Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Technische Informatik, Zuverlässige Systeme

Konsistentes Hinzufügen und Entfernen von Replikaten in XtreemFS

Johannes Dillmann
Matrikelnummer: 4476004
jdillmann@inf.fu-berlin.de

Betreuer: Dr. Florian Schintke
Eingereicht bei: Prof. Dr. Katinka Wolter

Berlin, 10.09.2013

Zusammenfassung

In replizierten Systemen muss es möglich sein ausgefallene Server zu ersetzen und Replikate dynamisch hinzuzufügen oder zu entfernen. Veränderungen an der Menge der eingesetzten Replikate können zu inkonsistenten Daten führen. In dieser Arbeit wird ein Algorithmus für das konsistente Hinzufügen und Entfernen von Replikaten für das verteilte Dateisystem XtreemFS vorgestellt. Der Algorithmus nutzt einen fehlertoleranten Service, der die Replikatliste speichert sowie Veränderungen koordiniert und diese aus globaler Sicht atomar ausführt.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

10.09.2013

Johannes Dillmann

Inhaltsverzeichnis

1 Einleitung

Die Nachfrage nach und die Anforderungen an verteilte Dateisysteme steigen mit der fortschreitenden Verbreitung von Cloud Anwendungen und verteilten Systemen. Um in entsprechenden Systemen Skalierbarkeit zu ermöglichen und ihre Verfügbarkeit und Leistung zu verbessern, können Replikate von Dateien auf mehreren Servern gespeichert werden. Veränderungen der Dateien müssen dann jedoch auf allen Replikaten konsistent ausgeführt werden. Hierzu benötigen sowohl Clients, als auch Verwaltungskomponenten, Informationen über die beteiligten Replikate. Werden Replikate hinzugefügt, entfernt oder ersetzt, muss sichergestellt werden, dass die Änderung der Replikatliste allen Komponenten bekannt gemacht wird.

Veränderungen an der Replikatliste sind notwendig, wenn Server dauerhaft ausfallen und ersetzt werden. Zudem ist die manuelle Anpassung des Replikationsfaktors, also die Anzahl der Replikate, die für eine Datei existieren, nützlich, da hierüber die Belastbarkeit des Systems (Load), sowie die Verfügbarkeit (Availability) und Beständigkeit (Durability) der Daten dynamisch beeinflusst werden kann. Des Weiteren kann über das Verschieben von Replikaten eine Lastbalancierung erreicht werden.

Um die Konsistenz der Daten zu garantieren, müssen Operationen mit einer konsistenten Sicht der Replikatlisten ausgeführt werden. Finden durch einen Schreibzugriff Veränderungen an den Daten statt, darf keine darauf folgende Leseoperation die unveränderten, alten Daten zurückgeben. Werden jedoch neue Replikate hinzugefügt oder auf Basis einer veralteten Replikatliste Daten geschrieben, ist nicht garantiert, dass alle Replikate der gültigen Replikatliste aktuell sind. Ein darauf folgender Lesezugriff auf ein nicht aktualisiertes Replikat gibt somit alte Daten zurück. Analog treten Inkonsistenzen auf, wenn auf Basis einer veralteten Replikatliste von einem Replikat gelesen wird, welches schon entfernt und bei Schreibvorgängen nicht mehr aktualisiert wurde.

Im Rahmen dieser Bachelorarbeit wird ein Algorithmus für das konsistente Hinzufügen und Entfernen von Replikaten für das am Zuse-Institut Berlin (ZIB) entwickelte verteilte Dateisystem XtreemFS entwickelt. Der Algorithmus folgt dem von Schiper und Toueg vorgeschlagenen Ansatz der Set Membership Verwaltung durch einen replizierten Koordinator [ST06] und stellt

sicher, dass Veränderungen an der Replikatliste aus globaler Sicht atomar geschehen und Zugriffe, die auf veralteten Replikatlisten basieren, verhindert werden. Des Weiteren trägt er dafür Sorge, dass die Konsistenz der Daten bei Veränderungen der Replikatliste erhalten bleibt und nimmt gegebenenfalls notwendige Aktualisierungen der Replikate vor.

Im folgenden Abschnitt 2 werden die Grundlagen verteilter Dateisysteme diskutiert und eine Übersicht über die Architektur von XtreemFS gegeben. Dabei werden die verschiedenen Komponenten und die eingesetzten Replikationsverfahren vorgestellt.

Darauf aufbauend wird in Abschnitt 3 der Algorithmus zum konsistenten Hinzufügen und Entfernen von Replikaten in mehreren Iterationen entwickelt. Dazu werden konkrete Fehlerfälle, die zu Inkonsistenzen führen können, besprochen und Erweiterungen des Algorithmus vorgestellt, die deren Auftreten verhindern.

Der dabei entwickelte Algorithmus wurde für die Java basierten Server in XtreemFS implementiert und in deren Rahmen getestet. In Abschnitt 4 werden die durchgeführten Testfälle vorgestellt und die Fehlertoleranz, Laufzeit sowie der Nachrichtenaufwand analysiert.

Abschließend werden in Abschnitt 5 verwandte Arbeiten und alternative Lösungsansätze diskutiert und in Abschnitt 6 ein zusammenfassendes Fazit gezogen, sowie ein Ausblick auf mögliche Weiterentwicklungen gegeben.

2 Grundlagen

2.1 Verteilte Dateisysteme

Für verteilte Systeme wie bspw. Cloudanwendungen ist es erforderlich über das Netzwerk nutzbaren Speicher zur Verfügung zu stellen. Da der Einsatz zentralisierter Dateisysteme nicht skaliert und bei einer steigenden Anzahl von Zugriffen die Last auf den Dateiserver zu groß werden kann, werden oftmals Cloud Storage Systeme eingesetzt, welche die Last auf mehrere Server verteilen. Verteilte Dateisysteme sind eine spezielle Art von Cloud Storage, die einen hierarchischen Namensraum und lokale Dateisystemsemantiken unterstützen.

Wenn Daten in verteilten Systemen gespeichert werden ist *Verlässlichkeit* bzw. *Verfügbarkeit* eine unumgängliche Anforderung. Der Zugriff auf die Daten muss auch dann möglich sein, wenn Teile des Netzwerks oder des Systems ausfallen (bspw. aufgrund von Netzwerkpartitionen, Hardwarefehlern oder Stromausfällen). Solche Fehler sollen, soweit das möglich ist, transparent behandelt werden und keine Auswirkungen auf die Benutzung des Systems haben.

Cloud Storage ermöglicht den Zugriff auf unbeschränkten Speicherplatz bei Bedarf ("on demand"). Dies hilft gegen die Überdimensionierung der Hardware, da zu Beginn keine bestimmte (maximale) Größe festgelegt werden muss. Um dies zu ermöglichen muss Speicherplatz jedoch *dynamisch* bzw. *elastisch skalierbar* sein, ohne längere Ausfallzeiten oder Veränderungen in der Zugriffssemantik (bspw. neue Adressen). Dies wird erreicht indem statt vertikal ("scale up"), horizontal ("scale out") skaliert wird. Dabei werden nicht wie sonst die bestehenden Hardwareressourcen in einer bestimmten, zentralen Komponente durch leistungsfähigere Ressourcen erweitert oder ersetzt, sondern neue Komponenten nahtlos in das dezentrale, verteilte System eingebunden.

Da die Datenspeicher in einem Cloud System nicht nur über ein speziell gesichertes Netzwerk verfügbar sind und die einzelnen Ressourcen aus Effizienzgründen von mehreren Anwenderinnen zugleich genutzt werden, muss für eine hohe *Sicherheit* und *Isolation* der Daten gesorgt werden. Um den Zugriff und damit die Integrität der Daten abzusichern, werden daher Au-

thentifizierungsmechanismen eingesetzt. Die Vertraulichkeit der Daten kann hingegen mit Verschlüsselungsverfahren sicher gestellt werden.

2.2 XtreemFS Architektur

XtreemFS ist ein verteiltes und repliziertes Dateisystem [HCK+08]. Es folgt dem Konzept objektbasierter Datenspeicher, die die Vorteile von NAS (Network Attached Storage) und SAN (Storage Area Network) Systemen kombinieren [MGR03, FMN+05]. Die Dateiinhalte werden in Objektcontainern organisiert, die auf Object Storage Devices (OSD) gespeichert werden und über eine globale Kennung direkt adressiert werden können. Da dies ohne die Kommunikation mit einer zentralisierten Komponente möglich ist, bleibt das System leistungsfähig und skalierbar.

XtreemFS implementiert eine POSIX [IEE08] kompatible Schnittstelle. Metadaten wie Verzeichnisstrukturen, Dateinamen, Zugriffsrechte oder Replikationsparameter, deren Organisation rechenintensiv ist, werden von einem separaten Service, dem Metadata and Replica Catalog (MRC) verwaltet. Auf den OSDs findet nur die I/O intensive Verwaltung der Dateiinhalte statt. Neben diesen beiden Services, existiert in XtreemFS noch ein Client und ein Directory Service (DIR) (siehe Abbildung 1).

2.2.1 Object Storage Devices (OSD)

Die OSDs speichern die Dateiinhalte persistent auf lokalen Datenträgern und stellen eine Schnittstelle zum Lesen, Schreiben oder Löschen von Daten bereit. Sie haben keinen Zugriff auf die Metadaten der Dateien und sind auf die MRCs zur Authentifizierung der Clients angewiesen. Neben den grundsätzlichen Dateioperationen sind die OSDs für die konsistente Replikation der Daten zuständig.

2.2.2 Metadata and Replica Catalog (MRC)

Der MRC speichert die Metadaten der in Volumes gruppierten Dateisysteme. Dies umfasst unter anderem Verzeichnisbäume, Dateinamen und -größen, Zugriffsrechte und erweiterte Dateiattribute. Daneben verwaltet er aber auch die Liste der Replikate, die für eine Datei existieren. Intern nutzen MRCs

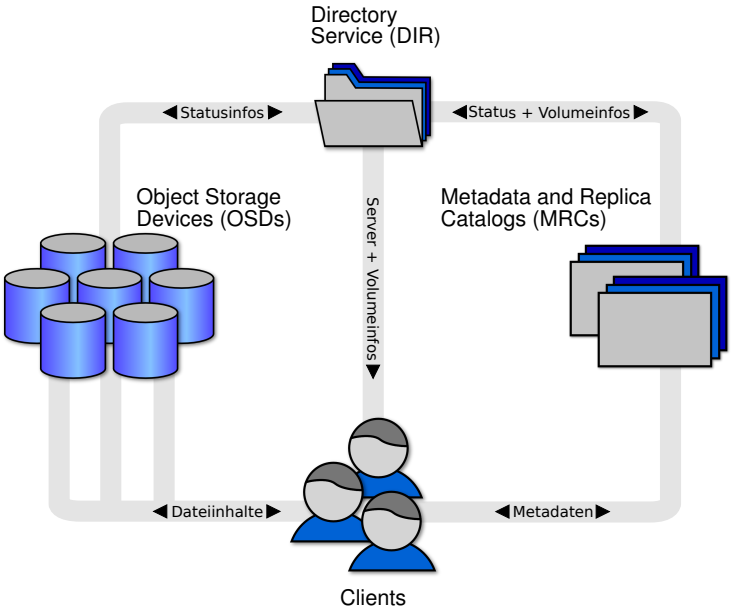

Abbildung 1: XtreemFS Architektur (Abb. nach [SBR13]).

die nicht-relationale Datenbank BabuDB [SKHH10], die durch den Einsatz transparenter Replikation die Fehlertoleranz erhöht und Failovers ermöglicht.

Eine weitere Aufgabe des MRC ist die Autorisierung der Clients und die Autorisierung von Dateioperationen. Will ein Client auf eine Datei zugreifen, muss er sich erfolgreich beim MRC authentifizieren und erhält von diesem bei ausreichenden Zugriffsrechten ein zeitlich beschränktes Sicherheitstoken, genannt *Capability*. Die Capability wird vom MRC signiert und bei Zugriffen auf den OSDs von diesen überprüft bevor eine Operation ausgeführt wird und ermöglicht somit die asynchrone Autorisierung ohne Kommunikation zwischen OSDs und MRC.

2.2.3 Client

Über den Client kann ein Volume in das lokale Dateisystem eingebunden werden. Der Client interpretiert lokale POSIX konforme Systemaufrufe und übersetzt diese in entsprechende Anfragen an die XtreemFS Dienste. Dazu greift er auf die plattformübergreifende Bibliothek *libxtreemfs* zurück.

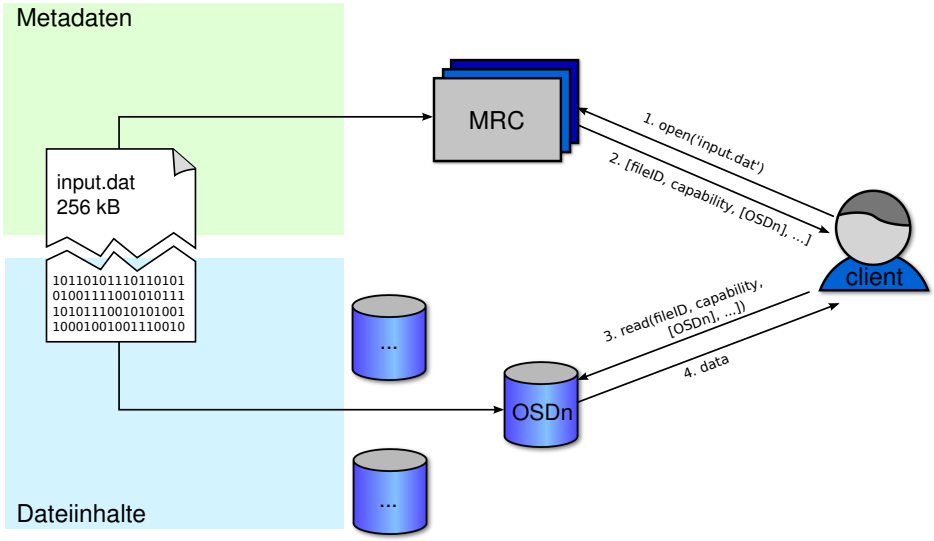

Abbildung 2: Dateizugriffe in XtreemFS (Abb. nach [SBR13]).

Dateizugriffe

Dateizugriffe folgen in XtreemFS der POSIX Semantik und finden in mehreren Phasen statt (siehe Abbildung 2). Beim Öffnen von Dateien kontaktieren Clients den MRC (1) und erhalten von diesem neben der globalen `fileID` die Liste der für die Datei zuständigen Replikate und eine Capability (2). Nachfolgende Schreib- oder Leseoperationen werden direkt an die OSDs gestellt (3). Die OSDs prüfen die übergebene Capability und führen die Operationen auf Basis der vom Client übertragenen Replikatliste aus.

2.2.4 Directory Service (DIR)

Als zusätzlicher Service agiert der DIR als Nameserver und zentrale Registrierungsstelle für alle XtreemFS Dienste sowie die verfügbaren Volumes. MRCs und OSDs registrieren sich initial beim DIR mit einer eindeutigen Kennung und senden ihm regelmäßige Statusinformationen. Innerhalb von XtreemFS werden statt direkter Adressen, global eindeutige Kennungen zur Identifizierung der Dienste verwendet und über den DIR aufgelöst. Zudem erhalten MRCs vom DIR Informationen über verfügbare OSDs sowie Clients die Kennungen der für ein Volume verantwortlichen MRCs. Der DIR nutzt die replizierte Datenbank BabuDB und ist damit fehlertolerant.

2.3 Replikation

Um die Verfügbarkeit, aber auch die Leistung verteilter Dateisysteme zu erhöhen, können von Dateien identische Kopien auf verschiedenen Servern angelegt werden. Eine solche Kopie auf einem bestimmten Server wird als Replikat bezeichnet [CDK12].

Verfügbarkeit. Ein repliziertes System kann Ausfallsicherheit ermöglichen, wenn ein Server oder ein Teil des Systems ausfällt. Solange noch ein Server mit den replizierten Daten existiert, kann der Client seine Anfrage, die von einem ausgefallenen Server nicht mehr bearbeitet wurde, an einen funktionierenden weiterleiten.

Leistungsverbesserung. Replikation kann dazu beitragen die Leistung zu verbessern, da nicht ein einzelner Server die gesamte Last verarbeiten muss, sondern Anfragen auf mehrere Replikate verteilt werden können. Ein weiterer Vorteil ergibt sich in Systemen, die über einen großen geographischen Raum verteilt sind. Wenn Zugriffe von Clients auf Replikate in geographischer Nähe getätigt werden, kann die Verzögerung einer Anfrage durch das Netzwerk verringert werden.

2.3.1 Konsistenz

Eine Herausforderung in replizierten Systemen ist die Einhaltung von Konsistenz, also einer Zusicherung darüber wann Daten, die von verschiedenen Prozessen verändert wurden, auf den Replikaten sichtbar werden.

Es gibt verschiedene Konsistenzmodelle, die verschieden starke Zusicherungen garantieren [FR10]. Die Stärkste wird als *strenge Konsistenz* bezeichnet und garantiert, dass jede Leseoperation den Wert zurück gibt, der zuletzt geschrieben wurde. Dieses ideale Konsistenzmodell kann in verteilten Systemen jedoch nicht eingehalten werden, da es eine globale Zeit voraussetzt [TvS03].

Eine Abschwächung der strengen Konsistenz, die umgesetzt werden kann, ist die *sequentielle Konsistenz* nach Lamport. Diese fordert, dass "das Ergebnis jeder Ausführung dasselbe [ist], als wären die (Lese- und Schreib-) Operationen von allen Prozessen auf dem Datenspeicher in einer sequentiellen Reihenfolge ausgeführt worden, und die Operationen jedes einzelnen

Prozesses erscheinen in dieser Abfolge in der von dem Programm vorgegebenen Reihenfolge" ([Lam79], zitiert nach [TvS03]).

Es existiert also statt einer totalen, eine sequentielle Ordnung. Um erstere zu erhalten muss *Linearisierbarkeit* eingehalten werden. Dazu werden Operationen mit einem Zeitstempel einer synchronisierten Uhr versehen und gefordert, dass eine Operation mit einem kleineren vor einer Operation mit einem größeren Zeitstempel ausgeführt wird.

2.3.2 Replikationsverfahren in XtreemFS

In XtreemFS sind zwei Replikationsverfahren implementiert: eine vereinfachte Variante für unveränderliche Dateien (Read-Only) und eine für veränderliche (Read-Write). Replikationsparameter, wie der Replikationsfaktor, -modus oder auch verwendete OSDs, können sowohl für ein Volume, als auch für einzelne Dateien festgelegt werden.

Die *Read-Only* Replikation setzt eine "write-once" Semantik um und erlaubt es durch das Hinzufügen weiterer Replikate die Zugriffszeiten zu verringern und den nutzbaren Durchsatz zu erhöhen. Im Rahmen dieser Arbeit wird auf die Read-Only Replikation nicht weiter eingegangen, da die diskutierten Probleme hauptsächlich bei der Read-Write Replikation auftreten und eine Lösung mit geringem Aufwand auf die Read-Only Replikation übertragen werden kann.

Die *Read-Write* Replikation erlaubt vollen Dateizugriff und ist für Applikationen transparent. Die Replikation erhöht die Verfügbarkeit und Sicherheit der Daten und ermöglicht auch bei Teilausfällen des Systems Zugriff auf die Daten.

Um die Ausfallsicherheit zu erhöhen und die Netzwerklast zu reduzieren wird in XtreemFS ein mehrheitsbasiertes Protokoll eingesetzt, bei welchem Lese- und Schreiboperationen immer an eine Mehrheit der Replikate gestellt wird (*WqRq: Write Quorum, Read Quorum*). Damit wird sichergestellt, dass es eine Überschneidung von mindestens einem Replikat gibt, das sowohl bei der letzten Änderung, also auch beim Lesen berücksichtigt wurde (siehe Abbildung 3). Das zugrunde liegende Konzept wurde erstmals 1979 als "Quorum Intersection Property" von Thomas diskutiert [Tho79]. Das System funktioniert korrekt, solange mindestens eine Mehrheit der Replikate verfügbar

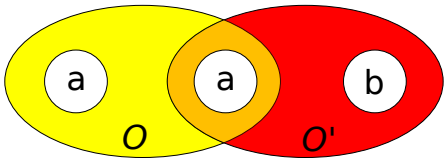

Abbildung 3: Quorum Intersection Property am Beispiel einer Schreiboperation O bei welcher der Wert "b" mit "a" ersetzt wird und einer darauf folgenden Leseoperation O' (Abb. nach [Kol12]).

ist. Die Invariante, die in einem System mit N Replikaten für die Anzahl an Replikaten, die gelesen (R) und geschrieben (W) werden müssen, gilt ist $W + R > N$. In XtreemFS wird diese Invariante für die WqRq Replikation mit $R = W = \left\lceil \frac{N+1}{2} \right\rceil$ eingehalten.

In mehrheitsbasierten Systemen muss sichergestellt werden, dass in jeder Mehrheit der zuletzt geschriebene Zustand identifiziert werden kann. Werden auf einer Mehrheit verschiedene Werte festgestellt, muss entscheidbar sein welcher davon aktuell ist. Am Beispiel von Abbildung 3 muss die Leseoperation o' erkennen, dass "a" der aktuelle Wert ist und diesen zurückgeben. Dies kann erreicht werden, indem den Daten eine monoton steigende Versionsnummer hinzugefügt wird.

2.3.3 Mehrheitsbasierte Primary/Backup Replikation

XtreemFS verfolgt einen Primary/Backup Ansatz auf Dateiebene (vgl. zu diesem Ansatz [BMST93]). Dabei wird ein Replikat als Primary bestimmt, welches alle Anfragen für eine Datei entgegen nimmt und Aktualisierungen an die Backups sendet. Der Primary agiert als Sequenzer und serialisiert alle Operationen. Des Weiteren weist er bei Veränderungen den Daten eine stetig aufsteigende Versionsnummer zu, die es erlaubt unter mehreren Werten den aktuellen zu bestimmen.

Die Serialisierung der Operationen erlaubt Linearsierbarkeit und ist notwendig um eine POSIX kompatible Schnittstelle zu realisieren. Der POSIX Standard fordert, dass "Schreibzugriffe [...] in Bezug zu anderen Lese- oder Schreibzugriffen [serialisiert werden können]. Wenn ein read() auf einer Datei (auf irgendeine Weise) nachweislich nach einem write() der Daten auftritt,

muss es den write() widerspiegeln, selbst wenn die Zugriffe von verschiedenen Prozessen durchgeführt werden. Eine ähnliche Forderung gilt für mehrfache Schreibzugriffe auf die gleiche Dateiposition. Dies ist erforderlich um zu garantieren, dass Daten von write() Zugriffen, für nachfolgende read() Zugriffe sichtbar sind. Diese Anforderung ist insbesondere wichtig für verteilte Dateisysteme, in welchen Cachingmechanismen diese Semantiken verletzen"[1] [IEE08].

Da zu jeder Zeit nur ein Primary existieren kann, werden Failovers mit Hilfe von Leases realisiert [Kol12]. Ein Lease ist die koordinierte Zusicherung an ein Replikat für eine bestimmte Zeit als Primary zu agieren. Wenn der Primary ausfällt und das Lease ausläuft, kann ein anderes Replikat ein Lease erhalten und die Rolle des Primarys übernehmen. Die Gültigkeitsdauer des Lease bestimmt somit die maximale Zeit, die das System nicht verfügbar ist, wenn der Primary ausfällt.

Optimierung von Lesezugriffen. Statt bei jeder Leseoperation eine Mehrheit der Replikate anzufragen, findet während der Wahl des Primarys ein *ReplicaReset* statt, bei welchem alle Daten auf dem Primary aktualisiert werden. Nachfolgende Lesezugriffe können somit vom Primary lokal beantwortet werden.

Beim ReplicaReset wird der Zustand der Daten einer Mehrheit der Replikate erfragt und daraus der aktuelle Zustand und die Speicherorte der Daten, der *AuthoritativeState*, bestimmt. Auf Basis des AuthoritativeState werden nicht vorhandene Daten von den entsprechenden Replikaten nachgeladen.

Die Dateiinhalte werden in XtreemFS in Chunks gleicher Größe aufgeteilt und auf den OSDs gespeichert. Die Bestimmung der Mehrheiten und die Replikation der Daten findet auf der Ebene der Chunks statt. Im Folgenden wird jedoch zur Vereinfachung angenommen, dass für jede Datei nur ein einzelner Chunk existiert.

[1] Eigene Übersetzung. Im Original: "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."

Replizierte Dateizugriffe. Die Replikation ist für den Client transparent und wird allein unter den OSDs auf Basis der vom Client übergebenen Replikatliste koordiniert. Sie findet in mehreren Phasen statt:

1. Über den dezentralen Flease Algorithmus [KHSH11] wird ein Primary bestimmt, dem für eine begrenzte Zeit ein Lease übergeben wird. Flease basiert auf Mehrheitsentscheidungen und ist fehlertolerant. Stellt ein Client eine Anfrage an ein Backup, während ein gültiger Primary existiert, so wird er an diesen weitergeleitet.

2. Wurde ein neuer Primary bestimmt, führt dieser ein ReplicaReset durch um sicherzustellen, dass die lokalen Daten aktuell sind.

3. Der Primary beantwortet Leseanfragen aus dem lokalen Speicher und verteilt Schreiboperationen an die Backupreplikate. Wenn eine Mehrheit der Replikate die Schreiboperation durchgeführt hat, bestätigt der Primary dem Client die Schreiboperation.

3 Algorithmus zum konsistenten Hinzufügen und Entfernen von Replikaten

Wie bereits erwähnt wurde, erhalten Clients für Dateioperationen vom MRC eine Replikatliste (siehe Abbildung 2). Alle Operationen der Replikate zur Herstellung der Mehrheiten oder zur Bestimmung eines Primarys werden auf Basis der vom Client übergebenen Liste ausgeführt. Von der Replikatliste existieren also an vielen Stellen im System Kopien, was kein Problem ist solange sie unveränderlich ist. In einem verteilten System muss es jedoch möglich sein ausgefallene Replikate zu entfernen oder die Anzahl der Replikate zur Lastanpassung zu verändern. Änderungen an der Replikatliste müssen konsistent auf allen Kopien stattfinden, da Operationen auf der Basis veralteter oder inkonsistenter Listen zu inkonsistenten Daten führen können.

Im Folgenden werden diese Fehlerfälle anhand von Beispielen diskutiert und ein Algorithmus entwickelt, der das konsistente Hinzufügen und Entfernen von Replikaten in XtreemFS garantiert. Der Algorithmus wird, ausgehend vom aktuell in XtreemFS Version 1.4 implementierten Stand, in zwei Iterationen entwickelt und in Pseudocode dargestellt. Im Rahmen dieser Arbeit wurde der vorgestellte Algorithmus auch für die Java basierten Server in XtreemFS umgesetzt.

3.1 Zentralisierte Verwaltung von Replikatlisten

Vom Group zum Set Membership Problem

Das grundlegende Problem ist die Verwaltung einer Gruppe bzw. einer Menge von Replikaten, die miteinander kommunizieren bzw. die gemeinsam als Gruppe adressiert werden können. Dieses Problem wurde schon 1987 unter dem Begriff Group Membership von Birman und Joseph beschrieben [BJ87b]. In dieser und in nachfolgenden Forschungsarbeiten wurde einerseits der Frage nachgegangen wie fehlerhafte Prozesse erkannt werden können, und andererseits wie sich, die Prozesse, die Teil einer Gruppe sind, auf Veränderungen an der Gruppe einigen können.

Wie Schiper und Toueg bemerken sind dies jedoch grundsätzlich verschiedene Probleme, deren gemeinsame Lösung zu komplizierten und fehleranfälligen

Algorithmen führt [ST06]. Unter dem Begriff des Set Membership befassen sie sich daher mit der allgemeineren Frage, wie sich eine Menge von Prozessen über den Inhalt einer Menge beliebiger Elemente verständigen und an dieser Veränderungen vornehmen kann. Unter dieser Perspektive stellt Group Membership einen Spezialfall dar, bei welchem die verwaltenden Prozesse zugleich Elemente der zu verwaltenden Menge sind.

In ihrer Lösung stellen sie einen Set Membership Service vor, der für die Verwaltung der Menge zuständig ist und Operationen zum Hinzufügen oder Entfernen von Elementen sequentiell verarbeitet und die veränderte Menge letztlich an alle beteiligten Prozesse verteilt. Zur Vereinfachung des Algorithmus wird angenommen, dass dieser Service keine Fehler aufweist und nicht abstürzen kann. Obwohl dies in der Praxis nicht möglich ist, kann der Service unter Verwendung des State-Machine Ansatzes [Sch90] in einen replizierten und fehlertoleranten Service umgewandelt werden.

MRC als fehlertoleranter Set Membership Service

Der beschriebene Ansatz findet sich auch zur Verwaltung von Replikatlisten in der letzten stabilen XtreemFS Version 1.4 wieder. Der MRC agiert als Set Membership Service und stellt eine Schnittstelle zur Koordination des Hinzufügens und Entfernens von Replikaten bereit. Die Replikatlisten werden wie erweiterte Metadaten von Dateien betrachtet und in der fehlertoleranten, replizierten Datenbank BabuDB gespeichert.

Die Erkennung fehlerhafter Prozesse bzw. Replikate wird in XtreemFS nicht vom MRC übernommen. Eine dafür zuständige Komponente kann jedoch über die Schnittstelle eine Veränderung der Replikatliste veranlassen.

Während eine Veränderung der Replikatliste stattfindet, dürfen keine weiteren Anfragen ausgeführt werden. Um dies zu erreichen, wird die angefragte Operation zum Hinzufügen oder Entfernen von Replikaten in den Metadaten gespeichert und damit die Replikatliste gesperrt. Nachfolgende Veränderungen werden erst ausgeführt, wenn die neue Replikatliste erfolgreich gespeichert wurde.

Die Funktionalität zur Verwaltung der Replikatlisten beim MRC wird in Algorithmus 1 als an Python angelehnter Pseudocode dargestellt.

Algorithmus 1 Verwaltung von Replikatlisten / MRC.

```
1  def changeReplicaSet(operation={ADD, REMOVE}, replicas):
2      global currentReplicaSet
3
4      lockReplicaSetChange(operation, replicas)
5
6      if operation == ADD:
7          newReplicaSet = currentReplicaSet ∪ replicas
8      elif operation == REMOVE:
9          newReplicaSet = currentReplicaSet \ replicas
10
11     storeReplicaSet(newReplicaSet)
12     unlockReplicaSetChange()
```

3.2 View-Synchronous Communication

Die OSDs speichern keine Informationen über andere Replikate oder die Replikatlisten. Die Koordination findet allein über die vom Client übergebene Replikatliste statt, die dieser beim Öffnen von Dateien vom MRC erhält und für mehrere Operationen auf den Replikaten verwenden kann. Das kann jedoch dazu führen, dass Clients eine Operation auf Basis einer Replikatliste ausführen, die nicht mehr aktuell ist. Solche Zugriffe, auf Basis einer veralteten Replikatliste, können die Konsistenz der Daten verletzen.

3.2.1 Inkonsistenzen bei Zugriffen mit veralteten Replikatlisten

Inkonsistenten können sowohl beim Lesen, als auch beim Schreiben auftreten. Im Folgenden werden entsprechende Fehlerfälle erläutert.

Lesen ungültiger Mehrheiten auf Basis veralteter Replikatlisten

Werden Replikate entfernt, erhalten sie von nachfolgenden Schreiboperationen keine Aktualisierungen mehr. Im Beispiel von Abbildung 4 werden aus der Replikatliste $RL = \{A, B, C, D, E\}$ die Replikate $\{A, B, C\}$ entfernt und eine neue Liste $RL' = \{D, E\}$ installiert. Wird nun auf Basis von RL' auf $\{D, E\}$ der neue Wert “2” geschrieben ist nicht garantiert, dass ein nachfolgender Lesezugriff auf Basis von RL diesen zurückgibt. Die entfernten Replikate $\{A, B, C\}$ bilden zwar eine gültige Mehrheit in RL, haben aber keine Kenntnis von der letzten Aktualisierung.

Analog können Fehler auftreten, wenn neu hinzugefügte Replikate selbst eine Mehrheit bilden. Beim Zugriff mit einer veralteten Replikatliste kann nicht garantiert werden, dass es eine Überschneidung gibt, welche Änderungen enthält, die auf der Basis der neuen Replikatliste auf einer Mehrheit vorgenommen wurde.

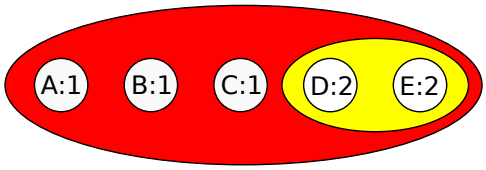

Abbildung 4: Lesen entfernter Knoten auf Basis einer veralteten Replikatliste.

Schreiben auf einer veralteten Mehrheit

Analog zum Lesen, kann auch das Schreiben auf einer veralteten Mehrheit zu Inkonsistenzen führen. Im Beispiel von Abbildung 5 wird die Replikatliste $RL = \{A, B\}$ um $\{C, D, E\}$ zu RL' erweitert. Bei einer darauf folgenden Schreiboperationen auf Basis von RL werden nur die Replikate A und B aktualisiert. Nachfolgende Leseoperationen auf Basis von RL' geben jedoch nur den aktualisierten Wert "3" zurück wenn A oder B in die Mehrheitsbildung eingeschlossen sind. Dies kann jedoch nicht garantiert werden, da $\{C, D, E\}$ in RL' auch eine gültige Mehrheit bilden.

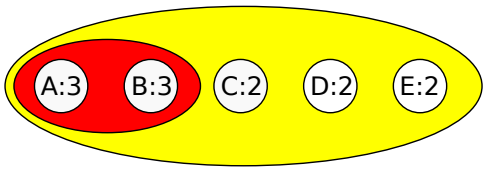

Abbildung 5: Unvollständiges Schreiben auf Basis einer veralteten Replikatliste.

3.2.2 Verwaltung und Validierung von Replikatlisten mit Hilfe von Views

Das Problem der inkonsistenten Daten bei Zugriffen auf Basis veralteter Replikatlisten kann gelöst werden, wenn garantiert wird, dass solche Zugriffe als ungültig erkannt und nicht ausgeführt werden. Das Problem ist im Rahmen

des Group bzw. Set Memberships als View-Synchronous Communication bekannt [CKV01, BJ87a].

Die grundlegende Idee dieses Ansatzes ist die Erweiterung der Replikatliste RL um eine monoton steigende Versionsnummer v. Das Tupel (RL, v) wird als View bezeichnet und den Clients statt der einfachen Replikatliste übergeben. Bei Änderungen der Replikatliste wird die Versionsnummer erhöht und der neue View an die Replikate verteilt. Diese speichern den View, genauer die Versionsnummer[2], persistent ab und überprüfen für Operationen, ob der übergebene View mit dem gespeicherten übereinstimmt.

Die Verteilung der Views an die Replikate läuft in zwei Phasen und ähnelt einem two-phase commit (siehe dazu [TvS03]). In der ersten Phase wird eine Invalidierungsaufforderung an die Replikate gesendet, die zu deren Sperrung führt und die Bearbeitung folgender Operation oder die Bestimmung eines Primarys verhindert. In der zweiten Phase wird der neue View installiert und die Replikate wieder entsperrt. Dieser Algorithmus garantiert, dass die Veränderung eines Views aus globaler Sicht atomar stattfindet und Operationen immer im aktuellen View ausgeführt werden.

Invalidierung der Replikate. In quorenbasierten Systemen ist es ausreichend die Mehrheit der Replikate zu invalidieren um das System zu sperren. Der Einsatz der Primary/Backup Replikation in XtreemFS benötigt jedoch besondere Berücksichtigung. Die Bestimmung des Primarys basiert auf einem vom Client übergebenen View und wird über den Flease Algorithmus realisiert. Während einer Änderung des Views muss daher sichergestellt werden, dass ein zuvor bestimmter Primary seine Rolle aufgibt. Zu diesem Zweck gibt er während der Invalidierung sein Lease zurück und signalisiert dies dem Koordinator. Wenn der Primary nicht auf die Invalidierungsaufforderung reagiert, muss der Koordinator sicherstellen, dass kein Primary mehr existiert und warten bis ein möglicherweise noch existierendes Lease abgelaufen ist.

Installation neuer Views. Wurde garantiert, dass die Mehrheit der Replikate invalidiert ist und kein Primary mehr existiert, installiert der Koordinator den neuen View und liefert diesen auf Anfragen von Clients zurück. Da

[2] Die XtreemFS Architektur erlaubt diese Optimierung, da die vollständige Replikatliste bei jeder Anfrage vom Client übergeben wird.

auf den Replikaten implizit während der Validierung neue Views installiert werden, kann auf die explizite Installation durch den Koordinator verzichtet werden.

Validierung von Views. Stellt ein Replikat bei der Überprüfung einer Anfrage fest, dass diese auf Basis eines neueren Views gestellt wurde, installiert es den neuen View und führt die Operation aus. Wenn die Anfrage auf Basis eines älteren Views gestellt wurde oder der View übereinstimmt, das Replikat jedoch invalidiert ist, wird die Anfrage abgebrochen und ein Fehler zurückgegeben, der den Client zur Aktualisierung des Views auffordert.

Die notwendigen Veränderungen des vorgestellten Algorithmus 1 zur Verwaltung von Replikatlisten sind in Algorithmus 2 für den Koordinator und in Algorithmus 3 für die Replikate dargestellt. Im Rahmen dieser Arbeit wurden die Java basierten XtreemFS Server ebenfalls in einer ersten Iteration entsprechend erweitert.

Algorithmus 2 Verwaltung von Replikatlisten erweitert um Views / MRC.

```
1  def changeReplicaSet(operation={ADD, REMOVE}, replicas):
2      global currentReplicaSet, currentVersion
3
4      lockReplicaSetChange(operation, replicas)
5
6      invalidateReplicas()
7
8      if operation == ADD:
9          newReplicaSet = currentReplicaSet ∪ replicas
10     elif operation == REMOVE:
11         newReplicaSet = currentReplicaSet \ replicas
12
13     storeView(newReplicaSet, currentVersion + 1)
14     unlockReplicaSetChange()
15
16
17 def invalidateReplicas():
18     global currentReplicaSet, currentView
19     primaryResponded = False
20
21     for replica in currentReplicaSet:
22         send(replica, INVALIDATE, currentView)
23
24     while responses < ⌈(|currentReplicaSet|+1)/2⌉:
25         isPrimary = receive(INVALIDATED)
26         if isPrimary:
27             primaryResponded = True
28
29         responses = responses + 1
30
31     if not primaryResponded:
32         waitUntilLeaseExpires()
```

Algorithmus 3 Verwaltung von Replikatlisten erweitert um Views / OSD.

```
1  def invalidateOperation(request):
2      global localVersion, invalidated
3      (replicas, requestVersion) = request.getView()
4
5      if localVersion <= requestVersion:
6          storeLocalView(requestVersion, invalidated=True)
7
8          if localReplicaIsPrimary():
9              giveUpPrimaryStateAndReturnLease()
10             respond(request, INVALIDATED, True)
11         else:
12             respond(request, INVALIDATED, False)
13
14
15 def validateView(request):
16     global localVersion, invalidated
17     (replicas, requestVersion) = request.getView()
18
19     if localVersion == requestVersion and not invalidated:
20         deliver(request)
21
22     elif localVersion < requestVersion:
23         storeLocalView(requestVersion, invalidated=False)
24         deliver(request)
25
26     else:
27         abort(request, "Invalid View")
```

Erhalten des Views beim Entfernen von Replikaten

Im Rahmen der Entfernung von Replikaten können die gespeicherten Dateiinhalte unmittelbar gelöscht werden. Dies gilt jedoch nicht für die persistent gespeicherten Views, deren Information benötigt wird um Zugriffe auf Basis veralteter Replikatlisten zu verhindern (siehe Abschnitt 3.2.1). Der Invalidierungsstatus und der gespeicherte View V eines entfernten Replikats dürfen erst gelöscht werden, wenn garantiert werden kann, dass keine Anfragen mehr auf Basis von V gestellt werden. Da Anfragen neben dem View auch eine gültige Capability enthalten müssen, kann hierfür deren Gültigkeitsdauer Δ_{Cap} herangezogen werden. Wurde ein Replikat invalidiert und aus einem View V entfernt, muss diese Information mindestens für die Dauer Δ_{Cap} erhalten werden. Eine entsprechende Funktionalität könnte über regelmäßige Aufru-

fe des XtreemFS Wartungswerkzeugs realisiert werden. Aktuell wird dies jedoch noch nicht unterstützt und sowohl der Invalidierungsstatus als auch der View von entfernten Replikaten bleiben für unbegrenzte Zeit gespeichert.

3.3 Konsistenzerhaltung beim Hinzufügen und Entfernen von Replikaten

Mit der Einführung und Überprüfung von Views können Fehler von Operationen, die einer Änderung der Replikatliste folgen, verhindert werden. Allerdings können Inkonsistenzen auch durch das Hinzufügen oder Entfernen von Replikaten selbst auftreten. Die Grundlage mehrheitsbasierter Systeme ist die Einhaltung der Quorum Intersection Property, welche jedoch verletzt werden kann, wenn nicht garantiert wird, dass die Mehrheit der Replikate im neuen View den aktuellen Zustand der Datei hat.

3.3.1 Inkonsistenzen beim Hinzufügen und Entfernen von Replikaten

Verletzung der Quorum Intersection Property durch hinzugefügte Replikate

Werden Replikate neu hinzugefügt, haben sie initial einen leeren Zustand. Das bisher besprochene Protokoll überträgt zwar den aktuellen View, aber keine Daten an die Replikate. Das bedeutet, dass sie weder Inhalte, noch Metainformationen der Dateien kennen. Beim Lesen einer Mehrheit von Replikaten, die alle einen leeren Zustand haben, wird der Client informiert, dass keine Daten vorhanden sind.

Das kann wie im Beispiel von Abbildung 6 zu Inkonsistenzen führen, wenn die Replikatliste $RL = \{A, B\}$ um $\{C, D, E\}$ zu RL' erweitert wird. Die initial leeren Replikate $\{C, D, E\}$ bilden in RL' eine gültige Mehrheit und geben statt des korrekten Werts “1” die Information zurück, dass keine Daten vorhanden sind.

Allgemein tritt der Fehler auf, wenn aufgrund neu hinzugefügter, leerer Replikate die Bildung einer Mehrheit möglich ist, die keine Überschneidung zu einer Mehrheit hat, auf der zuvor geschrieben wurde. In diesem Fall werden dann, wie im Beispiel, keine oder veraltete Daten zurückgegeben.

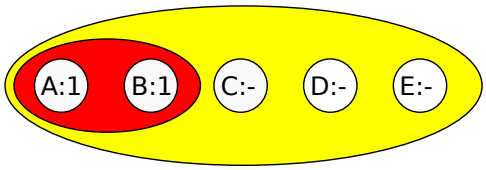

Abbildung 6: Lesen inkonsistenter Daten von leeren Replikaten.

Datenverlust beim Entfernen einer Mehrheit

Ebenso können durch das Entfernen einer Mehrheit der Replikate benötigte Daten gelöscht und damit die Quorum Intersection Property verletzt werden. Im Beispiel von Abbildung 7 bilden die Replikate $\{A, B, C\}$ auf der Basis der Replikatliste $RL = \{A, B, C, D, E\}$ eine gültige Mehrheit, die ausreicht um eine Aktualisierung erfolgreich auszuführen. Die Quorum Intersection Property ist auch erhalten, wenn D und E bspw. aufgrund eines Netzwerkfehlers bei einer Aktualisierung keine Berücksichtigung fanden. Werden nun die Replikate $\{A, B, C\}$ entfernt und eine neue Liste $RL' = \{D, E\}$ installiert, bilden $\{D, E\}$ eine gültige Mehrheit. Die Informationen der letzten Aktualisierung sind diesen jedoch nicht bekannt, weshalb bei einem folgenden Zugriff statt des zuletzt geschriebenen Werts "1" veraltete oder wie im Beispiel keine Daten zurückgegeben werden.

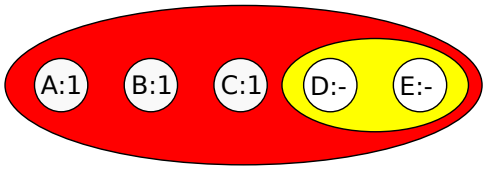

Abbildung 7: Datenverlust beim Entfernen einer Mehrheit.

3.3.2 Konsistenzerhaltung bei der Installation neuer Views

Um Inkonsistenzen bei Änderungen an der Replikatliste zu verhindern, muss sichergestellt werden, dass die Quorum Intersection Property nach der Installation eines neuen Views erfüllt ist. Um dies zu garantieren, müssen ausreichend Replikate aktualisiert werden, während die Mehrheit des Systems invalidiert ist.

In einem mehrheitsbasierten System mit N Replikaten muss die Invariante $W+R > N$ eingehalten werden, wobei W die Anzahl der Replikate bestimmt,

die mindestens beschrieben werden müssen und R die Anzahl der beim Lesen angefragten. Im Fall der WqRq Replikation, für die $R = W = \left\lceil \frac{N+1}{2} \right\rceil$ garantiert wird, ist die Invariante offensichtlich erfüllt.

Werden r neue Replikate hinzugefügt, muss garantiert werden, dass im neuen View mindestens $W'_r = \left\lceil \frac{N+r+1}{2} \right\rceil$ Replikate aktuell sind. Analog gilt, dass nach der Entfernung von s Replikaten noch mindestens $W'_s = \left\lceil \frac{N-s+1}{2} \right\rceil$ aktuell sein müssen. Allgemein gilt, dass vor der Installation eines neuen Views $V' = (RL', v+1)$ garantiert werden muss, dass $W' = \left\lceil \frac{|RL'|+1}{2} \right\rceil$ Replikate aus RL' aktuell sind.

Bestimmung des aktuellen Zustands der Replikate. Wie in Abschnitt 2.3.3 ausgeführt ist, können einzelne Replikate einen ReplicaReset durchführen, bei welchem auf Basis der Statusinformationen einer Mehrheit der Replikate der AuthoritativeState berechnet und fehlende Daten nachgeladen werden. Diese Funktionalität kann auch für die Aktualisierung der Replikate bei der Installation eines neuen Views genutzt werden.

Dazu wird die Bestätigung der Invalidierung um den Status des Replikats erweitert. Sobald eine Mehrheit der Replikate invalidiert wurde, kann eine Datei nicht mehr verändert werden und es ist damit sichergestellt, dass ein AuthoritativeState, der auf Basis der Antworten der Invalidierung berechnet wurde, bis zur Installation des neuen Views korrekt ist.

Herstellung der Quorum Intersection Property im neuen View. Nach der Bestimmung des AuthoritativeState, wird dieser als Teil einer UPDATE Nachricht an alle Replikate aus V' geschickt. Die UPDATE Nachricht muss von der Validierung des Views ausgenommen werden, da zu diesem Zeitpunkt die Mehrheit der Replikate invalidiert ist und keine Anfragen mehr akzeptiert.

Die Replikate starten im Hintergrund die Aktualisierung und senden dem Koordinator eine Bestätigung sobald diese abgeschlossen ist. Der Koordinator wartet auf $W' = \left\lceil \frac{|RL'|+1}{2} \right\rceil$ Bestätigungen bevor der neue View installiert wird.

Diese Vorgehensweise hat den Vorteil, dass irgendwann alle erreichbaren Replikate aktualisiert werden, die Installation des neuen Views jedoch schon

statt findet, wenn erst eine Mehrheit geantwortet hat. Der Algorithmus ist fehlertolerant und funktioniert auch, wenn Replikate ausfallen oder Nachrichten verloren gehen, da die Aktualisierungsaufforderungen wiederholt werden können. Da offensichtlich die Replikate mit der höchsten Responsivität (wenig Last und geringe Netzwerklatenz) die Aktualisierung zuerst ausführen, ist auch die Laufzeit des Koordinators minimal.

Der vollständige Algorithmus zum konsistenten Hinzufügen und Entfernen von Replikaten in XtreemFS ist als Erweiterung des Algorithmus zur View basierten Kommunikation aus Abschnitt 3.2.2 in den Algorithmen 4 für den MRC und im Algorithmus 5 für die Replikate dargestellt. Der vollständige Algorithmus wurde ebenfalls in einer zweiten Iteration für die XtreemFS Server implementiert.

Optimierungen

Der oben beschriebene Algorithmus kann mit einfachen Mitteln optimiert werden. Aus dem AuthoritativeState kann abgeleitet werden, welche Replikate aus RL' schon auf dem aktuellen Stand sind (im Folgenden als $I\hat{N}V$ bezeichnet). Offensichtlich müssen diese nicht mehr aktualisiert werden und es ist ausreichend, nur an die Replikate aus $RL' \setminus I\hat{N}V$ Aktualisierungsaufforderungen zu senden und auf $W' - \left|I\hat{N}V\right|$ Bestätigungen zu warten. Es ist möglich, dass unter dieser Bedingung überhaupt nicht auf Bestätigungen gewartet werden muss und der neue View direkt installiert werden kann.

Um mehr Informationen über schon aktuelle Replikate zu erhalten, kann im Invalidierungsschritt auf mehr als $\left\lceil \frac{N+1}{2} \right\rceil$ Antworten gewartet werden. Das ist insbesondere dann sinnvoll, wenn noch keine Antwort des Primarys vorliegt und der Koordinator erst nach Ablauf der Gültigkeitsdauer des Lease fortfahren kann. In dieser Zeit können weitere Antworten gesammelt und damit die Wahrscheinlichkeit erhöht werden, dass nicht auf die Aktualisierung weiterer Replikate gewartet werden muss.

Algorithmus 4 Verwaltung von Replikatlisten erweitert um Views und Erhaltung der Konsistenz / MRC.

```
1  def changeReplicaSet(operation={ADD, REMOVE}, replicas):
2      global currentReplicaSet, currentVersion
3
4      lockReplicaSetChange(operation, replicas)
5
6      replicaStates = invalidateReplicas()
7      authState = calculateAuthState(replicaStates)
8
9      if operation == ADD:
10         newReplicaSet = currentReplicaSet ∪ replicas
11     elif operation == REMOVE:
12         newReplicaSet = currentReplicaSet \ replicas
13
14     for replica in newReplicaSet:
15         send(replica, UPDATE, authState)
16
17     while responses < ⌈(|newReplicaSet|+1)/2⌉:
18         receive(UPDATED)
19         responses = responses + 1
20
21     storeView(newReplicaSet, currentVersion + 1)
22     unlockReplicaSetChange()
23
24
25 def invalidateReplicas():
26     global currentReplicaSet, currentView
27
28     replicaStates = []
29     primaryResponded = False
30
31     for replica in currentReplicaSet:
32         send(replica, INVALIDATE, currentView)
33
34     while responses < ⌈(|currentReplicaSet|+1)/2⌉:
35         (isPrimary, replicaState) = receive(INVALIDATED)
36         replicaStates.add(replicaState)
37         if isPrimary:
38             primaryResponded = True
39
40         responses = responses + 1
41
42     if not primaryResponded:
43         waitUntilLeaseExpires()
44
45     return replicaStates
```

Algorithmus 5 Verwaltung von Replikatlisten erweitert um Views und Erhaltung der Konsistenz / OSD.

```
1  def invalidateOperation(request):
2      global localVersion, invalidated
3      (replicas, requestVersion) = request.getView()
4
5      if localVersion <= requestVersion:
6          storeLocalView(requestVersion, invalidated=True)
7
8          localReplicaState = fetchLocalReplicaState()
9          if localReplicaIsPrimary():
10             giveUpPrimaryStateAndReturnLease()
11             respond(request, INVALIDATED,
12                         (True, localReplicaState))
13         else:
14             respond(request, INVALIDATED,
15                         (False, localReplicaState))
16
17
18 def validateView(request):
19     global localVersion, invalidated
20     (replicas, requestVersion) = request.getView()
21
22     if localVersion == requestVersion and not invalidated
23             or request.getOperation() == UPDATE:
24         deliver(request)
25
26     elif localVersion < requestVersion:
27         storeLocalView(requestVersion, invalidated=False)
28         deliver(request)
29
30     else:
31         abort(request, "Invalid View")
```

4 Auswertung

4.1 Validierung des Algorithmus mit Integrationstests

Die Funktionalität des implementierten Algorithmus wurde mit JUnit Tests validiert[3]. Dazu wurde auf das vorhandene Testframework und die Java Implementierung von *libxtreemfs* zurückgegriffen. Die Tests orientieren sich an den in Abschnitt 3.2.1 und 3.3.1 diskutierten Fehlerfällen und prüfen sowohl ob Zugriffe mit veralteten Views abgebrochen werden, als auch die Einhaltung der Quorum-Intersection Property. Alle Tests wurden in der vorgelegten Implementierung erfolgreich abgeschlossen.

4.1.1 Zugriff mit veralteten Views

Um den Zugriff mit einem veralteten View zu testen, werden zwei voneinander getrennte Clientinstanzen $C1$ und $C2$ über die *libxtreemfs* initialisiert. Über $C1$ wird eine Datei geöffnet und auf Basis des Views $v = 1$ beschrieben. Ohne die Datei bei $C1$ zu schließen, wird dann von $C2$ ein neues Replikat hinzugefügt und damit der View auf $v = 2$ erhöht sowie die Replikate invalidiert. Ein erneuter Zugriff von $C1$ muss nun von den Replikaten mit einem Fehler beantwortet werden, da der View von $C1$ veraltet ist. Der eingesetzte JUnit Test versichert, dass der letzte Zugriff nicht ausgeführt wurde, sondern einen entsprechenden Fehler zurückgibt.

Der Test ist in vier Variationen für das Hinzufügen und Entfernen sowohl bei der Read-Write, als auch bei der Read-Only Replikation umgesetzt. Er folgt jedoch immer der grundsätzlichen Schematik, die in Abbildung 8 dargestellt ist.

[3] Die aktuelle Version der Testklasse (Revision 4a74e6cabd5e im Repository https://code.google.com/r/jdillmannxtreemfs-personal/) ist online unter http://goo.gl/vqgF3e einsehbar.

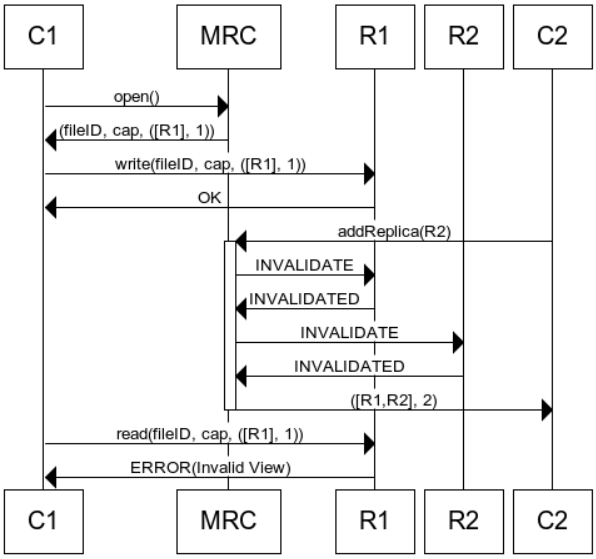

Abbildung 8: Testfall für Zugriffe mit veralteten Views.

4.1.2 Konsistenzerhaltung bei Veränderungen des Views

Um die Konsistenz der Daten beim Hinzufügen und Entfernen von Replikaten zu gewährleisten, wurde in Abschnitt 3.3.2 eine Erweiterung des Algorithmus zur Konsistenzerhaltung vorgestellt. Diese Funktionalität wird ebenfalls durch zwei JUnit Tests überprüft. Um den Fehlerfall sicher zu reproduzieren, wurde eine Funktion implementiert um einzelne OSDs zu unterbrechen und damit den Verlust von Nachrichten zu simulieren.

Der erste Test zum Hinzufügen von Replikaten öffnet und schreibt im ersten Schritt eine Datei mit einem Replikationsfaktor von 2 auf den Replikaten $\{R1, R2\}$. Im zweiten Schritt werden 3 weitere Replikate $\{R3, R4, R5\}$ hinzugefügt, die für sich im neuen View eine Mehrheit bilden. Um zu testen, ob die Aktualisierung erfolgreich war, werden im letzten Schritt die beiden Replikate $\{R1, R2\}$ unterbrochen und auf Basis des neuen Views von der Datei gelesen. Der JUnit Test versichert, dass die dabei gelesenen Daten, den im ersten Schritt geschriebenen entsprechen. Der Ablauf des Tests ist in Abbildung 9 dargestellt.

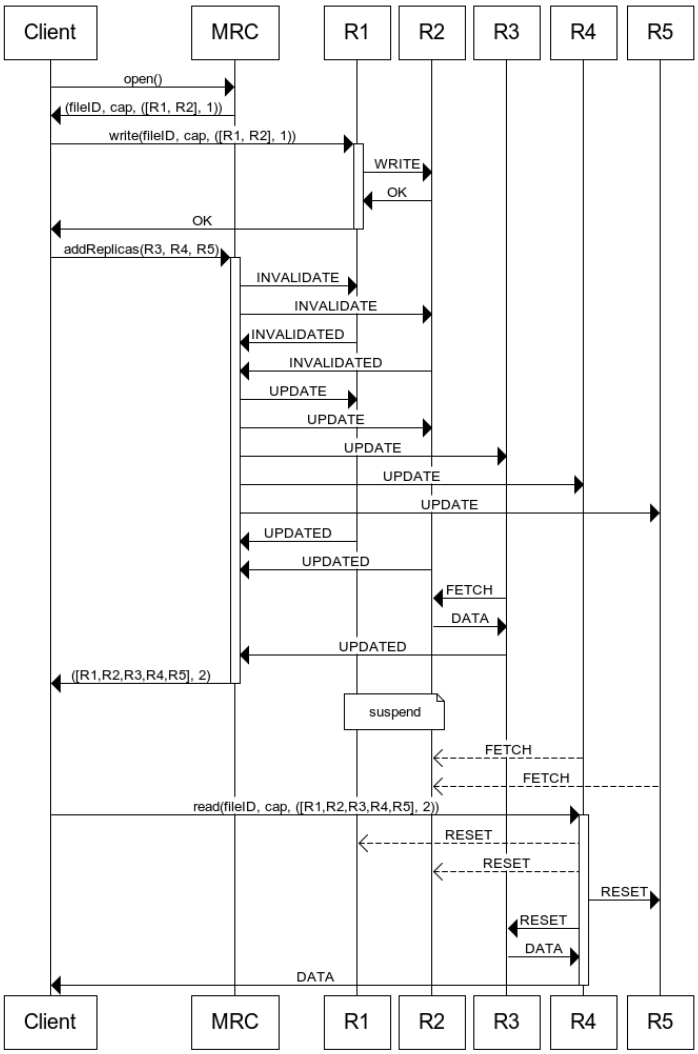

Abbildung 9: Testfall für das Hinzufügen einer neuen Mehrheit.

Für den zweiten Test wird eine Datei mit dem Replikationsfaktor 5 beschrieben. Durch die Unterbrechung der Replikate $\{R1, R2\}$ wird garantiert, dass diese von der Änderung keine Kenntnis haben, während die Replikate $\{R3, R4, R5\}$ eine gültige Mehrheit bilden und aktualisiert werden. Im zweiten Schritt werden die Replikate $\{R1, R2\}$ wieder fortgesetzt und die Replikate $\{R3, R4, R5\}$ entfernt. Um zu validieren, dass die Aktualisierung erfolgreich war, wird im letzten Schritt von den Replikaten $\{R1, R2\}$ im neuen View gelesen und per JUnit versichert, dass die gelesenen Daten korrekt sind. Der Ablauf des Tests ist in Abbildung 10 dargestellt.

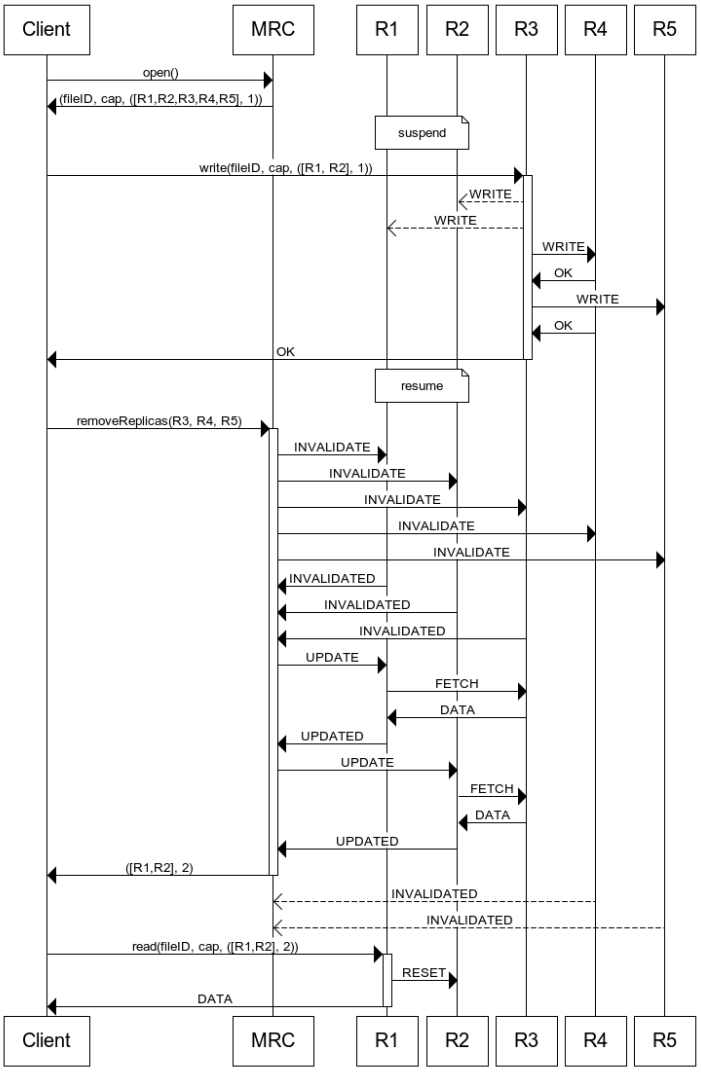

Abbildung 10: Testfall für das Entfernen einer Mehrheit.

4.2 Fehlertoleranz

Der vorgestellte Algorithmus ist fehlertolerant, da für die Invalidierung der Replikate und die Aktualisierung zur Konsistenzerhaltung jeweils nur eine Mehrheit aller Replikate verfügbar sein muss. Der Algorithmus kann somit auch ausgeführt werden, wenn $\left\lfloor \frac{N-1}{2} \right\rfloor$ Replikate aufgrund von Fehlern oder Netzwerkpartitionen nicht verfügbar sind.

Der MRC als Koordinator ist ebenfalls fehlertolerant, da er auf der repli-

zierten Datenbank BabuDB aufbaut. Zu Beginn des Algorithmus wird die Replikatliste gesperrt und die die auszuführende Operation persistent gespeichert. Im Falle eines Failovers kann diese Information ausgewertet und die Installation des neuen Views von einem neuen Koordinator fortgesetzt werden.

Sowohl die Invalidierung, als auch die Aktualisierung der Replikate und die implizite Installation des Views über die Anfragen des Clients sind idempotent und können mehrfach nacheinander ausgeführt werden ohne eine weitere Zustandsveränderung zu bewirken. Aufgrund dieser Eigenschaften können die Operationen im Falle von Netzwerkfehlern oder bei einem Failover des MRC wiederholt werden.

Die untere Grenze von $\left\lceil \frac{N+1}{2} \right\rceil$ verfügbaren Replikaten, die notwendig sind um den Algorithmus auszuführen, deckt sich mit den Anforderungen, die erfüllt sein müssen, damit XtreemFS funktioniert.

4.3 Kommunikationsaufwand

Obwohl für den Algorithmus nur eine Mehrheit der Replikate verfügbar sein muss, werden die Nachrichten zur Invalidierung und Aktualisierung vom Koordinator an alle Replikate gesendet. Die Anzahl der vom Koordinator gesendeten und empfangen Nachrichten ist insgesamt folglich höchstens$(2 \cdot N) + (2 \cdot N')$, wobei vom Koordinator nur $\left\lceil \frac{N+1}{2} \right\rceil + \left\lceil \frac{N'+1}{2} \right\rceil$ Antworten bearbeitet werden müssen.

Die Anzahl der benötigten Nachrichten, die zwischen Replikaten zur Aktualisierung der Daten gesendet werden, kann mit $N' - \left\lceil \frac{N+1}{2} \right\rceil$ abgeschätzt werden, da bekannt ist, dass im alten View mindestens $\left\lceil \frac{N+1}{2} \right\rceil$ Knoten den aktuellen Zustand haben. Insgesamt werden also $(3 \cdot N') + \left(2 \cdot N - \left\lceil \frac{N+1}{2} \right\rceil\right)$ Nachrichten zur Aktualisierung ausgetauscht. Das dabei übertragene Datenvolumen ist abhängig vom tatsächlichen Zustand der Replikate und ist in einem fehlerfreien System, in welchem immer annähernd alle Replikate bei Schreiboperationen aktualisiert werden, gering.

4.4 Ausführungszeit

Die konkrete Zeit, die der Algorithmus zur Ausführung benötigt, kann nicht allgemein bestimmt werden. Sie ist primär davon abhängig wie viele Replikate zusätzlich aktualisiert werden müssen und der Gültigkeitsdauer der Leases Δ_{Lease}. In einem annähernd fehlerfreien System kann davon ausgegangen werden, dass nicht auf die Aktualisierung zusätzlicher Replikate gewartet werden muss und die Ausführungszeit hauptsächlich von der Gültigkeitsdauer der Leases abhängt. In der aktuellen Implementierung des Flease Algorithmus ist es nicht möglich Leases aktiv zurückzugeben, weshalb entweder der Primary oder der Koordinator wartet bis ein aktives Lease abgelaufen ist. Die Wartezeit wird im Mittel $\frac{\Delta_{Lease}}{2}$ betragen und könnte durch eine Erweiterung des Flease Algorithmus vollständig beseitigt werden.

5 Verwandte Arbeiten

Im Rahmen dieser Arbeit konnten nur die Grundlagen der XtreemFS Architektur dargelegt werden. Für ausführlichere Informationen sei daher an dieser Stelle auf bereits veröffentlichte Arbeiten zu XtreemFS [HCK+08, SBR13] und die Internetseite des Projekts http://www.xtreemfs.org/ verwiesen. Das in XtreemFS verwendete Dateireplikationsprotokoll ist im Detail unter [Kol12] beschrieben. Es verwendet Flease, einen Paxos basierten Algorithmus, der eine dezentrale und skalierbare Lease-Koordination in verteilten Systemen ermöglicht [Kol12, KHSH11]. Weitergehende Informationen zu der beim MRC und DIR eingesetzten replizierten Key/Value Datenbank BabuDB finden sich unter [SKHH10].

Die Grundlage des in dieser Arbeit vorgestellten Algorithmus bildet der von Schiper und Toueg vorgestellte Set Membership Algorithmus [ST06]. Dieser trennt die Erkennung fehlerhafter Prozesse von der Verwaltung der Gruppenmitgliedschaften und grenzt sich damit von Group Membership Algorithmen ab. Eine umfassende Untersuchung von Spezifikationen und Algorithmen zu gruppenorientierter Kommunikation findet sich unter [CKV01].

Der in dieser Arbeit verwendete Ansatz, Veränderungen der Replikatliste durch einen separaten Service zu koordinieren, findet sich auch im RAMBO (Reconfigurable Atomic Memory for Basic Objects) Algorithmus wieder [LS02]. RAMBO nutzt einen mehrheitsbasierten Algorithmus für das Lesen und Schreiben der Objekte, sowie einen Rekonfigurationsservice über welchen jederzeit Veränderungen an der Replikatliste vorgenommen werden können. Für die Rekonfiguration wird zunächst über ein an PAXOS [Lam98] angelehntes Konsensprotokoll die nächste Konfiguration bestimmt und installiert. Um die Konsistenz zu erhalten, werden folgende Lese- und Schreiboperationen sowohl auf einer Mehrheit der Replikate der alten, als auch der neuen Konfiguration ausgeführt. Alte Konfigurationen werden dann in einem weiteren Schritt von einem eigenen Garbage-Collector entfernt, sobald die Konsistenz der Daten in der neuen Konfiguration sichergestellt wurde.

Wenn der Garbage-Collector zu langsam ist und mehrere Konfigurationen gleichzeitig aktiv sind, steigt jedoch der Nachrichtenaufwand und die Fehlertoleranz sinkt. Beim Einsatz von RAMBO in Systemen mit unzuverlässiger Kommunikation oder häufigen Rekonfigurationen wurden entsprechende

Probleme festgestellt, die zu Weiterentwicklungen des ursprünglichen Algorithmus führten. Um die Verzögerungen beim Entfernen alter Konfigurationen zu minimieren, werden bei RAMBO II daher Rekonfigurationen parallel ausgeführt [GLS03]. Der auf RAMBO aufbauende RDS (Reconfigurable Distributed Storage) Algorithmus [CGG+06] verfolgt das selbe Ziel, setzt jedoch auf ein effizienteres Protokoll, welches Konsens und Garbage-Collection vereint und keinen separaten Rekonfigurationsservice benötigt.

Aufbauend auf Apaches ZooKeeper, einem Service für die Koordination von Prozessen und verteilten Systemen [HKJR10], wurde vor Kurzem von Shraer et al. ein Algorithmus für die dynamische Rekonfiguration von Primary/Backup Systemen veröffentlicht [SRMJ12]. Ihre Lösung setzt voraus, dass die *primary order* Eigenschaft erfüllt wird [JRS11]. Diese fordert, dass ein Backup, welches eine Änderung von einem Primary erhält, alle Änderungen die zuvor von diesem Primary gesendet wurden ebenfalls erhalten hat (*local primary order*) und neue Primaries alle Änderungen, die zuvor von anderen Primaries gesendet wurden, erhalten bevor sie eigene Änderungen versenden (*global primary order*). Die Rekonfiguration benötigt keinen separaten Service, sondern wird vom Primary koordiniert und kann parallel zu anderen Operationen ausgeführt werden. Der Algorithmus wurde speziell für ZooKeeper entwickelt, kann aber grundsätzlich auf andere Systeme mit primary order Eigenschaften übertragen werden.

Bei den hier vorgestellten Lösungen sind Lese- und Schreiboperationen möglich während eine Veränderung der Replikatliste vorgenommen wird. Sie sind damit offensichtlich effizienter, als der in dieser Arbeit vorgestellte Algorithmus, der im Rahmen der XtreemFS Architektur entwickelt wurde. Eine Implementierung auf Basis der hier vorgestellten Ansätze, hätte tiefgreifende Veränderungen an den Protokollen der Replikationsverfahren zur Folge gehabt. Da XtreemFS ein fortgeschrittenes System ist, dessen Stabilität nicht gefährdet werden sollte, wurde eine Lösung gewählt, die auf der bestehenden Architektur aufbaut. Die Refaktorisierung der Kernfunktionalität bleibt somit zukünftigen Arbeiten vorbehalten.

Primary/Backup Replikation ist ein spezieller Fall der State Machine Replikation [LMZ10], bei welcher über Konsensprotokolle eine Ordnung der Operationen bestimmt und garantiert wird, dass diese auf allen Replikaten in der selben Reihenfolge ausgeführt werden. Eine Schwierigkeit bei der dyna-

mischen Rekonfiguration von State Machines ergibt sich, wenn Operationen parallel ausgwählt werden. In diesem Fall muss die Anzahl der parallelen Operationen auf α beschränkt und die Rekonfiguration um α verzögert werden. Im SMART Algorithmus [LAB+06] wird dieser Ansatz verwendet um ein dynamisches repliziertes System zu implementieren.

6 Fazit und Ausblick

Wie einleitend dargelegt wurde, ist es in verteilten Systemen notwendig Replikate hinzuzufügen oder zu entfernen um dynamisch auf Lastveränderungen zu reagieren oder fehlerhafte Replikate zu ersetzen. Über die Verwaltung der Replikatliste beim MRC war es in XtreemFS grundsätzlich möglich solche Änderungen vorzunehmen (vgl. Abschnitt 3.1). Dabei konnten jedoch Inkonsistenzen auftreten, wie an Hand der in den Abschnitten 3.2.1 und 3.3.1 vorgestellten Fehlerfälle deutlich gemacht wurde.

Im Rahmen dieser Arbeit wurde daher ein Algorithmus entwickelt, der die Konsistenz der Daten bei Veränderungen der Replikatliste garantiert. Die Anlehnung an den Set Membership Ansatz von Schiper und Toueg hat sich dabei als sehr sinnvoll erwiesen, da der MRC als zentraler, aber fehlertoleranter, Koordinator eingesetzt werden kann und grundlegende Veränderungen an der Architektur von XtreemFS nicht notwendig waren. Wie in Abschnitt 4 ausgeführt wurde, ist der implementierte Algorithmus fehlertolerant und löst die zuvor beschriebenen Probleme.

Ein zentraler Aspekt des Algorithmus ist die Aktualisierung veralteter oder neuer Replikate während der Installation eines Views. Diese Aktualisierung folgt derzeit keinem bestimmten Schema und die Daten werden von einem beliebigen aktuellen Replikat angefordert. In einem global verteilten System kann dies jedoch zu starken Verzögerungen oder auch Kosten führen, wenn Daten von einem Replikat in einem weit entfernten Rechenzentrum geladen werden. Um dies zu verhindern, könnte die Aktualisierungsmethode in folgenden Arbeiten um Strategien erweitert werden, die bspw. Replikate aus dem selben Rechenzentrum bevorzugen.

Da der implementierte Algorithmus die Konsistenz der Daten garantiert, könnten zudem Werkzeuge entwickelt werden, die eine dynamische Lastanpassung ermöglichen und automatisiert Replikate hinzufügen oder entfernen. Ebenfalls könnten ausgefallene Replikate automatisch ersetzt werden, sobald weitere Server in einem Rechenzentrum installiert und dem System hinzugefügt werden. Entsprechende Werkzeuge könnten im Rahmen weiterer Arbeiten zu XtreemFS entwickelt werden.

Literatur

[BJ87a] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, November 1987.

[BJ87b] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, 1987.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach, 1993.

[CDK12] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems - concepts and designs (3. ed.)*. International computer science series. Addison-Wesley-Longman, 2012.

[CGG+06] Gregory Chockler, Seth Gilbert, Vincent Gramoli, PeterM. Musial, and AlexA. Shvartsman. Reconfigurable distributed storage for dynamic networks. In JamesH. Anderson, Giuseppe Prencipe, and Roger Wattenhofer, editors, *Principles of Distributed Systems*, volume 3974 of *Lecture Notes in Computer Science*, pages 351–365. Springer Berlin Heidelberg, 2006.

[CKV01] Chockler, Keidar, and Vitenberg. Group communication specifications: A comprehensive study. *CSURV: Computing Surveys*, 33, 2001.

[FMN+05] Michael Factor, Kalman Meth, Dalit Naor, Ohad Rodeh, and Julian Satran. Object storage: The future building block for storage systems. In *2nd International IEEE Symposium on Mass Storage Systems and Technologies*, pages 119–123, 2005.

[FR10] Alan David Fekete and Krithi Ramamritham. Consistency models for replicated data. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication*, volume 5959 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2010.

[GLS03] S. Gilbert, N. Lynch, and A. Shvartsman. Rambo ii: rapidly reconfigurable atomic memory for dynamic networks. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 259–268, 2003.

[HCK+08] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Martí, and Eugenio Cesario. The xtreemfs architecture - a case for object-based file

systems in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[IEE08] Ieee standard for information technology- portable operating system interface (posix) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1–3826, 2008.

[JRS11] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[KHSH11] Bjorn Kolbeck, Mikael Hogqvist, Jan Stender, and Felix Hupfeld. Flease - lease coordination without a lock server. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 978–988, Washington, DC, USA, 2011. IEEE Computer Society.

[Kol12] Björn Kolbeck. *A fault-tolerant and scalable protocol for replication in distributed file systems.* Ph.D. dissertation, 2012.

[LAB+06] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA, 2006. ACM.

[Lam79] L. Lamport. How to make a multiprocessor that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[LMZ10] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1):63–73, March 2010.

[LS02] Nancy A. Lynch and Alexander A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 173–190, London, UK, UK, 2002. Springer-Verlag.

[MGR03] M. Mesnier, G. Ganger, and E. Riedel. Object-based storage. *IEEE Communications Magazine*, 8:84–90, 2003.

[SBR13] Jan Stender, Michael Berlin, and Alexander Reinefeld. Xtreemfs - a file system for the cloud. In D Kyriazis, A Voulodimos, S Gogouvitis, and T Varvarigou, editors, *Data Intensive Storage Services for Cloud Environments*. 2013.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[SKHH10] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. Babudb: Fast and efficient file system metadata storage. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, pages 51–58, Washington, DC, USA, 2010. IEEE Computer Society.

[SRMJ12] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio Junqueira. Dynamic reconfiguration of primary/backup clusters. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[ST06] André Schiper and Sam Toueg. From set membership to group membership: A separation of concerns. *IEEE Trans. Dependable Sec. Comput.*, 3(1):2–12, 2006.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[TvS03] Andrew S. Tanenbaum and Maarten van Steen. *Verteilte Systeme - Grundlagen und Paradigmen*. Pearson Studium, 2003.