisystem-Zugriffe mehr möglich sind.

4.2 General Parallel File System

4.2.1 Allgemeines

Das von IBM entwickelte, verteilte General Parallel File System (**GPFS**) [SH02] ist ein weiteres Beispiel für ein fehlertolerantes, verteiltes Dateisystem. Es ist vor allem für Cluster-Computer gedacht und wird dort als übergreifendes Dateisystem zur gemeinsamen Nutzung der Festplatten verwendet. Neben professionellen Linux-Distributionen

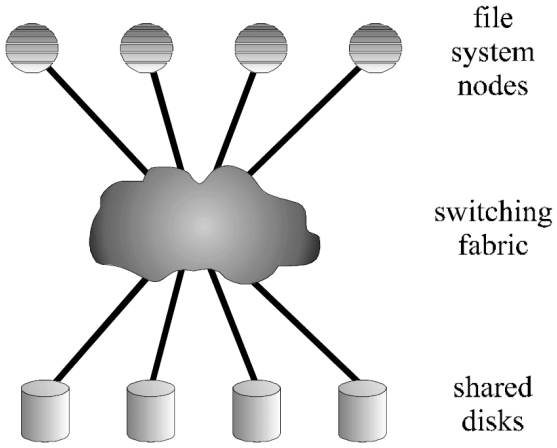

Abbildung 5: Abstrakte Architektur des General Parallel File System [SH02]

unterstützt es das ebenfalls von IBM entwickelte Betriebssystem AIX und die Server-Betriebssystem-Familie des Microsoft Windows Server 2003 und 2008. Hinzu kommt, dass es durch den Einsatz im Bereich der (IBM-)Supercomputer auch IBM Blue Gene als Plattform unterstützt. So setzt auch die heutige Generation von IBMs Supercomputern auf das GPFS als verteiltes, fehlertolerantes Dateisystem.

4.2.2 Anforderungen

Das GPFS entwickelte sich aus dem Tiger Shark Multimedia Dateisystem, dass ebenso zur Familie der verteilten Dateisysteme gehört und 1993 zum Zeitpunkt der Entwicklung vor allem im Zusammenhang mit großen Multimedia-Anwendungen eingesetzt werden sollte. Die Anforderungen für beide Dateisysteme waren eine große Datendurchsatzrate, hohe Speicherkapazitäten und natürlich eine hohe Zuverlässigkeit der gespeicherten Daten. Zusätzlich soll auch eine hohe Zahl an Zugriffen ermöglicht werden und dies ganz besonders schon in einem einzelnen Verzeichnis.

Damit z. B. die große Datendurchsatzrate erreicht werden kann, ist es nötig auf Dateien und deren Inhalte parallel zuzugreifen und einzelne Teile einer Datei aus mehreren Quellen zu lesen, um nicht an einer Stelle einen Engpass zu erhalten.

4.2.3 Architektur

Die Architektur des GPFS spiegelt die im vorhergehenden Abschnitt gelisteten Anforderungen wieder. So wird die Skalierbarkeit, was sowohl die Durchsatzrate als auch die Speicherkapazität angeht, durch den Zusammenschluss vieler Fesplattten zu einem Verbund von gemeinsam genutzten Platten erreicht. Auf diese Platten kann wiederum von mehreren Clients zugegriffen werden. Abbildung 5 zeigt die abstrakte Architektur des GPFS. Zwischen den Clients und dem Festplattenverbund befindet sich ein Netzwerk (*switching fabric*), zumeist ein Storage Area Network (SAN), oder alternativ übernehmen einzelne Knoten die Funktion von E/A-Knoten (welche den Zugriff auf „ihre“ Festplatten steuern). Für die Festplatten selbst gibt es keine weiteren speziellen Anforderungen außer den konventionellen Zugriff auf einzelne Blöcke.

4.2.4 Striping

Zum einen setzt das GPFS auf Striping, um die Last beim Abruf einer Datei besser zu verteilen, zum anderen um eine bessere Fehlertoleranz gewährleisten zu können. Allein durch die Verteilung der Last auf mehrere Netzwerkadapter, Festplatten-Controller und Festplatten kann die Ausfallwahrscheinlichkeit für eine Datei erhöht werden, da nicht alle Zugriffe immer den gleichen Weg nehmen.

Als Standardgröße für einzelne Dateifragmente setzt das GPFS auf 256 kB-Blöcke, die jedoch für spezielle Fälle auch im Bereich von 16 kB bis 1 MB angepasst werden können. Mehrere aufeinanderfolgende Blöcke werden mittels Round-Robin-Verfahren auf die Festplatten und Speicherknoten verteilt. Hierfür wird die Nähe einzelner Knoten zueinander beachtet, die durch Kenntnis der Topologie des Netzwerks ermittelt wird. Die Topologie muss dafür von einem Administrator vorgegeben werden.

Ein Fakt der beachtet werden sollte, ist dass Striping für das GPFS am besten funktioniert, wenn alle Festplatten die gleiche Größe und Performance haben, da unterschiedlich große Festplatten dazuführen, dass auf den großen Festplatten mehr Daten abgelegt werden und somit potentiell auch mehr Zugriff darauf erfolgen.

4.2.5 Journaling/Logging

Das GPFS nutzt ein sogenanntes Write-Ahead-Log für das Journaling und Logging in dem keine Nutzerdaten gespeichert werden. Für jeden Speicherknoten gibt es ein eigenes Log in dem die Änderung an dessen Metadaten protokolliert werden. Dies hat den Vorteil, dass im Falle eines Ausfalls dieses Log von einem beliebigen anderen Knoten gelesen werden kann (alle Knoten können die Logs lesen) und der lesende Knoten eine Wiederherstellung durchführen kann.

Die Wiederherstellung erfolgt in dem einfach alle aufgezeichneten Änderungen nochmals ausgeführt werden. Das sorgt für eine schnelle Wiederherstellung der Konsistenz, ohne dass auf die Wiederkehr des ausgefallenen Knotens gewartet werden muss.

4.2.6 Distributed Locking

Da das GPFS POSIX-konform ist, müssen alle Zugriffe auf Daten und Metadaten synchron erfolgen, was die mögliche Geschwindigkeit beim parallelen Zugriff jedoch verringert.

Um den parallelen Lese- und Schreibzugriff auf das Dateisystem zu erlauben, setzt das GPFS auf den Ansatz des Distributed Locking. Dabei sorgt ein Distributed Locking Protokoll dafür, dass gerade beim Schreiben keine Daten und vor allem keine Metadaten korrumpiert werden. Das Protokoll sichert, dass Änderungen (Schreiboperationen) die auf einem Knoten gemacht werden, für lesende Zugriffe entweder komplett oder gar nicht sichtbar sind.

Da sich Distributed Locking als problematisch und zeitkritisch herausstellt wenn gewisse (Meta-)Daten eines Knotens häufiger geändert werden (ständige Lock-Konflikte), gibt es zusätzlich die Möglichkeit der zentralisierten Verwaltung von Änderungen. Hierbei übernimmt ein Knoten zentral die Aufgabe Änderungen an den Daten durchzuführen.

4.2.7 Quorum & Fencing

Damit das GPFS eine Partitionierung erkennen und darauf reagieren kann, gibt es Monitore, die die einzelnen Knoten und deren Kommunikationspfade überwachen. Dies geschieht durch das regelmäßige Senden von Heartbeat-Nachrichten. Sollte eine diese Heartbeat-Nachrichten nicht zurückkommen, so informiert der Monitor die anderen Knoten über den Ausfall, welches widerum Wiederherstellungsfunktionen startet.

Sollte es einen Ausfall der Kommunikation zwischen den Clients geben, z. B. durch einen defekten Netzwerkadapter oder eine loses Kabel, und eine Partionierierung vorliegen, so kann die weiterhin unabhängige voneinander stattfindende Kommunikation der Partitionen zu inkonsistenten Daten führen.

Damit dies nicht passiert arbeitet das GPFS mit einem Quorum, wodurch der Partition mit der Mehrheit der Clients weiterhin erlaubt wird das Dateisystem zu nutzen, während die Minderheit den Zugriff stoppen muss, bis sie wieder zur Mehrheit gehört. Da gleichzeitig mit dem Ausfall die Mehrheit mittels Logs versucht eine Wiederherstellung der ausgefallen Knoten durchzuführen, ist das Abgrenzen der Minderheit (*Fencing*) notwendig. Hierfür werden alle gemeinsam genutzen Festplatten darüber informiert, dass sie die Lese- und Schreibzugriffe für die Clients der Minderheit einstellen sollen.

Für den Fall, dass zwei gleich große Partitionen entstanden sind, versuchen beide Partitionen nach einer zuvor festgelegten Reihenfolge die Festplatten für sich zu blockieren und der Teil, der dabei schneller ist darf weiterhin das Dateisystem nutzen.

4.2.8 Replikation

Da das GPFS die Daten und Metadaten zur besseren Lastverteilung über mehrere Festplatten verteilt, würde ein Ausfall einer Festplatte zu einem Verlust von überproportional vielen Festplatten führen. Aus diesem Grund wird auf Ebene der Festplatten von dem GPFS ein RAID eingesetzt, das den Ausfall einer Festplatte ohne Probleme verkraften kann.

Als Alternative dazu (oder auch zusätzlich) kann eine Replikation auch auf der Dateisystemebene stattfinden. Ist diese Replikation aktiviert, so legt GPFS immer zwei Kopien (Original plus ein Replikat) eines Daten- und Metadatenblocks ab. Diese werden nach Möglichkeit immer auf unterschiedlichen Festplatten gespeichert und bestenfalls auch auf unterschiedlichen Knoten.

Beide Kopien werden synchron gehalten bis im Falle eines Ausfalls nur noch eine Kopie übrig ist. Diese wird für einen gewisse Zeit als einzige Kopie verwendet bevor dann – im Fall des längeren Ausfalls der zweiten Kopie – eine neue Kopie erzeugt wird. Kommt die erste Kopie noch innerhalb der Wartezeit zurück, so wird sie mit Hilfe des Journals auf den aktuellen Stand gebracht und kann danach wieder normal verwendet werden.

Konfigurationen für die Replikation können separat für Daten und Metadaten vorgenommen und somit aktiviert werden. Meist ist es wichtiger eine weitere Kopie der Metadaten zu haben (vgl. Abschnitt 3.5).

4.2.9 Zusätzliche Techniken

Das GPFS setzt zusätzlich zu den zuvor beschriebenen Techniken auf die Erkennung von Zugriffsmustern beim Lesen und Schreiben von Dateiinhalten. Erkannt werden sequenzielle Vorwärts- und Rückwärtslesen sowie verschiedene Arten von Sprüngen.

4.3 Google File System

4.3.1 Allgemeines

Das Google File System (auch: **GFS**) [HGL03] wurde von Google nach den eigenen Anforderungen entwickelt. Dabei wurden auf Basis von Beobachtungen der eigenen Bedürfnisse und mit Hilfe von Annahmen über zukünftige Entwicklungen die Anforderungen festgelegt.

Heutzutage wird es fast überall bei Google als Speicherplattform verwendet. Dazu zählen sowohl die im Internet angebotenen Dienste als auch die noch in der Forschung und Entwicklung befindlichen Systeme. Demnach setzten fast alle Entwickler bei Google das Google File System für ihre Zwecke und die dabei entstehende Software ein.

4.3.2 Anforderungen

Als eine der Anforderungen an das Dateisystem galt, dass es möglichst auf kostengünstiger Standard-Hardware lauffähig sein muss. Dies ist seit der Gründung von Google eine Tatsache, da schon zur Anfangszeit ausschließlich Standard-Hardware für die Entwicklung und die Bereitstellung der Dienste genutzt wurde. Dazu kommt, dass eine weitere Anforderung ist, dass eine hohe, aggregierte Performance mit der gegebenen Hardware erreichbar wird und dadurch viele parallele Zugriffe ermöglicht werden.

Zu den Einschränkungen, die eine Verbesserung der Leistung möglich machen, gehört, dass es vor allem für große Dateien und Anfügevorgänge an diese Dateien optimiert sein kann. Aus Sicht des Zugriffs wird meist nur sequenziell gelesen, vor allem aber ist Streaming wichtig und somit das direkte Auslesen gerade geschriebener Daten.

Grundlegend ist das Google File System nicht POSIX-konform (Erläuterungen dazu siehe Abschnitt 3.1), wodurch die Entwicklung einer eigenen API zum Zugriff aus den darauf aufbauenden Anwendungen notwendig wurde.

4.3.3 Architektur

Die Architektur des Google File System sieht einen Master-Server vor, der die Metadaten verwaltet, und mehrere sogenannte Chunk-Server (allgemein Speicherknoten genannt)[8], die den Inhalt der eigentlichen Dateien speichern. Diese Chunk-Server besitzen als grundlegendes Dateisystem für die Festplatten ein EXT2-Dateisystem, welches zunehmend durch das EXT4-Dateisystem ersetzt wird. Darauf aufsetzend ist das verteilte Google File System eingerichtet.

Parallel zu den zum verteilten Dateisystem gehörenden Master- und Chunk-Servern existieren die verschiedenen Clients, die mit ihren Anwendungen auf das Dateisystem zugreifen.

Abbildung 6 zeigt die Architektur des Google File System und welche Kommunikation zwischen den einzelnen Computern stattfindet.

4.3.4 Replikation

Obwohl Google mit seinem verteilten Dateisystem auf kostengünstige Standard-Hardware setzt, wird RAID nicht als eine einfache Art zu replizieren verwendet. Stattdessen wird

[8] Die Chunk-Server sind nach den Dateifragmenten – *Chunks* – benannt, die sie speichern. Diese Chunks haben ein Größe von 64MB voran man erkennt, dass Google mit sehr großen Dateien rechnet.

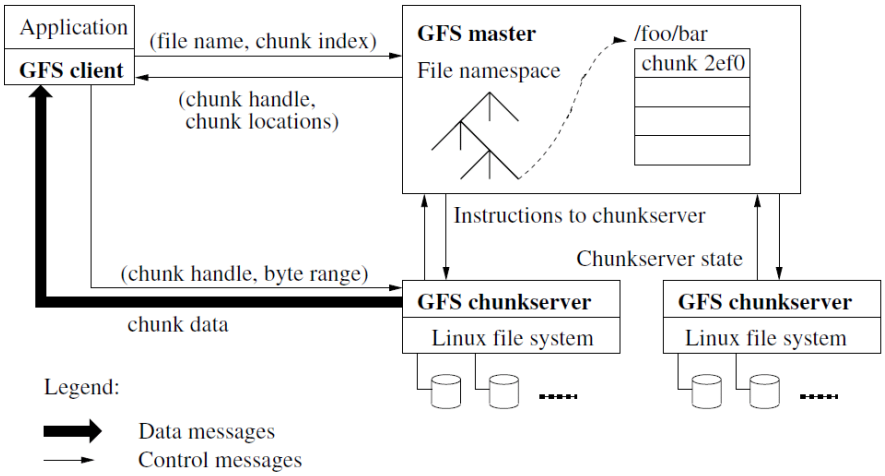

Abbildung 6: Architektur des Google File System [HGL03]

auf eine eigenständige Replikation der Chunks durch das Dateisystem gesetzt, die Dank relativ günstigen und großen Festplatten möglich ist.

Standardmäßig werden drei Replikate eines jeden Chunks erstellt, von denen der Master-Server mittels Heartbeat-Nachrichten an die Chunk-Server sicherstellt, dass alle davon noch existieren und auch zur Verfügung stehen. Erkennt der Master, dass eines der erforderlichen Replikate fehlt, so legt er automatisch eine neues an (was einer Art *Self-Healing* entspricht).

Neben den Dateiinhalten auf den Chunk-Servers werden auch die Metadaten und Logs des Masters repliziert (siehe Abschnitt 4.3.6).

4.3.5 Striping

Ein Striping der Dateiinhalte über mehrere Festplatten und Knoten findet durch die Zerlegung der Dateien in Chunks und deren Verteilung statt. Hierfür wird die Topologie des Netzwerks berücksichtigt, die automatisch aus den IP-Adressen der Server abgeleitet wird. Zwei Knoten sind demnach weit voneinander entfernt, wenn ihre IP-Adressen möglichst unterschiedlich sind.

Mit dieser Vorgehensweise soll zum einen sichergestellt werden, dass nicht nur eine Festplatte oder ein Festplatten-Controller ausfallen kann, sondern stattdessen sogar ein kompletter Speicherknoten des Clusters. Zum anderen ergeben sich durch eine starke Verteilung der Chunks über das Netzwerk aber auch Vorteile bei der Lastverteilung. So können nicht nur die Datenraten der Festplatten ausgenutzt werden, sondern auch die Datenraten der Netzwerkkadapter, wodurch eine noch schnellere Bereitstellung der Daten einer kompletten Datei ermöglicht wird.

4.3.6 Separate Metadaten

Das Google File System verwendet einen zentralen Master für die Verwaltung der Metadaten. Damit dieser Server nicht zum *Single Point of Failure* wird, werden die Daten des Master als Schattenkopien auf weitere Server repliziert. Diese dienen jedoch nicht zur

Bearbeitung von Anfragen, sondern existieren nur für den Fall eines Ausfalls des Master-Servers.

Der Master-Server hat mit den Metadaten den Überblick über Chunks und deren aktuellen Status. So kennt er u.a. die Datei- und Chunk-Namensräume, weiß wie die Zuordnung der Chunks zu den einzelnen Dateien ist und kennt ebenfalls die Orte an denen Replikate eines Chunks abgelegt sind. Der Status zu jedem Chunk gibt ihm Auskunft darüber, ob es notwendig ist ein weiteres Replikat anzufertigen oder ob die Anzahl der gerade verfügbaren Replikate ausreicht.

Alle Metadaten werden vom Master im Hauptspeicher gehalten, damit darauf ein schneller Zugriff möglich ist. Zusätzlich werden, um eine weitere Sicherheit bei Stromausfällen zu gewährleisten, die Metadaten auch auf die Festplatte persistiert. Da die Anzahl der verwaltbaren Chunks und Dateien durch die Größe des Arbeitsspeichers limitiert wird, lässt sich darüber indirekt steuern wie viele Dateien in dem verteilten Dateisystem abgelegt werden können. Sollte die Grenze der maximalen Anzahl an Dateien und Chunks erreicht werden, so muss (nur) die Größe des Hauptspeichers erhöht werden.

4.3.7 Journaling/Logging

Der schon in den vorhergehenden Abschnitten beschriebe Master führt auch ein Journal/Log aller Dateisystem-Operationen. Dieses Log enthält die Versionsnummer für jeden Chunk und diese Versionsnummer erhöht sich bei jeder Schreiboperation. So lassen sich eventuell veraltete Replikate erkennen, die durch einen kurzzeitigen Ausfall der Kommunikation oder des entsprechenden Knotens entstehen können. Mit Hilfe des Journals lassen sich jedoch die fehlenden Operationen auf einem Chunk ausführen, womit dieser wieder einen aktuellen Stand erreichen und erneut von dem Dateisystem verwendet werden kann.

Zusätzlich zu diesem Journal, welches die Dateioperationen überwacht und protokolliert, existieren noch weitere Logs, die im Fehlerfall helfen einen konsistenten Zustand des verteilten Dateisystems wiederherzustellen:

Diagnose-Log Das Diagnose-Log enthält alle wichtigen Ereignisse, die das Dateisystem betreffen. Hierzu zählen u.a. das Hoch- und Herunterfahren eines Chunk-Servers.

Interaktions-Log In dem Interaktions-Log wird jegliche Kommunikation via RPC (*remote procedure call*) festgehalten. Dabei wir bei jedem RPC die komplette Anfragen und deren Antworten protokolliert. Ausgenommen dabei sind die übertragenen Dateiinhalte, die die Logs nur unnötig aufblähen würden.

Nimmt man die Logs aller Knoten zusammen, so lassen sich dadurch erhaltene und gesendete Anfragen und Antworten vergleichen und so Probleme in der Kommunikation aufdecken.

Zusammengefasst machen alle drei Logs – das Journal, das Diagnose-Log und das detaillierte Interaktions-Log – ein Online-Monitoring möglich, wenn die letzten Ereignisse jeweils zum schnelleren Abrufen im Hauptspeicher gehalten werden. Dies findet so auch bei Google seine Anwendung.

4.3.8 Locking & Leases

Das Google File System kennt Lese- und Schreibvollmachten (*Leases*). Diese werden von dem Master-Server an einen Client vergeben wodurch dieser Client den Zugriff auf eines der Chunk-Replikate erhält, welches für diesen Client dann der primäre Chunk ist. Dieser primäre Chunk legt die im Falle eines Schreibvorgangs genutzte Reihenfolge in der Replikate Änderungen bekommen selbst fest.

Jede Vollmacht hat ein Verfallsdatum, welches standardmäßig auf 60 Sekunden gesetzt ist. So wird der Kommunikationsbedarf zwischen Client und Master reduziert, wenn hintereinander mehrere Aktionen auf einem Chunk durchgeführt werden. Das Verfallsdatum stellt weiterhin sicher, dass – im Falle eines Ausfalls des Clients – spätestens 60 Sekunden später ein anderer Client die Vollmacht bekommen kann. Dieses Verfallsdatum kann von dem Client aber auch herausgeschoben werden, in dem er den Master nach einer Verlängerung fragt. Ebenso kann der Master eine Vollmacht vorzeitig zurückziehen.

Diese Vollmachten sollen verhindern, dass gewisse Aktionen parallel passieren. So kann dadurch z. B. abgesichert werden, dass während einer Umbenennung kein Client probiert Dateiinhalte zu lesen, denn für Dateioperationen wie das Umbennenen wird sowohl die Lese- als auch die Schreibvollmacht benötigt.

4.3.9 Zusätzliche Techniken

Zusätzlich zu den generell herausgearbeiteten Techniken, welche in den vorangegangenen Abschnitten genauer betrachtet wurden, setzt das Google File System auf eine paar weitere Fehlertoleranz-Mechanismen.

Hierzu zählt die Möglichkeit der Erstellung von **Snapshots** und **Checkpoints**. Snapshots können genutzt werden, um aus einer Version einer Datei eine Verzweigung (*branching*) möglich zu machen. Die dabei genutzte Originaldatei bleiben im weiteren Verlauf unberührt, während auf der entstanden Kopien weitergearbeitet werden kann. Checkpoints hingegen werden häufig genutzt um den Zustand einer Version einer Datei so einzufrieren, dass bei der Wiederherstellung im Falle eines Fehlers von diesem Checkpoint wieder begonnen werden kann. Von da an müssen nur noch ausstehende Journaleinträge abgearbeitet werden und der Knoten ist wieder auf dem aktuellen Stand.

Weiterhin unterstützt das Dateisystem das Bilden von **Checksummen** über Chunk-Daten. Hierfür kommt ein spezieller Algorithmus zum Einsatz, der besonders auf das Anhängen von Daten an das Ende eines Chunks optimiert ist.

Das Google File System verzichtet auf Caching, damit eventuell mögliche Cache-Kohärenz-Probleme, die nach Änderungen an Daten in nur einer Chunk-Kopie zu inkonsistenten Daten führen können, nicht auftreten. Weiterhin gibt es eine Art **Garbage Collection** für Chunks und Logs, die belegten, doch nicht mehr benötigten Speicherplatz freiräumt.

Zusätzlich dazu sind Schreib- und Löschoperationen besonders sicher implementiert. Beim Schreiben gibt es **Recovery Blocks**, die verhindern sollen, dass ein Fehler beim Schreiben nicht bemerkt wird. So wird, im ersten Versuch nachdem das Schreiben eines Replikats fehlgeschlagen ist, ein zweiter Versuch unternommen. Schlägt auch dieser Schreibvorgang fehl, so wird der gesamte Schreibvorgang abgebrochen und der Client dazu aufgefordert die Daten noch einmal in einem neuen Schreibauftrag zu schicken (wodurch ein neuer Platz zum Schreiben des Chunks ausgewählt wird). Beim Löschen einer Datei wird diese nur versteckt und kann innerhalb eines Zeitraums von drei Tagen noch

wiederhergestellt werden. Sollte dies in dieser Zeitspanne nicht passieren, so werden alle dazugehörigen Chunks beim einem der nächsten Anläufe des Garbage Collectors entfernt.

Als letzte hier beschriebene Technik die Fehlertoleranz zu erhöhen, setzt Google auf den Ansatz der **Rejuvenation**. Hierbei ist es prinzipiell immer möglich einen laufenen Prozess des Dateisystems zu töten und durch einen Neustart des Prozesses wieder in einen fehlerfreien Zustand zu kommen. Das dabei verwendete Paradigma ist damit eine schnelle Vorwärts-Wiederherstellung.

5 Zusammenfassung

Die vorgestellten Dateisysteme wurden absichtlich so gewählt, dass sie verschiedene Bereiche und Entwicklungsstufen umfassen. Sie alle zeigen wie verschiedene Techniken zusammen eingesetzt werden, um hochverfügbare Systeme zu bauen. Eine detaillierte Analyse der erzielten Verfügbarkeit erwies sich als zu schwierig, weil jedes der betrachteten Systeme für sich wiederum mehrere, konfigurierbare Parameter anbietet. Der erfolgreichen Einsatz von etablierten verteilten fehlertoleranten Dateisystemen wie GPFS und Lustre im Supercomputing-Bereich stellen die Leistungsfähigkeit und Zuverlässigkeit dieser Konzepte in der Praxis unter Beweis.

Literatur

[Bon10] Jan Bonér. Scalability, Availability & Stability Patterns, Mai 2010. Online unter: http://www.slideshare.net/jboner/scalability-availability-stability-patterns.

[Bra99] Peter J. Braam. File Systems for Clusters from a Protocol Perspective, 1999. Online unter: http://www.cs.cmu.edu/~odyssey/docdir/extremelinux99.pdf.

[Bra02] Peter J. Braam. Lustre: the intergalactic file system. In *Proceedings of the 2002 Linux Symposium, Ottawa, Canada*, 2002. Online unter: http://www.linuxsymposium.org/archives/OLS/Reprints-2002/braam-reprint.pdf.

[CPS95] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813 (Informational), Juni 1995. Online unter: http://www.ietf.org/rfc/rfc1813.txt.

[Day08] Shobhit Dayal. Characterizing HEC Storage Systems at Rest. Technical Report CMU-PDL-08-109, Carnegie Mellon University Parallel Data Lab, Juli 2008. Online unter: http://www.pdl.cmu.edu/PDL-FTP/PDSI/CMU-PDL-08-109_abs.shtml.

[HGL03] Sanjay Ghemawat Howard, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *In Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–43, Oktober 2003. Online unter: http://labs.google.com/papers/gfs-sosp2003.pdf.

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and

Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988. Online unter: http://www.opengroup.org/onlinepubs/9699919799/.

[IG08] The IEEE and The Open Group. *The Open Group Base Specifications Issue 7 IEEE Std 1003.1-2008*. The IEEE and The Open Group, 2008. Online unter: http://www.opengroup.org/onlinepubs/9699919799/.

[KLS86] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXcluster: a closely-coupled distributed system. *ACM Trans. Comput. Syst.*, 4(2):130–146, 1986. Online unter: http://portal.acm.org/citation.cfm?id=214419.214421.

[LPGM08] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*. USENIX Association, 2008. Online unter: http://www.usenix.org/events/usenix08/tech/full_papers/leung/leung.pdf.

[Sat90] M. Satyanarayanan. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39:447–459, 1990. Online unter: http://www.cs.cmu.edu/~satya/docdir/coda.pdf.

[SG06] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN2006)*, 2006. Online unter: http://www.pdl.cmu.edu/PDL-FTP/stray/dsn06.pdf.

[SG07] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX conference on File and Storage Technologies*. USENIX Association, 2007. Online unter: http://www.usenix.org/events/fast07/tech/schroeder.html.

[SH02] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *In Proceedings of the 2002 Conference on File and Storage Technologies (FAST*, pages 231–244, Januar 2002. Online unter: http://www.cct.lsu.edu/~kosar/csc7700-fall06/papers/Schmuck02.pdf.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *In Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, Mai 2010. Online unter: http://storageconference.org/2010/Papers/MSST/Shvachko.pdf.

[SS07] H. Shan and J. Shalf. Using IOR to Analyze the I/O performance for HPC Platforms. Technical Report LBNL-62647, National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, 2007. Online unter: http://www.nersc.gov/projects/SDSA/reports/uploaded/cug07_shan.pdf.

[ST87] William E. Snaman and David W. Thiel. The VAX/VMS Distributed Lock Manager. *Digital Technical Journals*, (5):29–44, 1987. Online unter: http://www.dtjcd.vmsresource.co.uk/pdfs/dtj_v01-05_sep1987.pdf.

[TvS07] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, 2007.

[VLR+08] M. Vilayannur, S. Lang, R. Ross, R. Klundt, and L. Ward. Extending the POSIX I/O Interface: A Parallel File System Perspective. Technical Report ANL/MCS-TM-302, Oktober 2008. Online unter: http://www.ipd.anl.gov/anlpubs/2008/12/63022.pdf.

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, November 2006. Online unter: http://www.usenix.org/events/osdi06/tech/full_papers/weil/weil.pdf.

[Wei07] Sage A. Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California Santa Cruz, Dezember 2007. Online unter: http://ceph.newdream.net/weil-thesis.pdf.