Diplomarbeit

Ein Userspace-Dateisystem zur Verwaltung von Grid-Daten

Eingereicht von:

Roland Tuschl

Matrikelnummer: 185507

Betreuer:

Prof. Dr. Alexander Reinefeld

(Zuse-Institut Berlin / Humboldt-Universität zu Berlin)

Prof. Dr. Jens-Peter Redlich

(Humboldt-Universität zu Berlin)

Ort/Datum:

Berlin, den 8. April 2008

Humboldt Universität zu Berlin

Mathematisch-Naturwissenschaftliche Fakultät II

Institut für Informatik

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.
Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Berlin, den 8. April 2008 .

Danksagung

Mein Dank gilt Prof. Dr. Alexander Reinefeld, Leiter des Bereichs Computer-Science-Research am Zuse-Institut Berlin und Prof. Dr. Jens-Peter Redlich vom Lehrstuhl für Systemarchitektur am Institut für Informatik der Humboldt Universität zu Berlin. Insbesondere möchte ich mich bei Florian Schintke vom Zuse-Institut Berlin für die kompetente Beratung und konstruktiven Anregungen bedanken. Weiterhin bedanke ich mich bei meinen Freunden und meinen Eltern für ihre Geduld und Unterstützung.

Inhaltsverzeichnis

1 Einleitung

In der modernen Forschungslandschaft gibt es kaum einen Bereich, in dem auf den Einsatz von Computerberechnungen zur Bearbeitung spezifischer Aufgaben und Probleme verzichtet werden kann. Trotz des rasanten technologischen Fortschritts, der ein sprunghaftes Wachstum von Rechenleistung und Speicherkapazität hervorbringt, stoßen Wissenschaftler bei der Bearbeitung immer komplexerer Problemstellungen oft an die Grenzen der Leistungsfähigkeit verfügbarer Rechensysteme [16].

Die stetig zunehmende weltweite Vernetzung von Computersystemen und die darauf aufbauende Entwicklung des *World Wide Web* [99] hat eine einfach zugängliche Infrastruktur geschaffen, die die Kommunikation und den Informationsaustausch auf globaler Ebene ermöglicht. Diese hat die Arbeitsweise von Wissenschaftlern in den vergangenen zwei Jahrzehnten bereits stark verändert, da das Web nicht nur den Austausch von Erfahrungen und Forschungsergebnissen, sondern auch ein kollaboratives Arbeiten über weite geografische Distanzen hinweg ermöglicht beziehungsweise erleichtert.

Auf dieser Basis entstand die Vision, einen direkten Zugriff auf die über das Web erreichbaren vernetzten Ressourcen zu schaffen, um diese zur Bearbeitung komplexer Probleme nutzen zu können [17]. Anders als bei herkömmlichen Web-Diensten, deren Nutzung spezifische Kenntnisse, wie zum Beispiel deren Ort und die unterstützten Zugriffsmethoden, erfordert [4], soll dies transparent erfolgen. Ausgehend von der Vorstellung, dass ein Benutzer in der Lage sein soll, die verteilten Ressourcen ebenso unkompliziert zu nutzen, wie es möglich ist, Strom aus einer Steckdose zu beziehen, wird dieser Ansatz mit dem Begriff *Grid Computing*[1] bezeichnet [16]. Eine solche gemeinsame Nutzung von Ressourcen erfolgt notwendigerweise im höchsten Maße kontrolliert, wobei sowohl Anbieter als auch Konsumenten genau festlegen, welche Ressourcen unter welchen Umständen in welchem Maße genutzt werden können. Dabei werden globale *virtuelle Organisationen* gebildet, deren Mitglieder solche Richtlinien zur gemeinsamen Nutzung von Ressourcen definieren [19].

Rechen-Jobs, die in solchen Grid-Umgebungen durchgeführt werden, benötigen Eingabedaten, die auf jedem beteiligten Rechenknoten verfügbar sein müssen und erzeugen Ausgabedaten, die dem Benutzer zugänglich gemacht werden müssen. Um mit diesen meist sehr großen Datenmengen [38] umgehen zu können, werden spezielle *Grid-Datenmanagement-Systeme* eingesetzt, welche die gesamte Funktionalität für die Verwaltung und den Zugriff auf Daten-Ressourcen in Grid-Umgebungen bereit stellen [37].

[1] Abgeleitet vom englischen Begriff *Power Grid* (Stromnetz).

Das *ZIB-DMS* ist ein solches System zur Verwaltung von verteilten Daten in heterogenen Umgebungen [51, 58], das am *Zuse-Institut Berlin (ZIB)*[2] entwickelt wird. Es bietet Mechanismen und Methoden zur übersichtlichen und strukturierten Handhabung großer Datenmengen und ermöglicht einen transparenten Zugriff auf die verwalteten Speicher-Ressourcen. Neben der Bereitstellung dieser Funktionalität, ist die verwendete Benutzerschnittstelle ein zentraler Aspekt bei der Entwicklung eines solchen Systems. Zum einen ist eine möglichst intuitive Bedienung des Systems wünschenswert, da die Akzeptanz eines neuen Systems Seitens des Benutzers unter anderem von dem für die Verwendung notwendigen Lernaufwand abhängt. Zum anderen ist die Integration mit bestehenden Systemen und Anwendungen gerade in heterogenen Umgebungen wie global verteilten Grid-Systemen besonders wichtig.

Die wohl bekannteste und am weitesten verbreitete Form der Datenverwaltung ist die des hierarchischen Dateisystems, dessen intuitive Konzepte nicht nur leicht verständlich sind, sondern aufgrund der weiten Verbreitung die mitunter vertrautesten Mechanismen bei der Arbeit mit Computersystemen darstellen. Daher wird im Rahmen dieser Arbeit die Schaffung einer Möglichkeit angestrebt, die Funktionalität des ZIB-DMS über die Dateisystem-Schnittstelle nutzen zu können.

1.1 Ziel der Arbeit

Das Ziel dieser Arbeit ist die Schaffung einer Zugriffs-Komponente für das Grid-Datenmanagement-System *ZIB-DMS*, das dessen transparente Einbindung in den Verzeichnisbaum eines Linux-Systems erlaubt. Dazu wird unter Verwendung des FUSE-Rahmenwerkes ein Userspace-Dateisystem mit Anbindung an das *ZIB-DMS* konzipiert und implementiert. Im Fokus stehen dabei die Abbildung der erweiterten Verwaltungsmechanismen des Systems auf die limitierte Schnittstelle hierarchischer Dateisysteme und die dazu notwendigen Änderungen am *ZIB-DMS*.

1.2 Gliederung

Im Anschluss an diese Einleitung wird in Kapitel 2 als eine grundlegende Form des Datenmanagements das Dateisystem betrachtet. Es werden die verwendeten Konzepte, verschiedene Typen von Dateisystemen und Vorzüge sowie Schwächen dieser Form der Datenverwaltung erörtert. Weiterführend liefert Kapitel 3 einen knappen Überblick über spezielle Anforderungen bei der Verwaltung von Daten in Grid-Systemen und stellt einige bekannte Datenmanagement-Lösungen vor.

[2] http://www.zib.de

Im praktischen Teil dieser Arbeit soll eine Userspace-Dateisystem-Schnittstelle für das *ZIB-DMS* geschaffen werden. Daher wird dieses zunächst in Kapitel 4 eingehend vorgestellt. Es werden die Gestaltungsprinzipien, der Aufbau des Systems sowie die Möglichkeiten zur Organisation von Daten und im weiteren Verlauf die einzelnen Ebenen der verwendeten Software-Architektur und deren Komponenten dargelegt. Darauf folgen in Kapitel 5 Ausführungen zu allgemeinen Aspekten bei der Implementierung von Dateisystemen und dem Konzept von Userspace-Dateisystem, bevor eine detaillierte Beschreibung des FUSE-Rahmenwerks und dessen Verwendung zur Implementierung von Userspace-Dateisystemen gegeben wird.

Kapitel 6 ist das zentrale Kapitel dieser Arbeit. Hier werden zunächst, beginnend bei der Beschreibung der Aussgangssituation und vorbereitender Arbeiten, die vorgenommenen Modifikationen am ZIB-DMS dargelegt. Der Fokus liegt hierbei auf den Änderungen an der *Directory-View*-Komponente. Anschließend wird die Implementation der Userspace-Dateisystem-Varianten dokumentiert.

In Kapitel 7 werden die Ergebnisse der vorgenommenen Leistungsmessungen dargestellt und erörtert. Abschließend liefert Kapitel 8 einen zusammenfassenden Überblick und gibt einen Ausblick auf mögliche zukünftige Weiterentwicklungen.

2 Klassisches Datenmanagement: Dateisystem

Jedes Computer-Programm muss die Möglichkeit haben, Daten in irgendeiner Form entgegenzunehmen und abzulegen [74]. Dabei sollte es möglich sein, auch große Mengen von Daten persistent speichern zu können, so dass sie auch nach Beendigung eines Programms verfügbar bleiben. Ein wichtiger Aspekt ist auch der simultane Zugriff von mehreren Programmen auf den selben Datensatz. Der traditionelle Ansatz zur Erfüllung dieser Anforderungen ist das Ablegen der Daten auf externen Datenträgern. Dabei werden zusammenhängende Informationen in logischen Einheiten, *Dateien*, gespeichert. Deren Verwaltung ist Aufgabe des *Dateisystems*. Es legt fest, wie die abgelegten Informationen strukturiert werden, regelt Art und Umfang des Zugriffs und ist für die persistente Speicherung der Daten zuständig.

Die beiden folgenden Abschnitte geben einen Überblick über die Organisation sowie die wichtigsten Arten von Dateisystemen. Abschnitt 2.3 zeigt die Grenzen hierarchischer Dateisysteme auf. Aspekte zur Implementierung von Dateisystemen werden zu Beginn von Kapitel 5 erörtert.

2.1 Organisation von Dateisystemen

Eine Datei ist die Zusammenfassung von Daten zu einer logischen, mit einem Namen versehenen Einheit. Das Dateisystem bietet einen Abstraktionsmechanismus zur Bildung und Speicherung dieser logischen Einheiten. Die Datei ist das atomare Element eines Dateisystems; Daten können daher nur innerhalb von Dateien gespeichert werden. Die logische Strukturierung der Daten innerhalb einer Datei unterliegt gewöhnlich keiner Einschränkung seitens des Dateisystems. Schreiben und Lesen des Inhalts der Dateien erfolgt transparent für das Dateisystem. Während eine Datei vom Nutzer anhand ihres Namens identifiziert wird, verwendet das Dateisystem intern meist einen eindeutigen numerischen Bezeichner. Zur strukturierten Verwaltung wird jede Datei einer übergeordneten Einheit zugewiesen, dem Verzeichnis. Verzeichnisse werden ebenso wie Dateien mit Namen und Attributen versehen und gespeichert. Viele Dateisysteme implementieren Verzeichnisse in Form von speziellen Dateien. Ein Verzeichniseintrag besteht typischerweise aus dem Namen einer Datei und deren numerischen Bezeichner. Neben den

eigentlichen Daten werden zu jeder Datei Verwaltungsinformationen gespeichert, die *Dateiattribute*. Anzahl und Art der Attribute variiert von einem Dateisystem zum Anderen. Typische Attribute sind neben dem numerischen Bezeichner und des Dateinamens:

- Größe
- Eigentümer
- Zugriffsberechtigungen
- Zeitstempel
- Dateityp

2.1.1 Flache Struktur

Die einfachste Form einer Verzeichnisstruktur ist die eines einzigen globalen Verzeichnisses, dem Wurzelverzeichnis, das alle Dateien enthält. Es können keine weiteren Unterverzeichnisse angelegt werden. Diese Struktur wird heute hauptsächlich in einfachen eingebetteten Systemen vewendet, war aber bei Dateisystemen früher Personal-Computer weit verbreitet [74]. Da diese meist für Single-User-Systeme konzipiert waren und die Datenmengen auch aufgrund der limitierten Kapazität der Datenträger überschaubar waren, war dieses Vorgehen praktikabel. Zudem arbeiteten diese Systeme hauptsächlich mit Wechselmedien, so dass eine logische Strukturierung auch durch den Einsatz mehrerer Speichermedien möglich war. Da eine Datei anhand ihres Namens identifiziert wird, muss jeder Dateiname eindeutig sein. Mit der Verbreitung von Mehrbenutzersystemen ergab sich damit ein Problem: Es kann vorkommen, dass verschiedene Benutzer den selben Dateinamen für verschiedene Dateien verwenden wollen.

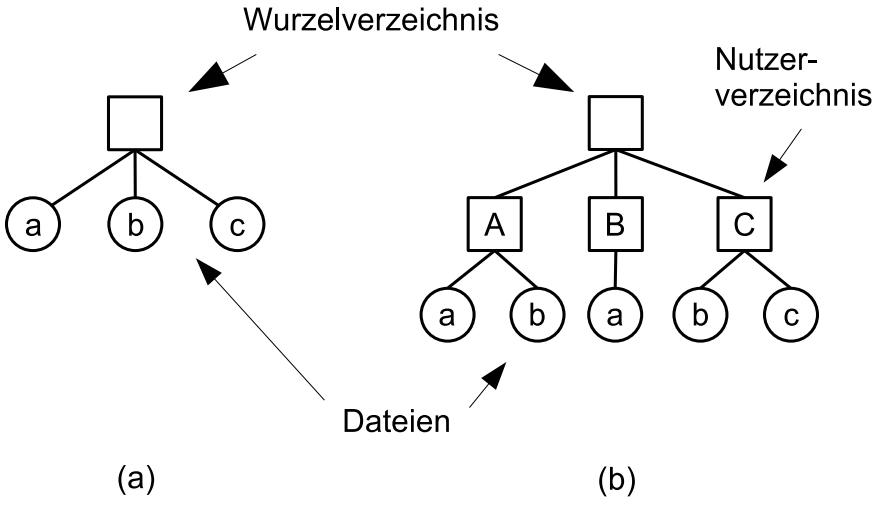

Abbildung 2.1: Beispiele für eine flache Dateisystemstruktur mit einer Ebene (a) bzw. zwei Ebenen (b).

Zur Vermeidung dieses Konflikts wurde eine zweite Verzeichnisebene eingeführt [74]. Jedem Nutzer wird unterhalb des Wurzelverzeichnisses ein eigenes Verzeichnis zur Verfügung gestellt, das alle seine Dateien enthält. Die Identifikation der Datei erfolgt dann durch die Kombination des Verzeichnis- und des Dateinamens. Abbildung 2.1 zeigt Beispiele für Verzeichnisstrukturen mit einer, beziehungsweise zwei Ebenen.

2.1.2 Hierarchische Struktur

Moderne Systeme verwenden Wechselmedien meist nur noch für den Austausch von Daten. Als zentrale Speichermedien dienen Festplatten, deren Kapazität stetig zunimmt. Somit werden immer mehr Dateien in einem einzigen Dateisystem gespeichert. Die Organisation in einer flachen Struktur gestaltet sich mit wachsender Anzahl von Dateien zunehmend unübersichtlich und mühsam. Daher sind fast alle modernen Dateisysteme hierarchisch organisiert [74]. Dateien und Verzeichnisse werden hierbei in einer Baumstruktur organisiert. Abbildung 2.2 zeigt ein Beispiel für solch einen Verzeichnisbaum. Die Wurzel des Baumes ist das Wurzelverzeichnis. Verzeichnisse bilden die Knoten, Dateien die Blätter des Baumes. Jedes Verzeichnis kann beliebig viele Dateien und Unterverzeichnisse enthalten.

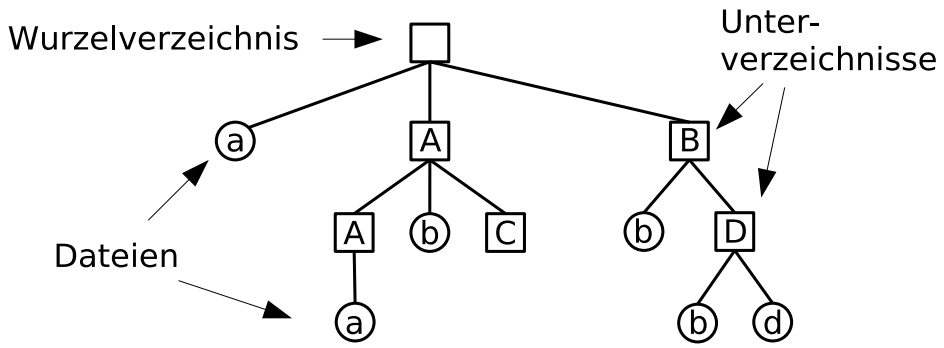

Abbildung 2.2: Beispiel für den Aufbau eines hierarchischen Verzeichnisbaumes.

Die eindeutige Identifikation einer Datei innerhalb der Hierarchie erfolgt durch ihren Pfadnamen. Dieser setzt sich aus den Namen aller Knoten entlang des Pfades vom Wurzelverzeichnis bis zur Datei zusammen. Die einzelnen Komponenten des Pfades werden mit einem system-spezifischen Trennzeichen separiert[1]. Für die Datei d aus Abbildung 2.2 ergibt sich damit beispielsweise der Pfad: */B/D/d*. Die Möglichkeit, unbeschränkt Unterverzeichnisse anlegen zu können, erlaubt es dem Benutzer seine Daten nach einem selbst definierten Schema logisch zu gruppieren. Viele Systeme bieten auch die Möglichkeit, mehrere hierarchische Dateisysteme zu einer einzigen Hierarchie zu kombinieren. Dabei wird einem inneren Knoten des ersten Dateisystems der Wurzelknoten des zweiten Dateisystems zugewiesen. Der so eingebundene zweite Verzeichnisbaum kann

[1] Die gebräuchlichsten Trennzeichen sind /und \

dann über den inneren Knoten des ersten Dateisystems erreicht werden. Elemente des zweiten Baumes werden durch die Kombination der Pfade in beiden Bäumen angesprochen.

Die Baumstruktur bewirkt allerdings, dass jede Datei genau einem Verzeichnis zugeordnet ist. Es ist daher nicht möglich, auf eine Datei an mehreren Stellen im Dateisystem zu verweisen. Verwaltet ein Nutzer beispielsweise alle Quelldateien eines Projektes in einem Verzeichnis `/src` und alle ASCII-Text Dateien in einem anderen Verzeichnis `/ascii`, so ist er nicht in der Lage, die als ASCII-Text abgelegten Quelldateien in `/src` auch über das Verzeichnis `/ascii` anzusprechen. Manche Dateisysteme versuchen dieses Problem durch die Einführung von Verweisen, sogenannter *Hard-Links*, zu lösen. Dabei wird das Dateisystem nicht als Baum, sondern als gerichteter azyklischer Graph aufgefasst [64]. Es können also in verschiedenen Verzeichnissen separate Einträge angelegt werden, die den selben numerischen Bezeichner einer Datei referenzieren. Ein zusätzliches Dateiattribut speichert die Anzahl vorhandener Verzeichniseinträge. Dieser Wert wird beim Anlegen eines Links inkrementiert, beim Entfernen dekrementiert. Eine Datei wird erst dann vollständig gelöscht, wenn der letzte Verzeichniseintrag, der auf sie zeigt, entfernt wird. Um die azyklische Eigenschaft und die Konsistenz der Verzeichniseinträge zu wahren, ist es nicht erlaubt Links auf Verzeichnisse anzulegen. Zudem sind die numerischen Dateibezeichner nur innerhalb des jeweiligen Dateisystems gültig. Daher ist es nicht möglich, dateisystemübergreifende Links zu erzeugen. Diese Einschränkungen können durch die Einführung von *symbolischen Links* überwunden werden.

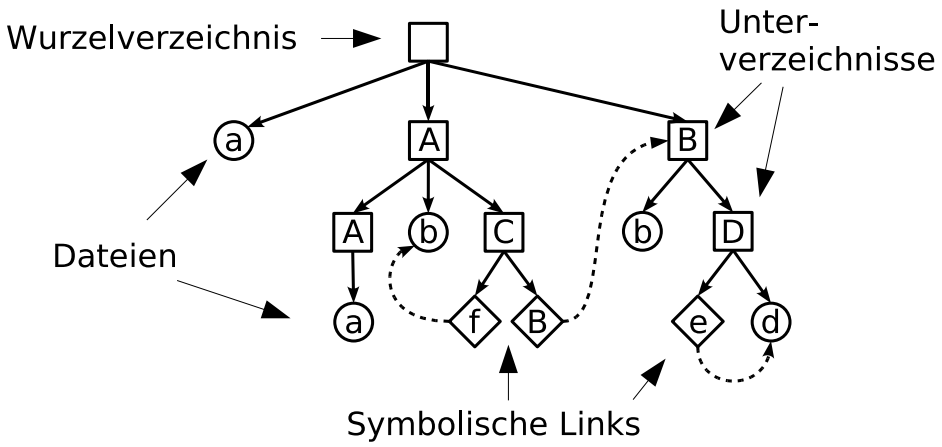

Abbildung 2.3: Beispiel für die Verwendung von symbolischen Links.

Im Gegensatz zu den eben beschriebenen *Hard-Links* wird beim Anlegen eines symbolischen Links nicht nur ein Verzeichniseintrag erstellt. Stattdessen wird eine eigenständige, spezielle Datei vom Typ *Link* erzeugt. Inhalt dieser Datei ist der Pfadname, auf den verwiesen werden soll. Dieser unterliegt keinen Beschränkungen Seitens des Dateisystems. Damit kann auch auf Knoten in der Hierarchie verwiesen werden, die außerhalb des Dateisystems liegen. Außerdem ist es möglich, symbolische Verweise auf Verzeichnisse

zu erzeugen. Abbildung 2.3 zeigt ein Beispiel hierfür. Wie bereits erwähnt, sind Verweise auf Verzeichnisse problematisch, da diese Zyklen im Graph erzeugen können. Durch die Einführung von symbolischen Links verliert der Dateisystem-Graph seine azyklische Eigenschaft. Daher sind Mechanismen und Algorithmen notwendig, um diese Zyklen zu vermeiden beziehungsweise zu erkennen. Ein weiteres Problem entsteht durch die Speicherung des Pfadnamens. Wird das Element, auf das verwiesen wird, gelöscht oder verschoben, wird der Verweis ungültig. Für gewöhnlich wird dieses Verhalten in Kauf genommen und die Behandlung solcher ungültiger Verweise dem Benutzer überlassen. Ein möglicher Lösungsansatz wird in [46] und in Kapitel 6 mit dem *Weaklink*-Konzept von ZIB-DMS beschrieben.

2.2 Arten von Dateisystemen

Die zentrale Bedeutung des Speicherns von Daten und der schnelle technologische Wandel führten dazu, dass im Laufe der Zeit eine Fülle von Forschungs- und Entwicklungsarbeit in das Thema Dateisystem investiert wurde. Das Ergebnis ist eine stetig wachsende Anzahl an Dateisystemen für unterschiedlichste Zwecke. Angefangen von der Unterstützung neuer Datenträgertypen über Optimierungen in der physischen oder logischen Struktur bis hin zu altbekannten Konzepten, die – früher inpraktikabel – heute durch neue Technologien wieder diskussionswürdig sind. Viele verschiedene Motive führen zu immer weiteren neuartigen Dateisystemen. Eine erschöpfende Diskussion aller Konzepte und Aspekte würde den Rahmen an dieser Stelle sprengen. Es soll aber ein kurzer Überblick über die wichtigsten Arten von Dateisystemen, die sich im Laufe der Zeit entwickelt haben, gegeben werden. Die hier vorgenommene Einteilung erfolgt nach dem Kriterium, wie und auf welchem Medium die Dateien gespeichert werden.

2.2.1 Dateisysteme für Datenträger

Ursprünglich wurden Dateisysteme für die Verwaltung lokaler Datenträger wie Disketten oder Festplatten entwickelt. Diese Geräte arbeiten meist *blockorientiert*. Das bedeutet, sie speichern Daten in direkt adressierbaren Blöcken fester Größe. Die Aufgabe des Dateisystems ist damit die Abstraktion von physischen Datenblöcken auf die oben beschriebene logische Hierarchie. Da lokale Datenträger direkt an ein System angeschlossen sind, hängt die Leistungsfähigkeit in erster Linie von der verwendeten Hardware und den I/O-Softwarekomponenten ab. Natürlich übt aber auch die Dateisystem-Software einen nicht geringen Einfluß auf das Leistungsverhalten aus. In diese Kategorie fällt ein Großteil moderner Dateisysteme. Eines der am weitesten verbreiteten ist das *File Allocation Table*-Dateisystem (FAT) [14]. Vielmehr handelt es sich hier um eine ganze Familie von sehr einfach aufgebauten Dateisystemen, die im Laufe der Zeit weiterentwickelt wurden, um Limitierungen der Vorgänger-Versionen zu beheben. Allen gemein ist die Verwen-

dung der im Namen manifestierten Datei-Zuordnungstabelle für die Zuordnung von Datenblöcken zu Dateien. Die prominentesten Vertreter *FAT16* und *FAT32* haben sich zu weit verbreiteten Standard-Dateisystemen entwickelt, die von einem Großteil moderner Betriebssysteme unterstützt werden. Aufgrund ihrer Schlichtheit werden sie bevorzugt auf Wechselmedien und eingebetteten Systemen verwendet. Eine weitere Familie von Dateisystemen gründet sich auf dem aus der Unix-Welt stammenden *Fast File System (FFS)* [35]. Anders als bei FAT-Dateisystemen wird hier die Allokation von Datenblöcken über eine, in reservierten Datenblöcken abgelegte, dynamische Index-Struktur (siehe auch Kapitel 5) realisiert.

2.2.2 Netzwerkdateisysteme

Aus der zunehmenden Vernetzung von Computersystemen erwuchs die Notwendigkeit, Daten einfach austauschen zu können. Anfangs wurden hierzu Anwendungen mit interaktiven Protokollen wie FTP [48] verwendet, die eine explizite Dateiübertragung durch den Nutzer voraussetzten. Dieses Vorgehen stellte sich aber schnell als zu unflexibel heraus. Ein transparenter Zugriff auf Dateien entfernter Rechner ist wünschenswert, so dass Benutzer mit entfernten Dateien ebenso arbeiten können, wie es mit lokalen Dateien möglich ist. Dies ist die Aufgabe von Netzwerkdateisystemen. Anders als bei Dateisystemen für lokale Datenträger, ist für die Leistungsfähigkeit nicht die verwendete Datenträger-Hardware ausschlaggebend. Der Flaschenhals liegt hier in der Stabilität und dem Durchsatz der verwendeten Netzwerkverbindung. Ein weiterer wichtiger Aspekt ist natürlich die Sicherheit, da hier die Gefahr besteht, dass die Verbindung zwischen den Systemen abgehört oder manipuliert wird. Der wohl bekannteste Vertreter ist das von *Sun Microsystems* entwickelte *Network File System* (NFS) [12, 63]. Das NFS erlaubt es einem Server-Knoten, mehrere Teilbäume seiner Verzeichnis-Hierarchie – auch als *Volumes* oder *Mounts* bezeichnet – für den entfernten Zugriff freizugeben. Beliebige NFS-Clients können dann diese freigegebenen Volumes wiederum als Teilbaum in ihren Verzeichnisbaum einbinden. Dabei bietet NFS eine vollständig POSIX-konforme (siehe Kapitel 5) Semantik und kann somit transparent genutzt werden, als ob es sich um ein lokales Dateisystem handelt.

2.2.3 Verteilte Dateisysteme

Netzwerkdateisysteme arbeiten vornehmlich nach einem Client/Server Prinzip, bei dem meherere Clients auf Dateien eines zentralen Servers zugreifen können. In Umgebungen, in denen Dateien auf mehreren Rechnern synchronisiert abgelegt werden, ist dieses Schema nicht praktikabel. Verteilte Dateisysteme liefern einen transparenten Zugriff auf Dateien, die verteilt in einem Rechnerverbund gespeichert werden. Diese Transparenz hat zur Folge, dass die herkömmliche Abbildung von logischen auf physische Dateikennungen um eine Dimension, den zuständigen Rechner-Knoten, erweitert wird. Zudem

kann durch Replikation der Dateien ein logischer Name auch auf mehrere physische Namen abgebildet werden. Dies ist vor allem für die Gewährleistung eines gewissen Maßes an Fehler- und Ausfall-Toleranz sinnvoll. Das Replizieren von Dateien erschwert aber die Bewahrung deren Konsistenz. Viele teilweise experimentelle Dateisysteme versuchen, diese Probleme auf verschiedene Art und Weise zu lösen. Ein sehr bekannter und in der Literatur oft zitierter Vertreter ist das *Andrew File System (AFS)* [26]. Es wurde entwickelt, um die Beschränkungen des zentral organisierten NFS auszuräumen, indem die gespeicherten Daten auf mehrere Server verteilt werden. AFS arbeitet sitzungsbasiert. Beim Öffnen einer Datei wird eine lokale Kopie auf dem Client angelegt, auf der gearbeitet wird. Änderungen an der Datei, die während einer Sitzung von einem anderen Client vorgenommen werden, teilt der zuständige Server dem Client über einen Callback-Mechanismus mit. Wird die Datei geschlossen, endet die Sitzung und die Änderungen an der lokalen Kopie werden auf den zuständigen Server übertragen. Das später entwickelte *Coda*-Dateisystem [10] erweitert dieses Schema um eine Transaktions-Semantik. Damit ist ein Client in der Lage mit den lokalen Kopien weiterzuarbeiten, für den Fall, dass kein Server erreichbar ist. Sobald ein Server wieder verfügbar wird, werden die lokalen Änderungen mit diesem abgeglichen.

2.2.4 Pseudo Dateisysteme

Ein *Pseudo Dateisystem* ist kein Dateisystem im herkömmlichen Sinne. Man versteht darunter den Ansatz, einen Zugriff auf generische Datenquellen über die beschränkten Möglichkeiten der Dateisystemschnittstelle zu schaffen. Motivation ist dabei der geringe Grad an gefordertem Vorwissen und die hohe Akzeptanz Seitens des Benutzers bei der Verwendung dieser Schnittstelle. Ein Pseudo Dateisystem schafft eine Abbildung des Inhalts einer Datenquelle in eine hierarchischen Baumstruktur. Diese ist oft schwierig zu realisieren und zuweilen unvollständig. Manche Pseudo-Dateisysteme gewähren nur lesenden Zugriff, während andere beschränkte oder vollständige Manipulation der Daten erlauben. Ein bekanntes Beispiel hierfür ist das *sysfs* [] des Linux Kernels. Es bietet einen Zugriff auf Betriebssystem-Informationen und Parameter. Diese sind in einer fest vorgegebenen Verzeichnis-Struktur angeordnet und als Dateien abgelegt. Durch Auslesen dieser Dateien können spezifische System-Informationen und -Parameter abgefragt werden. Manche Dateien erlauben durch das Schreiben binärer oder numerischer Werte eine Anpassung der entsprechenden Parameter. Ein völlig anderes Pseudo-Dateisystem liegt mit dem *mysqlfs* [92] vor. Hier werden die Tabellen einer MySQL-Datenbank [91] in eine Verzeichnisstruktur abgebildet. Zu einer Mischform von Netzwerk- und Pseudo-Dateisystem lassen sich Projekte wie *CurlFtpFS* [86] und *sshfs* [73] zählen. Sie erlauben den Zugriff auf entfernte Dateien über die Dateisystem-Schnittstelle unter Verwendung der interaktiven Protokolle FTP [48] und SFTP [22].

2.3 Grenzen hierarchischer Dateisysteme

Das Speichern von Daten in Dateien und deren Ablage in einer hierarchischen Ordnerstruktur hat sich im Laufe der Zeit zur dominierenden Form der Datenverwaltung für Endbenutzer entwickelt. Dies ist vor allem auf die leichte Verständlichkeit der Datei- beziehungsweise Verzeichnis-Abstraktion und deren einfache Bedienung zurückzuführen. Da die Dateisystemdienste einen Basisdienst des Betriebssystems darstellen, werden diese von jeder Anwendung in gleicher Weise genutzt. Viele Betriebssysteme bieten zudem eine einheitliche Oberfläche für die Verwaltung von Dateien. Andere Formen der Datenverwaltung, wie zum Beispiel Datenbanksysteme, setzen mehr Vorwissen und eventuell das Erlernen komplexer Abfragesprachen voraus. Ein Dateisystem dagegen kann mit geringen Vorkenntnissen und ohne lange Einarbeitungszeit benutzt werden.

Allerdings bringt eben diese Simplizität und der hohe Grad der Abstraktion viele Einschränkungen mit sich. Diese wirken sich mit wachsender Datenmenge zunehmend negativ aus. Die Studie *How much Information 2003* [33] hat eine jährliche Zuwachsrate von über 30% an neuen Daten ermittelt. Dateisysteme mit einer Größe von mehreren hundert Gigabyte sind bei einem Personal Computer inzwischen die Regel. Im wissenschaftlichen Umfeld sind Datenmengen im Terra- und Exabytebereich reelle Größen. Hinzu kommt die zunehmende Digitalisierung von Informationen aus allen Lebensbereichen – von Multimedia Anwendungen bis zur Verwaltung persönlicher Informationen. Bei der Verwaltung großer und ständig wachsender Datenmengen in einer starren hierarchischen Struktur stößt man schnell an die Grenzen der Übersichtlichkeit und Benutzbarkeit. Die folgenden fünf Aspekte werden in [46] als die gravierendsten Probleme hierarchischer Dateisysteme identifiziert:

Keine Multiexistenz
: Die hierarchische Struktur bedingt, dass eine Datei immer genau einem Verzeichnis zugeordnet wird. Somit kann beispielsweise eine Musikdatei nicht mehreren Kategorien zugeordnet werden. Die in Abschnitt 2.1.2 vorgestellten Links versuchen dies zwar zu bewerkstelligen, stellen aber aufgrund der genannten Einschränkungen keine optimale Lösung dar [20].

Schlechte Reorganisation
: Dateisysteme speichern Daten transparent. Eine sinnvolle logische Strukturierung liegt in der Verantwortung des Benutzers. Der semantische Kontext einer Datei kann sich im Laufe der Zeit verändern. Da das Dateisystem über keinen Einblick in die Semantik der Daten verfügt, ist eine automatische Reorganisation aber nicht möglich. Diese muss manuell durch den Nutzer stattfinden. Im Falle großer Datenmengen und eventuell komplexer Verzeichnisstrukturen gestaltet sich dies recht aufwändig.

Fehlende Suchmechanismen
: Das Auffinden einer Datei in einem Verzeichnisbaum setzt die Kenntnis des Datei- und Pfadnamens voraus. Fehlt eine dieser Informationen wird es schwierig, die

entsprechende Datei zu lokalisieren. Dateisysteme bieten im Allgemeinen keine Suchmechanismen. In kleineren Datenbeständen kann dieses Problem durch manuelles Durchsuchen überwunden werden. Bei wachsender Datenmenge wird dies zunehmend mühsam und inpraktikabel. Fehlt die Information über Pfad und Name völlig und die Datei soll anhand eines Kontextes oder semantischer Informationen gefunden werden, ist dieses Vorgehen nahezu aussichtslos. Viele Betriebssysteme bieten zwar Suchmechanismen, diese sind aber mit wenigen Ausnahmen meist als gewöhnliche Anwendungsprogramme realisiert und erfahren keine direkte Unterstützung durch das Dateisystem.

Vermischung von Bezeichner und Information

Dateien werden durch den Benutzer innerhalb der Hierarchie anhand ihres Pfades und des Dateinamens identifiziert. Der Dateiname unterliegt dabei für gewöhnlich nur syntaktischen Beschränkungen. Da das Dateisystem im Allgemeinen keine Informationen über den Inhalt einer Datei speichert, wird der Name einer Datei häufig dazu genutzt zusätzliche Informationen zu hinterlegen, um beispielsweise ein späteres Auffinden zu erleichtern. Dies ist zwar ein übliches und probates Mittel der Informationsverwaltung, widerspricht aber dem Konzept des Dateinamens als einem reinen Bezeichner, der ein Element identifizieren, nicht aber qualifizieren soll. Der Dateiname ist hierfür weder vorgesehen noch ausreichend geeignet [44].

Assoziation und Relation

Die Zugehörigkeit von Dateien zu einem Verzeichnis kann als Abbildung einer Relation zwischen den fraglichen Dateien aufgefasst werden. Damit lassen sich allerdings nur sehr einfache Beziehungen zwischen Dateien ausdrücken. Ein gewöhnliches hierarchisches Dateisystem bietet keine Möglichkeit, mehrere unterschiedliche und eventuell qualifizierte Verbindungen zwischen Dateien herzustellen. Assoziationen zwischen Dateien sind nur beschränkt möglich. Komplexe Relationen lassen sich überhaupt nicht ausdrücken.

Trotz der genannten Defizite wird die Verwaltung von Daten in einer hierarchischen Verzeichnisstruktur auch in Zukunft eine wichtige, wenn nicht tragende Rolle spielen, da das intuitive Konzept der Datei-Metapher und der hierarchischen Organisation in einer Ordnerstruktur nicht nur leicht verständlich ist, sondern aufgrund der weiten Verbreitung inzwischen auch einen der vertrautesten Mechanismen bei der Arbeit mit Computersystemen darstellt [28, 20] und sich derartig breit etablierte Standard-Technologien nur sehr schwerfällig wandeln lassen.

3 Datenmanagement in Grid-Systemen

Die Verwaltung von Daten ist eine der zentralen Herausforderungen auf dem Gebiet des Grid-Computing. In aktuellen Grid-Systemen, wie sie in Industrie und Wissenschaft heute verwendet werden, liegt das Datenaufkommen im Petabyte-Bereich [27]. Die Verwaltung solch großer Datenmengen erfordert eine hoch-parallele, verteilte Speicher- und Verwaltungs-Infrastruktur.

Ein Grid kann als ein Verbund von heterogenen Rechen- und Speicher-Ressourcen angesehen werden. Da es sich überlicherweise aus Rechner-Clustern verschiedener Organisationen zusammensetzt, unterliegt es nicht der Kontrolle einer einzelnen, zentralen Instanz. Während bei der traditionellen Datenverwaltung in Dateisystemen von der vollen, zentralen Kontrolle über lokale oder entfernte Speicherressourcen ausgegangen wird, muss in einem Grid-Datenmanagement-System mit weit verteilten, heterogenen Daten- und Speicherressourcen umgegangen werden, deren volle Kontrolle auf verschiedene Organisationen verteilt ist. Für solche Systeme spielt Dezentralisierung eine wichtige Rolle, da ein zentral organisiertes System die Verfügbarkeit von Daten aufgrund des Leistungsspektrums von WAN-Verbindungen und des hohen Risikos von Netzwerk-Ausfällen stark beeinträchtigen würde.

Grid-Computing und im Speziellen Grid-Datenmanagement ist eine sehr umfangreiche und komplexe, aber auch relativ junge Disziplin. Für die Grid-Datenverwaltung existiert auf Anwendungsebene bisher keine einzelne weit verbreitete Standard-Lösung. Dieser Mangel und die Komplexität der Aufgaben, die in Grid-Umgebungen bearbeitet werden, führten bislang dazu, dass viele Grid-Systeme ein inviduell angepasstes Datenmanagement-System (DMS) verwenden. Dies führt zu vielen verschiedenen Datenmanagement-Lösungen, deren Funktionalität sich zwar in weiten Teilen überdeckt, die aber keinen zufriedenstellenden Grad an Kompatibilität untereinander aufweisen. Zudem erschweren es diese Insel-Lösungen Software-Entwicklern, Anwendungen zu entwickeln, die ohne größere Anpassungen in unterschiedlichen Grid-Umgebungen eingesetzt werden können.

Die Entwicklung eines DMS, das in jedem Grid eingesetzt werden kann, ist eine große Herausforderung. Aufgrund der Fülle an aktueller Forschungsarbeiten auf diesem Gebiet und dem Umfang der Thematik kann hier keine tiefgreifende Erörterung des Themas erfolgen. Bevor im folgenden Kapitel das Grid-Datenmanagement-System *ZIB-DMS*

vorgestellt wird, das untersucht, wie solch ein System geschaffen werden kann, soll in diesem Kapitel ein kurzer Einblick gegeben werden, welche Anforderungen an ein derartiges System gestellt werden. Zusätzlich sollen einige der bekannteren Grid-Datenmanagement (Teil-)Lösungen übersichtshalber vorgestellt werden.

3.1 Spezielle Anforderungen an Grid-Datenmanagement-Systeme

AstroGrid-D [79], *C3Grid* [82] und *MediGRID* [90] sind drei etablierte Grid-Projekte aus verschiedenen wissenschaftlichen Bereichen. Aufgrund der unterschiedlichen und speziellen Anforderungen der einzelnen Projekte, wird das Datenmanagement in jedem der Systeme auf eigene Art und Weise realisiert. Durch einen Vergleich dieser drei Grid-Projekte werden in [45] deren gemeinsame Ansprüche an das Datenmanagement erarbeitet und auf dieser Basis grundlegende Anforderungen an ein Grid-DMS abgeleitet, die in den folgenden Abschnitten erörtert werden.

Transparenz

Eines der zentralen Ziele in Grid-Umgebungen ist es, eine möglichst einfache Nutzung der verteilten Ressourcen zu gewährleisten. Im Bereich Datenmanagement konzentriert sich dies im Wesentlichen auf zwei Aspekte: Orts- und Zugriffstransparenz. Ein ortsunabhängiger Zugriff erfordert die Etablierung von Indirektions-Ebenen zur Abstraktion des Namensraumes. Auf der oberen Ebene wird ein stabiler, logischer Namensraum geschaffen, der zur Adressierung von Ressourcen auf Anwendungsebene verwendet wird. Dieser abstrahiert von der im jeweiligen Grid-System tatsächlich verwendeten Adressierung, dem sogenannten physikalischen Namensraum. Veränderungen, die im Laufe der Zeit in dieser unteren Ebene auftreten, werden durch die obere Ebene vor dem Benutzer verborgen. In einem weit verteilten Grid-System werden Daten in teils unterschiedlichster Form auf verschiedenartigen Systemen gespeichert. Der transparente Zugriff auf diese inhärent heterogenen Datenquellen des Grid-Systems bedingt die Schaffung eines einheitlichen Mechanismus für den Zugriff auf Datenressourcen, unabhängig in welcher Form diese vorliegen und gespeichert werden.

Gemeinsamer Zugriff

In einigen Grid-Systemen ist der geteilte Zugriff, also die Möglichkeit für Benutzer auf Daten anderer Benutzer ortsunabhängig zugreifen zu können, ein wichtiger Aspekt. Dies erfordert Möglichkeiten zur Beschreibung und Publizierung der eigenen Daten, sowie Such- und Zugriffsmechanismen für die Daten anderer Benutzer.

Metadaten

Die Mehrheit der Grid-Systeme nutzt Metadaten, um die Daten-Verwaltung zu vereinfachen und effizienter zu gestalten. Dabei liegen die Unterschiede in Format und Struktur der Metadaten, in deren hauptsächlich intendierten Verwendungszweck, sowie in der Ausprägung der unterstützten Operationen auf und mit den Metadaten. Als Metadaten werden hier alle Arten von Zusatzinformationen zu den verwalteten Daten bezeichnet. Dazu zählen:

- Verwaltungs-Informationen, die ähnlich wie die aus Dateisystemen bekannten Attribute Auskunft über grundlegende Informationen zu den Datenobjekte geben, wie beispielsweise dessen Benennung und Adressierung im logischen Namensraum oder die Größe und Typ des Datenobjekts.
- Anwendungsspezifische Informationen, die von bestimmten Anwendungen automatisiert erzeugt werden. Bekannte analoge Beispiele hierfür wären die in JPEG-Bildern oder MP3-Dateien hinterlegten Metadaten.
- Lokalitäts-Informationen, die Angaben über Person, Ort oder Zeit der Erstellung, Veränderung oder Nutzung geben.
- Benutzerdefinierte Informationen, um die Nutzung der Daten durch qualifizierende Informationen oder Annotationen in beliebiger Form zu vereinfachen beziehungsweise zu ermöglichen.

Die Verfügbarkeit solcher Metadaten wird mit wachsendem Datenvolumen zunehmend wichtiger, um eine effiziente und übersichtliche Verwaltung der Datenobjekte gewährleisten zu können und die Verarbeitung der Daten zu unterstützen, teilweise sogar zu ermöglichen. Das Nutzungsspektrum der Metadaten erstreckt sich von der Verwendung zur simplen Identifikation bekannter Datenobjekte durch fest vorgegebene Attribute, bis hin zu komplexen Volltext-Suchanfragen für das Auffinden unbekannter Datenobjekte anhand bestimmter Kriterien. Aufgrund dieser teilweise stark unterschiedlichen Anforderungen ist eine flexible, leicht anzupassende Verwaltung von Metadaten notwendig.

Replikation

Ausfälle der Hardware und eventuell auftretende, vorübergehende Partitionierungen der Netzwerktopologie stellen ein besonderes Problem in solchen global verteilten Systemen dar. Ein Grid-DMS muss auch bei Ausfällen von einzelnen Knoten eine hohe Verfügbarkeit der Daten sowie einen effizienten Zugriff gewährleisten. Dies wird üblicherweise durch *Replikation*, also das redundante Anlegen von Kopien auf verschiedenen Knoten, erreicht. Die Verfügbarkeit solcher Kopien eröffnet einige Möglichkeiten, die den Zugriff auf die Daten verbessern. Beispielsweise kann durch die Schaffung paralleler und partieller Zugriffe auf verschiedene Replikate eines Datenobjekts ein höherer Durchsatz und ein gewisser Grad an Last-Balanzierung erreicht werden. Bei Verfügbarkeit mehrerer

Replikate kann durch geschickte Wahl eines günstigen Replikats ebenfalls die Leistung verbessert werden. Zudem kann beim Ausfall einer Datenquelle eine alternative Speicher-Ressource ermittelt und verwendet werden.

Virtualisierung von Ressourcen

Grundlegende Ziele bei der Verwendung geteilter und verteilter Speicher-Ressourcen sind unter anderem die Erhöhung der Gesamt-Speicherkapazität und die Schaffung einer höheren Dienstgüte und Auslastung der Ressourcen. Die dazu notwendige Virtualisierung der Speicher-Ressourcen wird gewöhnlich ebenso wie die Abstraktion des Namensraumes durch Indirektion erreicht. Abbildungen logischer Dateneinheiten auf verteilte Speicher-Ressourcen können beispielsweise durch Zuordnungs-Kataloge verwaltet und zugänglich gemacht werden. Die Flexibilität und Erweiterbarkeit dieser Abbildung ist ein wichtiger Aspekt für generische Grid-Datenmanagement-Systeme.

Co-Scheduling

Es gibt einen Zusammenhang zwischen der Datenverwaltung und der Planung und Ausführung von Rechen-Jobs innerhalb des Grids. Die Jobs operieren meist auf riesigen Datenmengen, die oft auf den zuständigen Rechenknoten kopiert werden müssen. Somit hat die Platzierung der Jobs im Grid einen direkten Einfluss auf die Menge an Daten, die übertragen werden muss. Daher ist ein Mechanimus wünschenswert, der es dem DMS ermöglicht die Planung und Platzierung der Jobs zu beeinflussen. Beispielsweise könnten Jobs bevorzugt auf Knoten platziert werden, auf denen entweder Replikate der benötigten Daten vorhanden sind oder eine möglichst kurze, direkte Verbindung zu den Knoten besteht, auf denen die fraglichen Daten abgelegt sind. Weiterhin kann das DMS die zeitliche Planung von Jobs durch die Angabe der erwarteten Übertragungsdauer der Daten unterstützen.

3.2 Grid-Datenmanagement-Systeme

Wie bereits erklärt, existiert eine Fülle von verschiedenen Datenmanagement-Lösungen für Grid-Umgebungen. Eine erschöpfende Darlegung der verfügbaren Systeme ist aufgrund des Umfangs hier nicht möglich. Um dennoch einen Einblick zu geben, wird in den folgenden Abschnitten ein knapper Überblick über einige der bekanntesten Datenmanagement-Lösungen gegeben.

Globus

Eines der populärsten Rahmenwerke zur Entwicklung von Grid-Software ist das *Globus Toolkit* [18]. Es stellt auch grundlegende Mechanismen zur Datenverwaltung und -übertragung bereit. Auf den einzelnen Knoten des Systems wird der *GridFTP*-Dienst [1] ausgeführt. Dieser exportiert die Speicherressourcen des Knotens und macht sie über das Netwerk nutzbar. Der *Replica Location Service (RLS)* [53] führt einen Index über diese Speicher-Quellen, etabliert einen logischen Namensraum und bietet Basis-Mechanismen für die Verwaltung von Datenobjekten. Das GridFTP-Rahmenwerk ermöglicht in Verbindung mit dem *Reliable File Transfer (RFT)*-Dienst [2] einen performanten parallelen Datenaustausch zwischen den Knoten. Diese Lösungen lassen sich leicht in die lokalen Verwaltungs- und Zugriffsberechtigungs-Strukturen integrieren, da der physikalische Namensraum nicht verändert wird.

SRB

Der am *San Diego Supercomputer Center (SDSC)* entwickelte *Storage Resource Broker (SRB)* [3] bietet eine umfassende Datenmanagement-Lösung für Grid-Systeme. Anders als bei Globus wird hier nicht versucht, den Namensraum des Dateisystems der Speicher-Knoten zu bewahren. Stattdessen wird ein eigener physikalischer Namensraum aus eindeutigen Datei-Bezeichnern geschaffen, unter dem die Datenobjekte abgelegt werden. SRB stellt eine einheitliche Schnittstelle auf Anwendungsebene bereit, um einen transparenten Zugriff auf heterogene, verteilte Speicher-Ressourcen wie Datei- oder Datenbanksysteme zu bieten. Das System verwendet einen Metadaten-Katalog *(MCAT)* zur Schaffung eines logischen hierarchischen Namensraums, zur Verwaltung von Medataden und zur Bereitstellung eines attributbasierten Zugriffs auf den Datenbestand.

iRODS

Das *integrated Rule-Oriented Data System (iRODS)* [50, 57] ist ein ebenfalls am SDSC entwickeltes regelbasiertes DMS. Es wurde als Nachfolgeprojekt des SRB gestartet, welcher sich in der Praxis als zu unflexibel herausgestellt hatte. iRODS verwaltet Daten-Ressourcen ähnlich wie SRB mittels eines Medataden-Katalogs in einem eigenen logischen Namensraum. Die Funktionalität und das Verhalten des Systems kann jedoch durch benutzerdefinierte Regeln definiert und erweitert werden. Dies schafft die Grundlage für ein an die Bedürfnisse verschiedener Grid-Umgebungen anpassbares DMS.

Grid Datafarm

Die *Grid Datafarm (Gfarm)*-Architektur [75] bietet eine Lösung zur Verwaltung von Grid-Daten in einem globalen verteilten Dateisystem, das mit Funktionen zur Platzierung von Rechen-Jobs und parallelen Zugriffsmechanismen kombiniert wird. Auf jedem Knoten des Systems wird der Gfarm-Dateisystem-Dienst (*gfsd*) ausgeführt, der Fragmente der Dateien des Systems speichert. Es wird ein logischer hierarchischer Namensraum etabliert. Die Abbildung der logischen Namen auf Dateifragmente und Replikate, sowie weitere vom System festgelegte Metadaten, werden in einer separaten Datenbank gespeichert. Durch die Nutzung jedes Knotens als Speicher- und Rechenknoten und der Integration mit einer Job-Scheduling-Komponente wird versucht, Datenlokalität bei Berechnungen auszunutzen, indem die Rechen-Jobs auf die zuständigen Speicherknoten verschoben werden, statt umgekehrt. Zudem werden manuell oder automatisch angelegte Replikate von Dateien genutzt, um Lastbalanzierung zu realisieren und eine höhrere Fehlertoleranz zu gewährleisten. Während in der ersten Version eine *Write Once, Read Many*-Semantik [64] verwendet wurde, wird in der Folgeversion *Gfarm v2* [76] eine vollwertige Dateisystem-Funktionalität angestrebt. Dabei wird für den Zugriff ebenfalls ein FUSE-Dateisystem entwickelt [88].

XtreemFS

XtreemFS [28, 27] ist ein Projekt zur Schaffung eines objektbasierten lose gekoppelten verteilten Dateisystem für den Einsatz in Wide-Area-Netzwerken, wie sie in Grid-Systemen auftreten. Es wird als Teil des XtreemOS-Projektes [100] in einer Kooperation von mehreren Organisationen entwickelt. Zu den Partnern zählt auch das *Zuse-Institut Berlin (ZIB)*. XtreemFS verwaltet Dateien in *Volumes*, von denen jedes eine separate hierarchische Verzeichnisstruktur mit eigenem logischen Namensraum enthält. Der Inhalt einer Datei wird als *Objekt* in sogenannten *Object Storage Devices (OSD)* repliziert gespeichert. Dabei kann ein Replikat als einzelnes Objekt in einem einzigen OSD abgelegt werden oder in Teilobjekte zerlegt auf mehrere OSDs verteilt gespeichert werden. Die Metadaten und Informationen über Replikate der Daten eines Volumes werden ebenfalls repliziert in mehreren Instanzen des *Metadata and Replica Catalogue (MRC)* abgelegt. Ein Verzeichnisdienst gibt Auskunft über alle registrierten MRC-Instanzen eines Volumes. Das System bietet eine POSIX-konforme Semantik und Schnittstelle. Letztere wird in der Zugriffsschicht des Systems durch ein FUSE-Dateisystem realisiert, das die Dateisystemanfragen in MRC- und OSD-Abrfragen umsetzt. Ein separater Dienst, der *Replica Management Service (RMS)*, steuert die Erzeugung, Platzierung und das Löschen von Replikaten.

4 Das ZIB-DMS

Das *ZIB Data Management System (ZIB-DMS)* ist ein Langzeit-Forschungsprojekt zur Schaffung eines Datenmanagement-Systems für Grid-Systeme. Es wird am *Zuse-Institut Berlin (ZIB)* von der Gruppe *Computer-Science Research (CSR)*[1], unter Mitarbeit des Autors dieser Arbeit, entwickelt. Ziel ist die Verwaltung großer heterogener, verteilter Datenmengen, wie sie beispielsweise in wissenschaftlich genutzten Grid-Umgebungen [79, 90, 82] auftreten, sowie die Bereitstellung eines effizienten Zugriffs auf diese Daten in einem skalierbaren, zuverlässigen Grid-System. Dieses Kapitel beschreibt die Entwurfsprinzpien und den Aufbau des Systems. Auf detaillierte Beschreibungen der einzelnen Komponenten und des Systems wird an den entsprechenden Stellen verwiesen.

Einige Komponenten des Systems wurden bereits realisiert; Andere befinden sich noch in Planung oder wurden bisher rudimentär umgesetzt. In den folgenden Abschnitten wird der Entwicklungsstand des Projektes zu Beginn dieser Arbeit geschildert. Veränderungen, die an dem System vorbereitend und im Zuge des praktischen Teils dieser Arbeit vorgenommen wurden, werden in Kapitel 6 erörtert.

4.1 Entwurfsziele und Aufbau

Wie in Kapitel 3 dargestellt, ist das Datenaufkommen in Grid-Umgebungen immens und die Art und Weise, wie die Daten abgelegt werden, vielfältig. Diese Variabilität stellt ein Problem in Bezug auf Übersichtlichkeit, Verwaltbarkeit und Interoperabilität dar. Dieser Problematik kann durch die Etablierung einer Vermittlungsinstanz, die eine effiziente, transparente und zuverlässige Verwaltung solcher Daten ermöglicht, entgegengewirkt werden. Innerhalb des ZIB-DMS-Projektes werden die Möglichkeiten und Ansätze zur Schaffung einer solchen Instanz untersucht. Das ZIB-DMS ist ein Management-System für Grid-Daten; Das heißt, es werden nicht die Daten selbst, sondern *Metadaten* zu dem jeweiligen *Datenobjekt* gespeichert und verwaltet. Die zentralen Entwurfsziele hierbei sind:

Ortstransparenz

Vom physikalischen Namensraum, der den Speicherort der Daten spezifiziert, soll

[1] http://www.zib.de/CSR

abstrahiert und eine Abbildung auf einen eigenen, logischen Namensraum vorgenommen werden [56]. Durch diese Abstraktion des Namensraumes können Daten unabhängig von ihrem physikalischen Speicherort mittels eines logischen Namens angesprochen werden. Änderungen im physikalischen Namensraum bewirken eine Anpassung der Abbildung, verlaufen für den Benutzer aber transparent.

Zugriffstransparenz

Ebenso soll die Art und Weise, in der Daten abgelegt sind, abstrahiert werden. Das System bietet einen uniformen, transparenten Zugriff auf registrierte Datenobjekte, unabhängig von der Form, in der diese hinterlegt sind und der Anzahl der registrierten physikalischen Namen für ein einzelnes Datenobjekt. Dabei wird davon ausgegangen, dass Daten häufiger gelesen als geschrieben werden und nur schwache Konsistenz für Datenobjekte mit mehreren Quellen gefordert wird [55]. Lesende Zugriffe werden vom System auf einer ausgewählten Quelle durchgeführt oder auf mehere Quellen verteilt. Auch dies erfolgt für den Benutzer transparent. In diesem Zusammenhang soll auch die Verwendung von partiellen Replikaten unterstützt werden. Schreibzugriffe werden mit einer *write-once* Semantik unterstützt, die das Schreiben auf Datenobjekte nur für solche gestattet, für die nur ein einzelne Quelle registriert ist.

Dynamisches Metadaten-Schema

Um größtmögliche Flexibilität und Interoperabilität zu gewährleisten, wird kein statisches Metadaten-Schema, sondern ein offenes, attributbasiertes Schema verwendet. Metadaten werden als Schlüssel-Wert-Paare aufgefasst und gespeichert. Das System definiert einige grundlegende, zur Verwaltung notwendige Basis- beziehungsweise System-Attribute. Darüberhinaus kann der Benutzer beliebige Attribute dynamisch setzen.

Fehlertoleranz

In Grid-Systemen muss aufgrund der weiträumigen Vernetzung und heterogenen Topologie ständig mit dem Ausfall eines Knoten gerechnet werden. Für das Verwalten von Daten bedeuten diese Ausfälle Einschränkungen bezüglich der Verfügbarkeit und Leistungfähigkeit. Durch Daten-Replikation und geeignete Strategien zur Selbstorganisation und Selbstoptimierung soll die Robustheit und Ausfalltolleranz des Systems gesteigert werden.

Aktuelle Grid-Systeme, wie das *Globus Toolkit* [18] sind oft monolithisch oder nach dem Toolkit-Ansatz aufgebaut. Das ZIB-DMS hingegen ist als eine Kombination lose gekoppelter, kooperierender Komponenten konzipiert [51, 58]. Dabei wird grundsätzlich zwischen funktionalen Komponenten und solchen zur Selbstorganisation unterschieden. Erstere stellen Dienste für den Benutzer und andere Komponenten zur Verfügung und realisieren somit die eigentliche Funktionalität des Systems. Selbstorganisations-Komponenten überwachen den Zustand des Systems und wirken stabilisierend auf diesen ein.

Jede Komponente ist dabei ein eigenes *Peer-to-Peer (P2P)* System [51], das über ein eigenes *Overlay*-Netzwerk verbunden ist. In einem P2P-System ist jeder Knoten, oder *Peer*[2], gleichgestellt und kann als Client, Server oder beides auftreten. Jeder Peer besitzt eine einfache Schnittstelle zur Kommunikation mit anderen Knoten und führt den selben Algorithmus aus, der die Zusammenarbeit der Knoten des Systems koordiniert. Jede Komponente definiert eine Schnittstelle zur Bereitstellung ihrer Dienste und steuert die Ausführung des Peer-Algorithmus gemäß der jeweilgen Anfrage. Der P2P-Ansatz weist nach [58] unter anderem die im folgenden beschriebenen vier Eigenschaften auf, durch die er sich besonders gut für die Implementierung von großen Grid-Systemen eignet.

Einfachheit
P2P-Systeme sind im Allgemeinen extrem spezialisiert und verrichten meist einfache, überschaubare Aufgaben. Dies resultiert in einem einfachen, leicht wartbaren Programm-Code.

Redundanz
Die Funktionalität des Gesamtsystems wird durch viele zusammenwirkende, gleichrangige Knoten erbracht. Beim Ausfall eines Knotens kann dessen Aufgabe problemlos von einem anderen Knoten übernommen werden. Zwar wird dadurch keine Ausfallsicherheit gewährleistet, wohl aber für ein gewisses Maß an Redundanz gesorgt. Diese ist in großen Grid-Systemen von zentraler Bedeutung, da Komponenten und Dienste jederzeit ausfallen können. Somit ist es wichtig, dass sich das System ohne Eingreifen des Benutzers dynamisch reorganisieren kann.

Flache Struktur
Das P2P-Paradigma basiert auf demokratischen Prinzipien. Es gibt keine hierarchische Top-Down Verwaltung, nur gleichrangige Knoten, die den selben Algorithmus ausführen. Vorausgesetzt es sind genügend Peers vorhanden, um die Funktionalität zu realisieren, kann somit kein Engpass oder Single-Point of Failure *entstehen.*

Lokalität
Einzelne Knoten sind nicht auf eine globale Sicht auf das System angewiesen. Stattdessen wird ihr Verhalten von einem beschränkten, lokalen Informationshorizont bestimmt. Lokalität ist eine Grundvoraussetzung für Skalierbarkeit.

Ein komplexer Dienst, wie die Verwaltung von Metadaten und Replikaten, kann durch Komibation solcher einfacher, aufeinander aufbauender P2P-Komponenten bereitgestellt werden [51]. Abbildung 4.1 zeigt beispielhaft drei kooperierende P2P-Komponenten für verteiltes Datenmanagement; Einen Katalog zur Erfassung und Verwaltung von Replikaten, sowie zwei Selbstorganisations-Komponenten zur Lastbalanzierung und Platzierung von Replikaten. Das Gesamtsystem besteht aus einer Anzahl von Knoten. Auf jedem

[2] engl. Gleichgestellter, gleichrangig

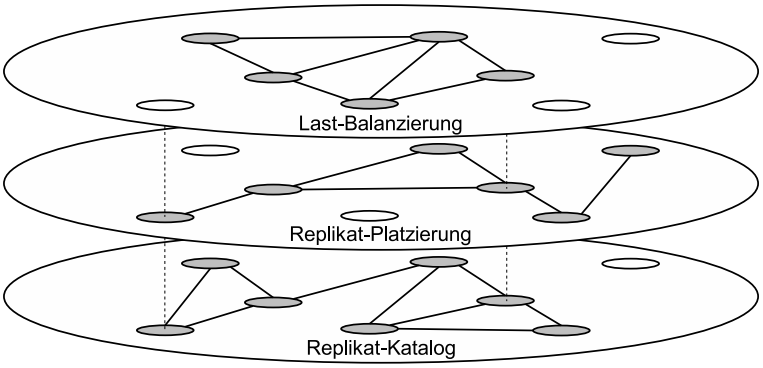

Abbildung 4.1: Drei P2P-Komponenten für verteiltes Datenmanagement nach [58].

Knoten werden Instanzen einer oder mehrerer Komponenten ausgeführt. Die Knoten einer Komponente sind über ein eigenes Overlay-Netzwerk miteinander verbunden.

Dieses Konzept ist in der aktuellen Version des ZIB-DMS allerdings nur in begrenztem Maße umgesetzt. Zwar setzt sich das System aus einzelnen, relativ unabhängigen Komponenten zusammen, als eigenständige P2P-Komponente ist derzeit aber nur der Metadaten-Katalog konzipiert. Es ist aber jederzeit problemlos möglich, auch andere Komponenten des Systems zu P2P-Komponenten auszubauen.

Die vorliegende Implementation des ZIB-DMS umfasst im Wesentlichen ein in *C++* geschriebenes Serverprogramm und eine mit der Programmiersprache *Java* realisierte graphische Oberfläche. Abbildung 4.2 zeigt den schematischen Aufbau eines ZIB-DMS Servers. Auf unterster Ebene, dem sogenannten *Backend*, befindet sich der verteilte Metadaten-Katalog. Hier werden alle Informationen des Systems abgelegt. Diese können über den *Directory-View* als hierarchischer Verzeichnisbaum dargestellt und verwaltet werden. Alternativ bietet der *Basic-View* einen direkteren Zugriff auf das System. Die funktionalen *Service-Komponenten* stellen weitere Dienste und Anwendungs-Logik zur Verfügung. Verschiedene *Optimierungskomponenten* überwachen und stabiliseren das System. Der Zugriff erfolgt grundsätzlich über die *ZIB-DMS API* oder die *NFS-Server*-Komponente. Für die ZIB-DMS API stehen neben der nativen *C++*-Schnittstelle Abbildungen für *CORBA*, *SOAP* und *Python* zur Verfügung, um die Anbindung an verschiedenste Client-Systeme zu ermöglichen.

4.2 Organisation der Daten im ZIB-DMS

Es wurde bereits mehrfach erörtert, dass sich die Verwaltung solcher Datenmengen, wie sie in Grid-Umgebungen auftreten, äußerst schwierig gestaltet. Vor allem das Auffinden von Daten wird mit zunehmendem Datenvolumen mühsamer. Der alleinige Einsatz altbekannter Verwaltungs-Mechanismen, wie hierarchische Verzeichnis-Systeme, stellt keine zufriedenstellende Lösung dar. Hierbei müssen Benutzer den Namen und den Pfad einer

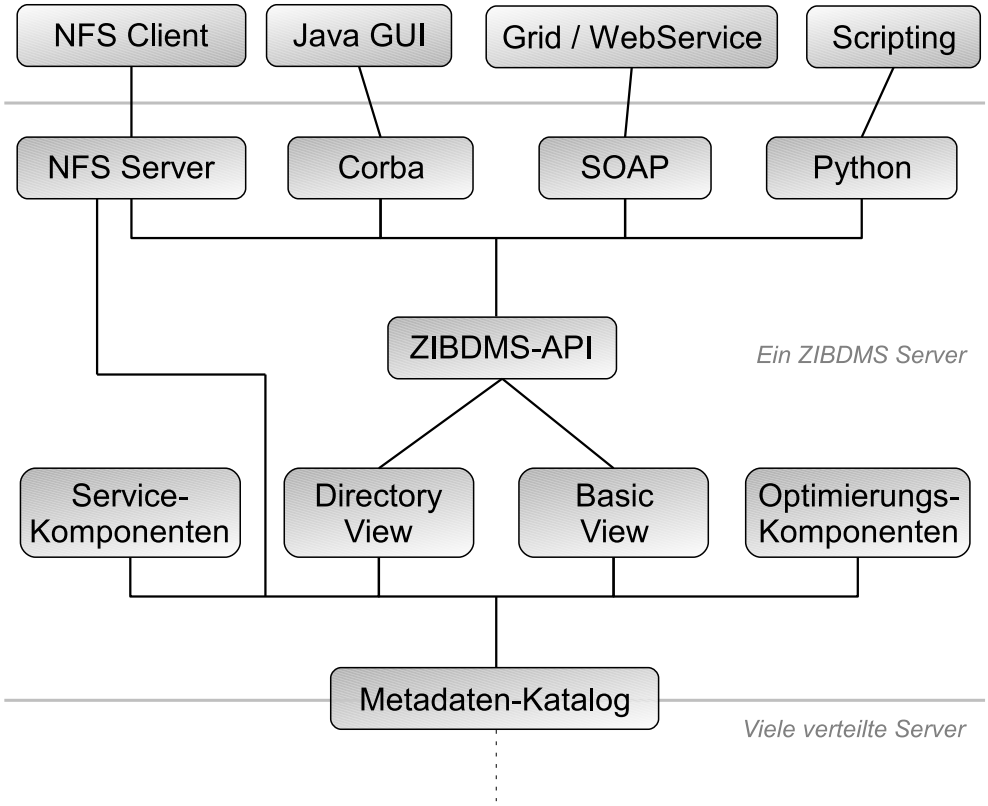

Abbildung 4.2: ZIB-DMS Architektur.

Datei kennen, um auf diese zugreifen zu können. Dieses Wissen ist oft nicht vorhanden, im Gegensatz zu anderen qualifizierenden Informationen. Beispielsweise ist einem Benutzer zwar nicht der Pfadname einer Datei, wohl aber der zeitliche und sachliche Kontext in dem die Datei erzeugt wurde, bekannt:

> *Lade die Daten des Strömungsexperimentes von letzter Woche, das Bob unter Assistenz von Paula durchgeführt hat.*

Die Grundvoraussetzung für derartige Anfragen ist die Verfügbarkeit von qualifizierenden Informationen. Diese können als Metadaten zu einem Datenobjekt abgelegt werden. Die Grundidee bei der Datenverwaltung des ZIB-DMS ist daher der *attributbasierte* Zugriff auf Datenobjekte. Das bedeutet, ein registriertes Datenobjekt wird als eine Sammlung von Schlüsseln und Werten im Metadaten-Katalog repräsentiert. Ein Schlüssel ist eine Zeichenkette, die ein Attribut eindeutig identifiziert. Einem Attribut können einzelne oder mehrere Werte zugewiesen werden. Diese werden als Zeichenketten oder 64-Bit große Zahlenwerte abgelegt. Das offene Metadaten-Schema des Systems erlaubt es dem Benutzer beliebige Attribute für ein verwaltetes Objekt einzutragen. Unter Verwendung eines einfachen Abfragemechanismus können diese Informationen verwendet werden, um Objekte im System direkt zu finden (siehe Abschnitt 4.3).

In anderen Situationen sind eventuell keine Informationen über ein Datenobjekt selbst, wohl aber über die Beziehung zu anderen bekannten Datenobjekten verfügbar:

> *Kopiere PDF-Versionen aller verfügbaren Artikel, die in ZIB-Report ZR_ 03_ 23 zitiert werden, in mein Home-Verzeichnis.*

Auch hier ist die Verfügbarkeit von Metainformationen Grundvoraussetzung. Um solche Zusammenhänge zwischen Datenobjekten verwenden zu können sind aber auch Mechanismen zur Erzeugung und Verwaltung dieser Beziehungen notwendig. Daher stellt das System neben dem attributbasierten Zugriff verschiedene Verwaltungsmechanismen zur Verfügung.

4.2.1 Verzeichnis-Hierarchie

Die für Benutzer gewohnte Form der Datenverwaltung ist deren Organisation in einem hierarchischen Verzeichnisbaum. In Kapitel 2 wurden die dabei verwendeten Konzepte, sowie deren Vor- und Nachteile erörtert. Diese bieten zwar keine Unterstützung für die oben beschriebenen Zugriffsmuster, eignen sich aber für das *Browsing* — also das manuelle Durchsuchen — und die grundlegende Organisation des Datenbestandes. Zudem sorgt der hohe Grad an Bekanntheit und Vertrautheit für ein großes Maß an Akzeptanz seitens der Nutzer.

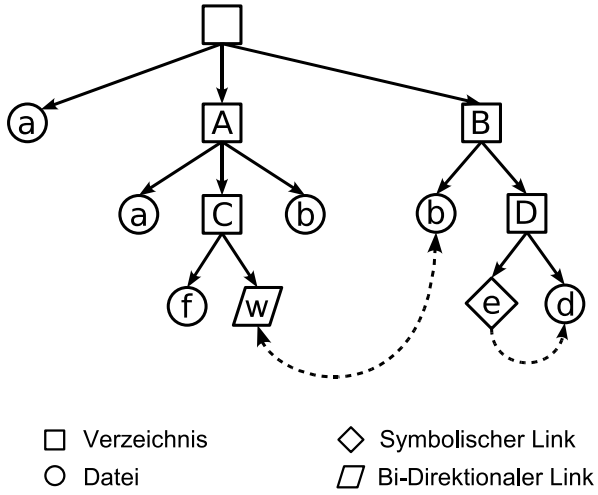

Abbildung 4.3: Hierachische Datenverwaltung im ZIB-DMS.

Daher ist die grundlegende Struktur, in der Datenobjekte im ZIB-DMS organisiert werden, ein hierarchischer Verzeichnisbaum. Dabei wird ein registriertes Datenobjekt als Datei aufgefasst, das in den systemeigenen Verzeichnisbaum eingeordnet wird. Abbildung 4.3 zeigt exemplarisch den Aufbau und die verschiedenen Elemente dieser Hierarchie. Es werden die in Kapitel 2 vorgestellten Konzepte *Datei*, *Verzeichnis*, *Hard-Link*

und *symbolischer Link (Symlink)* unterstützt und die dort beschriebenen Datei-Attribute als fest vordefinierte System-Attribute gespeichert. Zusätzlich wird das Konzept eines bi-direktionalen symbolischen Links, des *Weaklink* [46], eingeführt.

Ein Weaklink ist – ebenso wie ein symbolischer Link – ein eigenständiger Knoten im Verzeichnisbaum, der auf einen anderen Knoten verweist. Im Gegensatz zum symbolischen Verweis werden hier aber die erweiterten Möglichkeiten des Systems genutzt um diese Bindung auch bei Veränderungen im Verzeichnisbaum aufrecht erhalten zu können. Wird eine Datei, auf die ein symbolischer Link sowie ein Weaklink zeigt, verschoben oder umbenannt, zeigt ersterer ins Leere. Der Weaklink hingegen verweist auf den neuen Namen beziehungsweise den neuen Pfad der Datei. Zudem erlaubt es das System, eine Liste aller Weaklinks anzuzeigen, die auf eine bestimmte Datei verweisen. Auch dies ist mit symbolischen Links nicht direkt möglich.

Eines der zentralen Entwurfsziele des ZIB-DMS ist die Entkoppelung des logischen und physikalischen Namensraumes. Das System verwendet eine dreistufige Abstraktionsschicht, um diese *Ortstransparenz* bieten zu können. Diese verhält sich ähnlich zu der in manchen Dateisystemen verwendeten Inode-Abstraktion (siehe Abschnit 5.2, Kapitel 5). Zwischen logischem Bezeichner und den zugeordneten Daten wird eine Indirektions-Schicht etabliert, die beide Ebenen verbindet. Dies ist mit dem bekannten Konzept des *Foreign-Keys* [65] in Datenbank-Systemen vergleichbar.

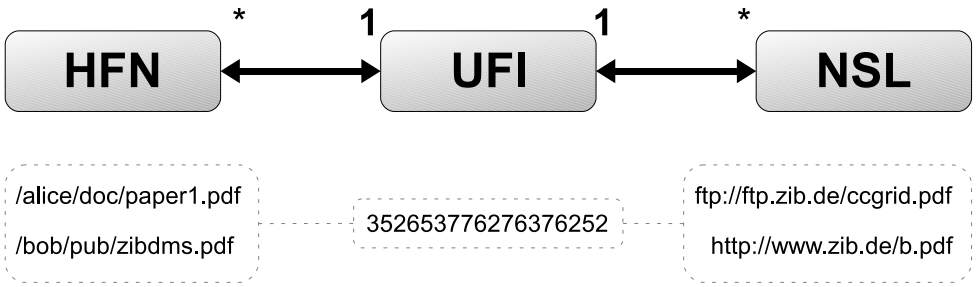

Abbildung 4.4: Beziehungen zwischen HFN, UFI und NSL.

UFI

Jedem Objekt in der Hierarchie wird ein *Unique File Identifier (UFI)* zugeordnet, der es eindeutig identifiziert. Da das ZIB-DMS verteilt arbeitet, muss sichergestellt werden, dass ein Objekt auch über Rechnergrenzen hinweg eindeutig identifizierbar bleibt. Daher wird an den Bezeichner die Forderung von globaler beziehungsweise universeller Eindeutigkeit gestellt. Zur Erzeugung solcher Kennungen können Algorithmen eingesetzt werden, wie sie in [29], [31] oder [43] beschrieben sind. In der aktuellen Implementation werden zur Generierung einer *Universal Unique ID (UUID)* die Funktionen der *libuuid*-Bibliothek verwendet. Diese Bibliothek ist Teil des *efsprogs*-Paketes [87] und wurde entwickelt, um *ext2*-Dateisysteme [13] universell eindeutig zu kennzeichnen.

HFN

Der Zugriff auf ein Objekt im Verzeichnisbaum erfolgt durch Angabe eines Pfadnamens, dem *Hierarchical File Name (HFN)*. Dessen Gestalt entspricht derer absoluter Pfadnamen, wie sie von hierarchischen Dateisystemen bekannt sind (siehe Abschnitt 2.1.2, Kapitel 2). Als Pfad-Trennzeichen wird, wie bei Unix-ähnlichen Systemen, der Schrägstrich ('/') verwendet.

NSL

Der phsyikalische Speicherort eines Datenobjektes wird als *Native Storage Location (NSL)* registriert. Dieser wird angegeben und gespeichert als *Uniform Resource Locator (URL)* [5]. Welche Arten von Datenquellen registriert weden können hängt davon ab, welche URL-Schemata das System erkennt und unterstützt. Die Unterstützung neuer Typen kann durch Erweiterung der für den Datenzugriff verantwortlichen Komponente des Systems erreicht werden (siehe Abschnitt 4.6).

Die Beziehungen zwischen diesen drei Abstraktionsebenen werden in Abbildung 4.4 zusammengefaßt und an einen einfachen Beispiel veranschaulicht. Es besteht eine $N:1:N$ Beziehung zwischen HFN, UFI und NSL. Damit ist es grundsätzlich möglich ein durch einen UFI identifiziertes Datenobjekt anzulegen, das keinen zugehörigen HFN besitzt. Dies ist allerdings nur für systeminterne Zwecke sinnvoll und wird von der Zugriffs-Schicht unterbunden. Somit wird sichergestellt, dass einem durch den Benutzer angelegten Datenobjekt immer mindestens ein HFN zugeordnet ist. Die Zuweisung mehrer HFNs entspricht dem Hard-Link Konzept hierarchischer Dateisysteme. Ebenso ist es möglich ein Datenobjekt ohne einen zugehörigen NSL anzulegen. Diese Situation entsteht beispielsweise als Zwischenschritt bei der Registrierung neuer Objekte und wird daher von der Zugriffs-Schicht nicht unterbunden.

4.2.2 Assoziationen

Die Zuordnung von Datenobjekten zu einem Verzeichnis kann als Abbildung einer Assoziation zwischen diesen aufgefasst werden. In einer hierarchischen Baum-Struktur ist jedes Datenobjekt aber genau einem Verzeichnis fest zugeordnet. Daher ist es nicht möglich, mehrere unterschiedliche Assoziationen zwischen den selben Datenobjekten auszudrücken. Dieses Problem kann durch das Anlegen zusätzlicher Verzeichnisse mit Verweisen auf die Objekte umgangen werden. Die Verwaltung und Pflege dieser Strukturen gestaltet sich aber ziemlich mühsam und unübersichtlich, so dass dieses Vorgehen keine zufriedenstellende Lösung darstellt. Das ZIB-DMS bietet einen separaten Mechanismus zur freien Gruppierung beziehungsweise Assoziierung von Datenobjekten – die *Collection*.

Eine Collection ist ein systemweit globaler, vom hierarchischen Verzeichnisbaum unabhängiger, benannter Container für Datenobjekte. Ein Benutzer kann beliebige Collection-Container anlegen und Datenobjekte als Mitglied registrieren. Jedem Collection-Objekt

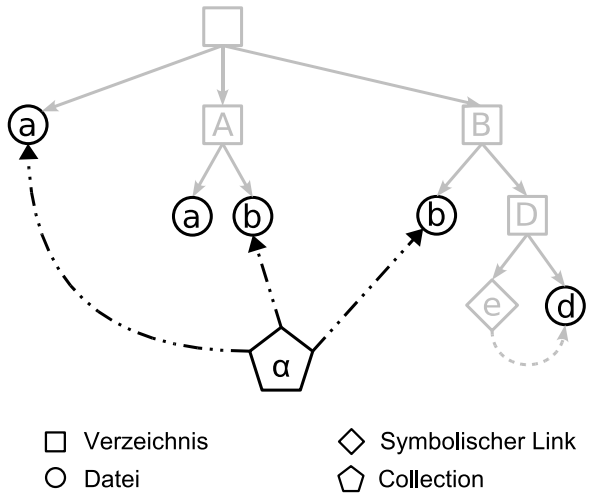

Abbildung 4.5: Collection – Verwaltung von Assoziationen im ZIB-DMS.

wird ebenso wie den Objekten der Verzeichnis-Hierarchie ein UFI-Bezeichner für die systeminterne Identifikation zugewiesen. Da Collections außerhalb der Verzeichnis-Hierarchie stehen, erfolgt die Benennung jedoch nicht durch HFN-Pfade, sondern innerhalb eines separaten, flachen Namensraumes. Ein Datenobjekt kann Mitglied in beliebig vielen Collections sein. Somit können beliebige Assoziationen von Datenobjekten, unabhängig von deren Position in der Verzeichnis-Hierarchie, ausgedrückt werden. Eine Collection kann als Bündelung von Datenobjekten zu einer Einheit gesehen werden. Diese kann ebenfalls mit beliebigen Attributen versehen werden, so dass der attributbasierte Zugriff auch auf Gruppierungen von Objekten möglich ist. Geplant ist hier auch die Erweiterung der Semantik für Operationen auf Datenobjekten, die Mitglied in einer Collection sind. Beispielsweise könnte das Verschieben in der Verzeichnis-Hierarchie eines in einer Collection befindlichen Datenobjektes auf alle Mitglieder der Collection übertragen werden. In Abbildung 4.5 wird das Konzept der Collection anschaulich dargestellt. Drei der fünf dargestellten Datenobjekte sind Mitglied in der Collection α. Die hellgrau dargestellten Objekte und Verbindungen zeigen, wie die Datenobjekte in der Verzeichnis-Hierarchie organisiert sind.

4.2.3 Relationen

Ein weiterer Schwachpunkt der hierarchischen Verwaltung von Daten ist die mangelnde Möglichkeit beliebige Beziehungen zwischen den einzelnen Objekten ausdrücken zu können. Auch hier bietet das System einen eigenen Mechanismus – das *Graph*-Konzept. Ein Graph ist ebenso wie eine Collection ein systemweit globales, von der Verzeichnis-Hierarchie unabhängiges, benanntes Konstrukt, dessen Identifikation über einen UFI-Bezeichner oder einen Namen erfolgt. Dabei wird auch hier ein eigener, flacher Namensraum verwendet. Ein Graph wird als eine benannte Sammlung von gerichteten *Kanten*

betrachtet. Eine Kante – durch einen UFI identifiziert – verbindet Datenobjekte als Quell- und Zielknoten. Hierbei können auch mehrere Quellen und Ziele angegeben werden. Sowohl das Graph-Objekt selbst, wie auch dessen Kanten können mit beliebigen Attributen versehen werden. Damit wird beispielsweise die Abbildung gewichteter Kanten ermöglicht.

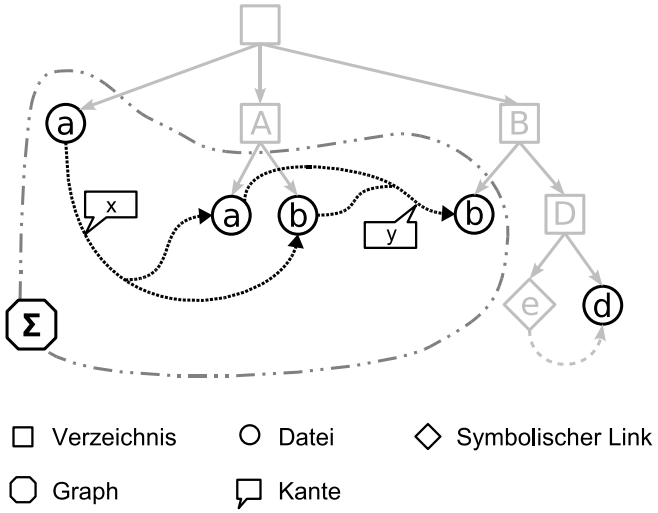

Abbildung 4.6: Graph – Abbildung von Relationen im ZIB-DMS.

Dieser Mechanismus ermöglicht es, parallel zur Verzeichnis-Hierarchie beliebige Graphen auf den Datenobjekten aufzubauen. Abbildung 4.6 zeigt hierfür ein Beispiel. Der Graph Σ umfasst zwei Kanten $\mathbf{x}$ und $\mathbf{y}$. Wobei erstere einen Quell- und zwei Zielknoten, die zweite hingegen zwei Quell- und einen Zielknoten besitzt. Auch hier ist die Organisation dieser Datenobjeke im Verzeichnisbaum grau dargestellt. Durch die zur Verfügung gestellten Funktionen der Graph-Komponente können zu jedem Datenobjekt alle zugehörigen Graphen und die dort eingetragenen Relationen abgefragt werden. Somit wird das oben beschriebene Abfragen von Objekten anhand von Beziehungen zu anderen Objekten ermöglicht.

4.3 Metadaten-Katalog

Der *Metadata Catalogue (MDC)* ist die zentrale funktionale Komponente des Systems. Hier werden alle Metadaten zu den im System registrierten Datenojekten gespeichert. Diese umfassen Attribute der Verwaltungsmechanismen und qualifizierende Informationen zu den Objekten, aber auch Angaben über Speicherorte und Replikate. Der MDC ist im Sinne der in Abschnitt 4.1 erörterten Entwurfsprinzpien als eigenständige P2P-Komponente konzipiert. Die Informationen des Katalogs werden verteilt in zahlreichen, durch ein Overlay-Netzwerk verbundenen MDC-Knoten gespeichert. Die Anwendung

des P2P-Paradigmas bedingt ein redundantes und dezentrales Speichern der Metadaten. Dadurch wird ein *Single Point of Failure* vermieden und eine höhere Ausfall- und Fehlertoleranz erreicht.

Der attributbasierte Katalog legt die Metadaten eines Objektes als eine Sammlung von Schlüsseln und Werten ab. Eine einfache, objektorientierte Schnittstelle ermöglicht die Kommunikationen zwischen MDC-Knoten und bietet anderen Komponenten des Systems Zugriff auf den Katalog. Dies erfolgt unter Verwendung von `Object`- und `Query`-Objekten. Eine `Object`-Datenstruktur repräsentiert ein registriertes Datenobjekt und dessen Attribute. Um Anfragen an den Katalog richten zu können, stehen verschiedene `Query`-Objekte für Vergleichs- und Bereichsabfragen, sowie logische Verknüpfungen zur Verfügung. Durch Kombination verschiedener `Query`-Objekte entsteht somit ein einfacher, sprachunabhängiger Abfragemechanismus.

```
Query q1 = new EqualsQuery("TYPE","STREAM_EXPERIMENT");
Query q2 = new  RangeQuery("CREATED", LastWeek.begin(), LastWeek.end());
Query q3 = new EqualsQuery("AUTHOR", "Bob");
Query q4 = new EqualsQuery("ASSIST", "Paula");
Query aq = new AndQuery(q1, q2, q3, q4);
set<Object> result = mdc.getObjects(aq);
```

Listing 4.1: Beispiel für eine MDC-Anfrage.

Der Pseudo-Code in Listing 4.1 zeigt exemplarisch, wie eine der in Abschnitt 4.2 formulierten Anfragen an den MDC gerichtet werden kann. Vorausgesetzt die entsprechenden Attribute wurden beim Anlegen gesetzt, enthält die Ergebnismenge das gewünschte Objekt.

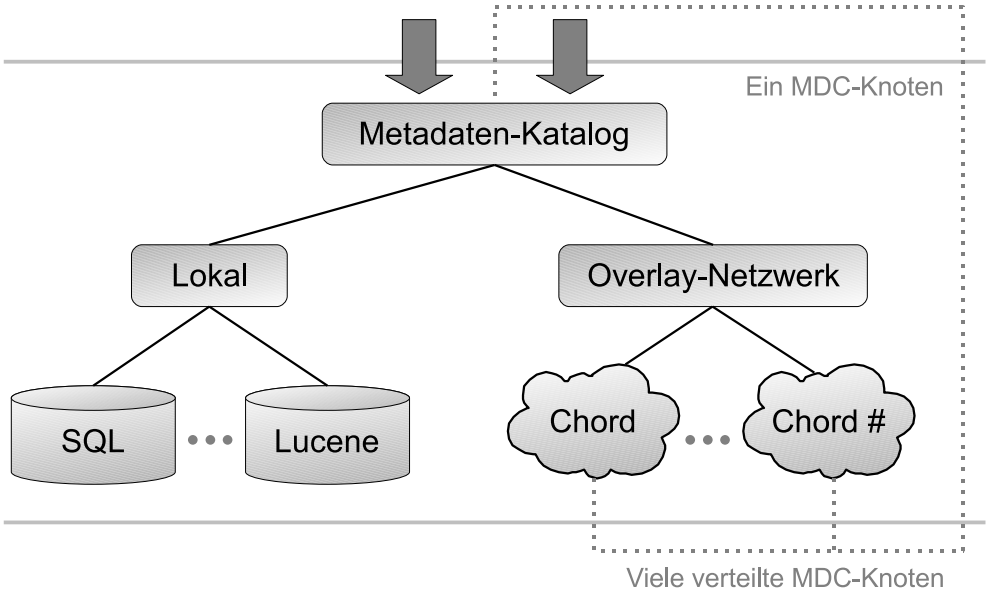

Abbildung 4.7: Aufbau der MDC-Komponente.

Abbildung 4.7 zeigt den Aufbau der Metadaten-Katalog-Komponente. Jeder MDC-Knoten verfügt über einen lokalen Datenspeicher zur persistenten Speicherung der Metadaten, für die er zuständig ist. Metadaten, die außerhalb des Zuständigkeitsbereiches eines MDC-Knotens liegen, werden über das P2P-Netzwerk von anderen Knoten abgefragt. Dieser bildet die MDC-Schnittstelle und die verwendeten Datenstrukturen auf einen bestimmten Speichermechanismus ab. Die aktuelle Implementation stellt Anbindungen für die SQL-Datenbanksysteme *MySQL* [91] und *sqlite* [97] bereit. Eine Anbindung an das Indexing-System *Lucene* [83] ist in Planung. Die Verbindung zwischen MDC-Knoten ist über verschiedene Overlay-Netzwerktopologien denkbar. Geplant ist der Einsatz der Protokolle *Chord* [69] und des ebenfalls am *ZIB* entwickelten *Chord#* [52, 62, 49].

4.4 Darstellung

Die Schaffung eines transparenten Zugriffs auf das System und die damit verbundene Virtualisierung der verwalteten Speicher-Ressourcen ist ein zentrales Entwurfsziel des ZIB-DMS. Neben eigens entwickelten proprietären Zugriffs-Möglichkeiten ist deshalb die Abbildung der Funktionalität des Systems und dessen verwalteten Ressourcen auf verschiedene, etablierte Zugriffsmechanismen vorgesehen. Um ein möglichst großes Maß an Flexibilität und Erweiterbarkeit zu gewährleisten, erfolgt die Umsetzung dieser Abbildungen auf zwei Ebenen. Ähnlich dem Prinzip der Anwendungs- und Darstellungs-Schicht des OSI-Modells [103], wird zwischen Zugriffs- und Darstallungs-Ebene unterschieden. Zugriffs-Komponenten (siehe Abschnitt 4.5) fungieren als Adapter [23] für dedizierte Schnittstellen. Darstellungs-Komponenten bieten eine eigene, abgeschlossene Sichtweise auf die funktionalen Komponenten des Systems und die damit verbundenen Operationen. Bisher wurden zwei solcher sogenannter *Views*[3] realisiert:

Die **Basic-View**-Komponente bietet eine direkte, objektorientierte Abbildung des Systems. Die verschiedenen Verwaltungsmechanismen und deren verwendeten Datenstrukturen, sowie andere funktionale Komponenten werden als Objekte mit eigenen Schnittstellen zur Abfrage und Manipulation dargestellt. Die Identifikation der Ressourcen erfolgt anhand von UFI-Bezeichnern. Der Name 'Basic-View' soll hierbei unterstreichen, dass diese Sichtweise die Struktur und Funktionalität der internen Komponenten direkt widerspiegelt.

Die **Directory-View**-Komponente stellt unter Verwendung der in Kapitel 2 dargestellten Datei-Metapher eine hierarchische Darstellung des Systems zur Verfügung. Sie ist, wie in Abbildung 4.8 dargestellt, in mehreren Ebenen aufgebaut, von denen jede für die hierarchische Abbildung einer ZIB-DMS-Komponente verantwortlich ist. Datenobjekte werden basierend auf der Struktur der hierarchischen Verwaltungs-Komponente

[3] Diese Bezeichnung wurde in Anlehnung an den aus Datenbank-Management-Systemen (DBMS) bekannten Begriff *View* [65] gewählt, um auszudrücken, dass diese Komponenten verschiedene abgeschlossene Sichtweisen auf den selben Datenbestand darstellen.

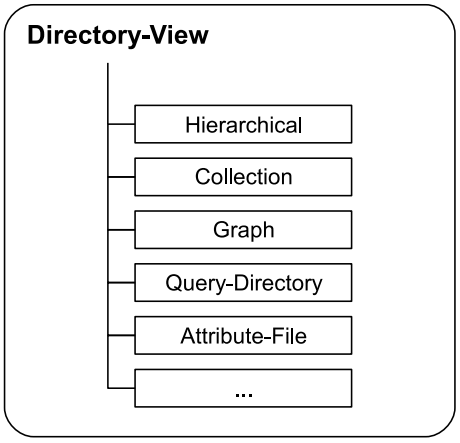

Abbildung 4.8: Aufbau der Directory-View Komponente.

in einer Verzeichnis-Hierarchie angezeigt. Die Semantik der übrigen Komponenten wird auf diese Struktur projiziert. Dies erfolgt durch die Abbildung der Funktionalität der verschiedenen Komponenten auf einen einheitlichen, limitierten Satz von Operationen. Listing 4.2 zeigt diese Schnittstelle. Die Signaturen und die Semantik wird größtenteils von den POSIX-Datei-Operationen (siehe Tabelle 5.1 auf Seite 45) übernommen, wobei die Benennung der Operationen in einigen Fällen abweicht. Die Ausnahme bilden hier die Operationen *read()* und *write()*, die anders als im POSIX-Standard definiert zustandslos arbeiten. Das heisst, Lese- und Schreiboperationen auf einem Datenobjekt können direkt erfolgen, ohne dieses vorher durch einen speziellen Mechanismus öffnen zu müssen. Daher werden diese beiden Operationen nicht mit einem *Filehandle*-Bezeichner, sondern mit dem Pfadnamen der Datei gerufen.

```
/* Verwaltung von Dateiattributen und Metadaten */
vector<pair<Key, Value> >
      getAllAttributes ( const Hfn &path, const vector<Key> &keys );
Value getAttribute     ( const Hfn &path, const Key &key );
void  addAttribute     ( const Hfn &path, const Key &key, const Value &value );
void  setAttribute     ( const Hfn &path, const Key &key, const Value &value );
void  removeAttribute  ( const Hfn &path, const Key &key );
void  removeAttribute  ( const Hfn &path, const Key &key, const Value &value );

/* Abbildung von UFI auf HFN */
vector<Hfn> getHFNV  ( const Uuid &ufi );

/* Verzeichnis- und Datei-Operationen */
vector<Hfn> ls ( const Hfn &path );
void mkdir     ( const Hfn &path );
void rmdir     ( const Hfn &path );
void create    ( const Hfn &path );
void ln        ( const Hfn &src, const Hfn &dest );
void unlink    ( const Hfn &path );
void symlink   ( const Hfn &path, const Hfn &linkinfo );
void rename    ( const Hfn &oldpath, const Hfn &newpath );
ssize_t read   ( const Hfn &path, off_t offset, void *buf, size_t count );
ssize_t write  ( const Hfn &path, off_t offset, void *buf, size_t count );
```

Listing 4.2: Ursprüngliche Directory-View Schnittstelle.

Diese Operationen bilden die Schnittstelle, die sowohl den Zugriff auf die Directory-View-Komponente regelt, als auch von den einzelnen Ebenen implementiert wird. Dabei wird nicht jede Operation von jeder Ebene unterstützt. Beispielsweise bietet die *Query-Directory*-Ebene, die Ergebnisse von Such-Anfragen an das System in virtuellen Verzeichnissen darstellt, keine Möglichkeit Dateien anzulegen, weswegen die `create()`-Funktion hier nicht implementiert ist. Eine eingehende Beschreibung der einzelnen Ebenen der Directory-View-Komponente, der dadurch realisierten Abbildungen und deren unterstützten Operationen erfolgt in Kapitel 6. Die Directory-View-Komponente entscheidet beim Aufruf einer Operation, abhängig von deren Art und Parameter, ob diese an alle Ebenen gerichtet und Ergebnisse akkumuliert oder eine dedizierte Ebene angesprochen werden soll. Die Identifikation der Datenobjekte sowie der zuständigen Ebene erfolgt dabei anhand des HFN.

4.5 Zugriff

Die Funktionalität und Informationen des Systems werden dem Benutzer durch die Komponenten der oberen Ebenen der ZIB-DMS-Architektur verfügbar gemacht. Um ein größtmögliches Maß an Interoperabilität gewährleisten zu können, bietet das ZIB-DMS Anbindungen für verschiedene Client-Anwendungen und Umgebungen.

Im Mittelpunkt steht hierbei die *ZIB-DMS-API*. Diese umfangreiche Programmier-Schnittstelle bietet Zugriff auf den vollen Funktionsumfang des Systems. Sie umfasst sowohl die Operationen des Basic-Views, als auch die des Directory-Views. Die native C++-Schnittstelle des ZIB-DMS-Servers ist für die lokale Verwendung durch Zugriffskomponenten und zukünftige Erweiterungen gedacht. Für den system- und plattformübergreifenden Zugriff wird die Schnittstelle auf unterschiedliche Weise exportiert:

CORBA

Die *Common Object Request Broker Architecture (CORBA)* [41, 25] der *Object Management Group (OMG)* [98] spezifiziert eine objektorientierte Middleware um die Entwicklung von verteilten Anwendungen in heterogenen Umgebungen zu vereinfachen. Die Architektur der Middleware ist nach dem Broker-Pattern [23] konzipiert. Ziel ist es, die Komplexität der zugrundeliegenden Netzwerk-Kommunikation vor dem Programmierer zu verbergen, so dass entfernte Aufrufe wie lokale Aufrufe erscheinen. Ein zentraler Aspekt ist dabei die Programmiersprachen- und Plattform-Unabhängigkeit. So ist die Spezifikation nicht an eine konkrete Programmiersprache gebunden. Stattdessen wird die ebenfalls von der OMG entwickelten *Interface Definition Language (IDL)* verwendet, um eine sprachunabhängige formale Spezifikation der Schnittstelle zu erstellen, die eine Software zur Verfügung stellt. Verschiedene IDL-Compiler erzeugen daraus den Quelltext für eine konkrete Programmiersprache. Basierend auf diesen sogenannten Stub- und Skeleton-Klassen können dann Implementationen der Client- beziehungsweise Server-Seite erzeugt

werden.

Die Datentypen und Operationen der ZIB-DMS-API werden in einer IDL-Spezifikation erfasst und veröffentlicht. Die Zugriffs-Komponente *CORBA-ZIB-DMS-API* stellt die serverseitige Implementation dieser CORBA-Schnittstelle. Sie übernimmt die Umwandlung der Datentypen (*Marshalling*), leitet die CORBA-Aufrufe entsprechend an die ZIB-DMS-API weiter und gibt deren Ergebnisse über die CORBA-Mechanismen zurück. Dabei wird die für C++ und Python verfügbare CORBA-ORB-Implementation *omniORB4* [94] verwendet.

SOAP

SOAP [8] ist ein vom *W3C*[99] entwickeltes, leichtgewichtiges Protokoll zum Austausch XML-kodierter Nachrichten über Standard-Internet-Protokolle, das zur Kommunikation mit einem *WebService* verwendet wird. Ein WebService ist ein meist über das *World Wide Web* erreichbarer, eindeutig identifizierbarer Dienst, dessen Schnittstelle unter Verwendung der *Web Services Description Language (WSDL)* beschrieben und veröffentlicht wird. SOAP definiert unter anderem Mechanismen für entfernte Prozeduraufrufe (*XML-RPC*) über SOAP-Nachrichten und Regeln für die Abbildung von Datenstrukturen in XML-Nachrichten. Für gewöhnlich wird für die Kommunikation eine Kombination aus HTTP und TCP verwendet. SOAP eignet sich dadurch besonders für verteilte Anwendungen in WAN-Umgebungen oder Umgebungen, in denen starke Beschränkungen auf die verwendeten Netzwerkprotokolle herrschen. Viele Grid-Systeme verwenden diese serviceorientierte Architektur (*SOA*) von WebServices zur Bereitstellung von Diensten und losen Koppelung von Systemen. Die vom *Global Grid Forum (GGF)* [95] entwickelte *Open Grid Services Architecture (OGSA)* [6] basiert beispielsweise auf WebService-Technologien, die synonym auch als *GridServices* bezeichnet werden.

Die SOAP-Zugriffs-Komponente des ZIB-DMS kann auf zwei Arten genutzt werden. Zum Einen wird die ZIB-DMS-API in Form einer WSDL-Schnittstellendefinition veröffentlicht und deren serverseitige Implementation gestellt, so dass die Prozeduren der API als WebService verwendet werden können. Zum Anderen kann die Komponente als Adapter für etablierte GridService-Schnittstellen, wie zum Beispiel OGSA-DAI [6] verwendet werden, um die Funktionalität des ZIB-DMS auf diese abzubilden und damit die Integration in bestehende Grid-Umgebungen zu erleichtern.

Python

Python [32] ist eine sehr flexible, dynamische, objektorientierte Programmiersprache, die sich unter anderem durch eine einfache übersichtliche Syntax und leichter Erlernbarkeit auszeichnet. Sie bietet eine starke dynamische Typisierung, automatisierte Speicherverwaltung (*Garbage Collection*), Unterstützung für aspektorientierte und funktionale Programmierung und die Möglichkeit, Programm-Module, die in anderen Programmiersprachen geschrieben wurden, einfach einzubinden. Python wird daher oft von Anwendungen zur Erweiterung als Automations- oder Skript-Sprache benutzt.

Unter Verwendung der *boost-python*-Bibliothek [81] werden ein Python-Interpreter ins System integriert und die API-Funktionen sowie deren verwendete Datenstrukturen in einem Python-Modul exportiert. Somit kann aus einem Python-Script heraus, das vom ZIB-DMS Server zur Laufzeit geladen wird, auf die ZIB-DMS-API zugegriffen werden. Dies bildet die Grundlage für einen Plugin-Mechanismus des Servers.

Graphische Schnittstelle

Mit der eigens entwickelten *ZIB-DMS-GUI* wird eine graphische Benutzerschnittstelle zur übersichtlichen Visualisierung, Zugriff und Steuerung des Systems bereitgestellt. Die Oberfläche erlaubt den Zugriff auf die Dienste der verschiedenen Verwaltungsmechanismen, sowie einiger Service-Komponenten. Das Bedienkonzept orientiert sich dabei im Grundsatz an den graphischen Dateibrowser-Anwendungen, wie sie in gängigen Betriebssystemen verwendet werden.

Der Datenbestand des ZIB-DMS wird übersichtlich dargestellt und kann mittels Kontext-Menüs oder *Drag and Drop*-Operationen manipuliert werden. Der Screenshot in Abbildung 4.9 zeigt die Darstellung der Verzeichnis-Hierarchie und die Attribut-Matrix zur Verwaltung der Metadaten. Die ZIB-DMS-GUI ist in Java [24] geschrieben und damit plattformunabhängig einsetzbar. Die Kommunikation mit einem ZIB-DMS Server erfolgt über die CORBA-Anbindung der ZIB-DMS-API.

NFS-Server

Die *NFS-Server*-Komponente implementiert das von der Firma *SUN* entwickelte NFS-Protokoll in der Version 3. Sie erlaubt es, den Datenbestand des ZIB-DMS in den Verzeichnisbaum eines Systems transparent einzubinden und darauf zuzugreifen. Dazu wird die Funktionalität des Systems auf die der Dateisystem-Schnittstelle abgebildet (siehe Abschnitt 4.4). Beispielsweise wird die Verwaltung von Metadaten über spezielle virtuelle Dateien realisiert, so dass die Attribute eines Datenobjektes mit einem gewöhnlichen Text-Editor bearbeitet werden können.

Die Nutzung dieser vertrauten Schnittstelle für den Zugriff auf das System hat mehrere Vorteile. Einerseits ermöglicht es Benutzern, das System ohne spezifisches Vorwissen zu verwenden. Andererseits wird dadurch die Integration in etablierte Umgebungen erleichtert, da vorhandene Anwendungen, die Daten aus dem Dateisystem laden oder sie dort ablegen, ohne oder nur mit geringfügigen Anpassungen verwendet werden können. Zudem ist die Automatisierung von Abläufen durch den Einsatz gewohnter Hilfsmittel wie Shell-Scripten möglich.

Diese Form des Zugriffs schafft die Grundlage für ein einfaches, transparentes und intuitives Arbeiten mit dem System. Die vorliegende Version der NFS-Komponente hat

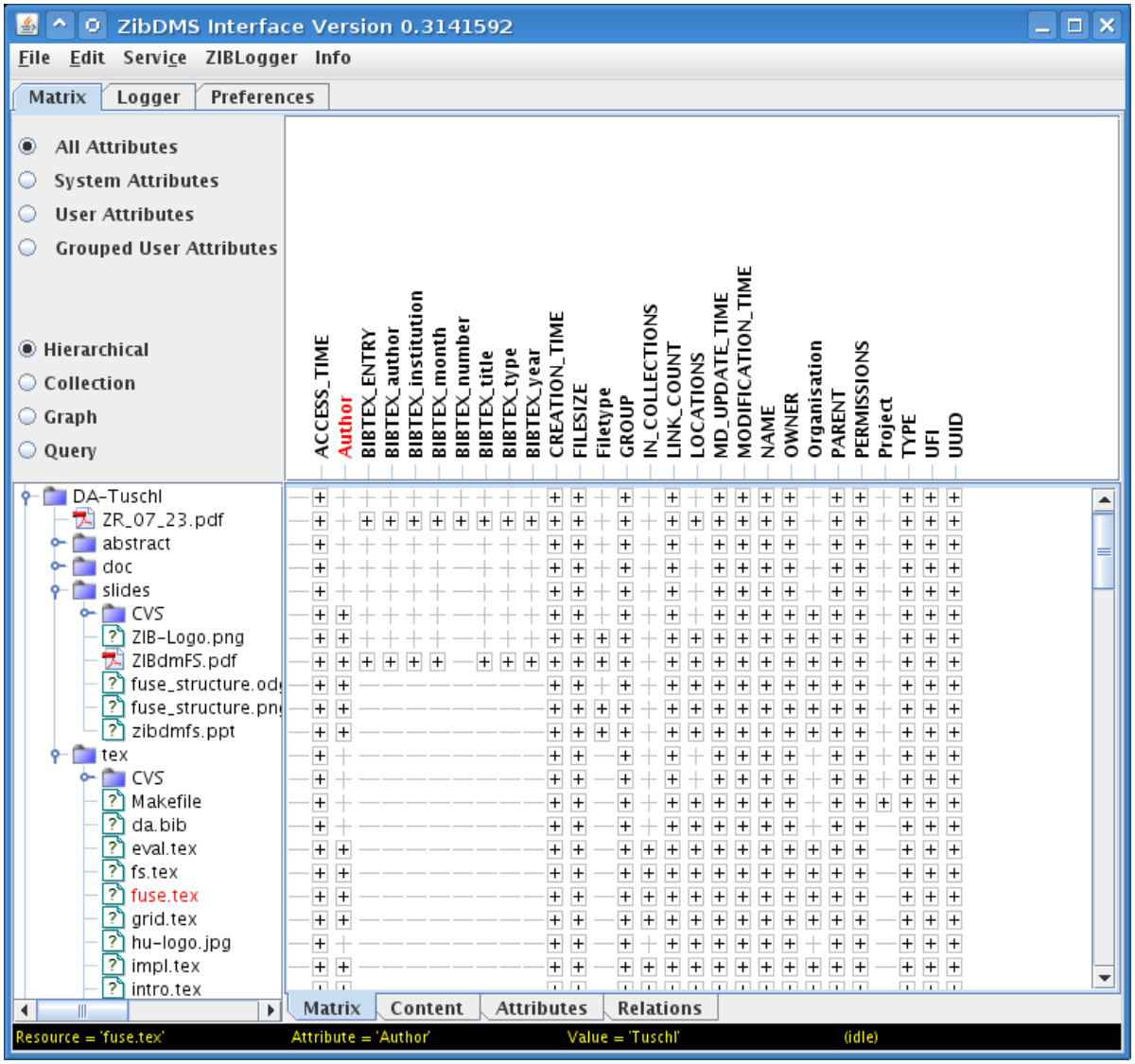

Abbildung 4.9: Die graphische Schnittstelle des ZIB-DMS.

allerdings mit einigen Problemen zu kämpfen. Diese werden in Kapitel 5 eingehend erläutert.

4.6 Service-Komponenten

Neben den Komponenten zur Verwaltung von Metadaten und Replikaten umfasst das System weitere funktionale *Service-Komponenten*. Diese stellen verschiedene Dienste zur Verfügung.

File-Access

Das ZIB-DMS liefert einen tranparenten Zugriff auf den Inhalt von Dateien, für die eine oder mehrere URLs als Replikate registriert wurden. Die *File-Access* Komponente definiert eine Schnittstelle zum Lesen und Schreiben einer als URL angegebenen Datenquelle. Die Unterstützung verschiedener URL-Schemata erfolgt durch Bereitstellung eigener Instanzen, welche die *File-Access*-Schnittstelle implementieren und den dateibasierten Zugriff des ZIB-DMS auf die jeweiligen Datenquellen abbilden. Aktuell stehen zwei solche Instanzen zur Verfügung: *CURL-File-Access* und *CORBA-File-Access*. Die *CURL-File-Access*-Komponente bietet Unterstützung für die Protokolle *FILE*, *HTTP*, *FTP* [5] unter Verwendung der *CURL*-Bibliothek [85]. Der Schreibzugriff wird hier allerdings nicht von allen Protokollen unterstützt. *CORBA-File-Access* bedient das proprietäre *CORBA Data Transfer Protocol (CDTP)* [39] zum Austausch von Daten zwischen ZIB-DMS Server-Instanzen über eine eigene CORBA-Schnittstelle. Dieses wird durch die unten beschriebene Service-Komponente *Storage-Management* verarbeitet. Andere Typen von Datenquellen und Zugriffsmechanismen können durch Hinzufügen weiterer *File-Access*-Module einfach integriert werden. So ist beispielsweise die Integration von Mechanismen für einen effizienten, partiellen Zugriff auf entfernte Dateien geplant [60].

File-Transfer

Das in Abschnitt 4.7 vorgestellte *Replica-Placement* legt bei Bedarf neue Replikate an beziehungsweise verschiebt diese. Das Verschieben und Kopieren kompletter Dateien wird durch die *File-Transfer*-Komponente unter Verwendung der *File-Access*-Dienste bewerkstelligt. Auch Funktionen dieser Service-Komponente werden als CORBA-Schnittstelle über das Netzwerk zugänglich gemacht.

Assimilation

Grundsätzlich müssen Objekte und deren Metadaten vom Benutzer manuell im ZIB-DMS registriert und gepflegt werden. Sollen große Mengen von Objekten registriert werden, kann sich dies sehr aufwändig gestalten. Die *Assimilation*-Komponente definiert eine Schnittstelle zur Implementierung von Diensten, die diesen Vorgang automatisieren, beziehungsweise zu unterstützen sollen. Hier sind vielfältige Anwendungsszenarien und Mechanismen denkbar. Zum Beispiel kann

die Registrierung ganzer Verzeichnis-Teilbäume durch Angabe einer URL automatisiert betrieben werden. Viele Dateiformate bieten desweiteren die Möglichkeit Metainformationen über die enthaltenen Daten innerhalb der Datei selbst zu speichern. Meist erfolgt dies innerhalb von fest definierten und strukturierten Bereichen der Datei. Als populäre Beispiele wären hier die Dateiformate *MP3*, *JPEG* oder *PDF* zu nennen. Die Informationen können aber auch in separaten Dateien oder Datenquellen abgelegt sein, wie beispielsweise eine zu einem *PDF* zugehörige *BibTeX* Datei. Bei der Registrierung im ZIB-DMS wäre es wünschenswert, solche Informationen automatisiert ins System übernehmen zu können. Dies kann durch die Bereitstellung entsprechender *Assimilation*-Module erfolgen. Die aktuelle Version des ZIB-DMS beinhaltet eine *URL-Assimilation*-Komponente. Mit deren Hilfe können ausgewählte Bereiche einer von *CURL* unterstützen Datenquelle automatisiert im System registriert werden.

Storage-Management

Grundsätzlich verwaltet das ZIB-DMS nur Metadaten und Quellangaben von Datenbeständen und bietet einen transparenten Zugriff auf die ansonsten aber extern gespeicherten Daten. Die Selbstorganisations-Komponenten des Systems benötigen aber die Möglichkeit, Daten gemäß gewählter Strategien verschieben und replizieren zu können. Zwar kann dies auch über externe Dienste wie HTTP oder FTP erfolgen, von der Verfügbarkeit oder Verwendbarkeit solcher Dienste kann jedoch nicht immer ausgegangen werden. Daher wird der ZIB-DMS Server mit einer *Storage-Management*-Komponente ausgestattet. Diese speichert Daten in einer flachen Datei-Struktur mit kodierten Dateinamen lokal auf dem jeweiligen System. Zudem wird ein separater Katalog mit Zusatzinformationen über die abgelegten Daten geführt. Der Pfad, in dem diese Dateien abgelegt werden und wieviel Speicherplatz dem System hierfür zur Verfügung stehen, wird in der Konfiguration des Systems festgelegt. Der Zugriff auf diese Daten erfolgt durch die *CORBA-File-Access*-Komponente über das dafür entwickelte *CDTP*-Protokoll. Eine detaillierte Beschreibung dieses Protokolls und den zugehörigen Komponenten wird in [39] gegeben.

Notification

Diese Komponente stellt Mechanismen zur Protokollierung der Systemaktivität und Benachrichtigung zur Verfügung. Status- und Fehlermeldungen des Systems werden an einen Protokollierungsdienst gesendet. Dieser kann über die Funktionen der *ZIB-DMS API* dynamisch zur Laufzeit abgefragt und konfiguriert werden. Beim Auftreten von Ereignissen, die durch die *Monitoring*- oder *Time-Service*-Komponenten initiiert weden, können zudem Benachrichtigungen verschickt werden. Derzeit werden Benachrichtigungen über Email und in einer experimentellen Version über das *Jabber*-Messaging-Protokoll [54] unterstützt.

Monitoring

Zur Überwachung des Systems wird ein Signalisierungsmechanismus nach dem *Publish-Subscribe*- beziehungsweise *Observer*-Prinzip [23] bereitgestellt. Beliebige Komponenten können als sogenannte *Observer* für die Überwachung des Zustandes

einer anderen Komponente registriert werden. Eine Zustandsänderung der beobachteten Komponente wird den Beobachtenden dann signalisiert.

Time-Service
Die *Time-Service*-Komponente ermöglicht die geplante und zeitabhängige Ausführung von Aktionen innerhalb des Systems. Sie wird von vielen anderen Komponenten zur Steuerung regelmäßig ablaufender oder terminlich gebundener Vorgänge genutzt.

4.7 Optimierungskomponenten

Die Optimierungs- und Selbstorganisations-Komponenten des Systems sind verantwortlich zur Erhaltung der Stabilität und Verbesserung der Leistungfähigkeit des Systems [58]. Dabei arbeiten diese Komponenten autonom ohne Eingriff des Benutzers. Anstatt das System als Ganzes optimieren zu wollen, ist jede Komponente für einen eng eingegrenzten Leistungsaspekt verantwortlich. Ein Ansatz zur Optimierung der Systemleistung ist das Anlegen und Platzieren zusätzlicher Replikate [51, 61]. *Replica-Placement* überwacht und optimiert die Systemleistung durch Bewertung und geschickter Platzierung von Replikaten. Dies kann gemäß verschiedener Replikationsstrategien erfolgen. *Replica-Selection* wählt bei einem Zugriff aus allen registrierten Replikaten das günstigste aus. Auch hier sind verschiedene Strategien denkbar. So kann die Auswahl beispielsweise anhand von Kenngrößen wie geografische Distanz, Netzwerklast oder Antwortzeit der Datenquelle getroffen werden. Welche Strategie hier verwendet wird, hängt auch davon ab, welche Replikationsstrategie in *Replica-Placement* verwendet wird. Die *Replica-Management*-Komponente regelt die Wahl der Strategie. Diese Optimierungs-Mechanismen und mögliche Strategien werden in [39] eingehend beschrieben.

4.8 Abgrenzung zu anderen Grid-Datenmanagement-Lösungen

Wie in Abschnitt 4.1 bereits erwähnt, liegt einer der Hauptunterschiede zu vergleichbaren Systemen, wie den in Kapitel 3 vorgestellten, in der Anwendung des P2P-Paradigmas beim Aufbau des Systems durch lose gekoppelte Komponenten.

Ein weiterer Unterschied liegt in der Flexibilität des Systems und der Integrierbarkeit mit Standard-Anwendungen. Die meisten Systeme verwenden entweder einen hierarchischen Namensraum zur Adressierung der Datenobjekte, oder bieten einen attributbasierten Zugriff auf diese. Eine Kombination beider Zugriffsmethoden, wie sie das ZIB-DMS bietet, ist selten möglich. Die Mechanismen zur Datenverwaltung mit den *Globus*-Diensten gestalten sich relativ rudimentär. Das ZIB-DMS verwendet bei der Metadaten-Verwaltung

beispielsweise ein offenes Datenschema, das es erlaubt, beliebig viele arbiträre Attribute zu setzen. Differenzierte Features wie diese müssen hier separat realisiert werden. Auch *Gfarm* bietet keine eigenen Mechanismen zur Verwaltung von benuterdefinierten Metadaten, sondern verwendet nur vom System festgelegte Verwaltungsattribute. Der *SRB* bietet zwar die Möglichkeit Datenobjekte mit benutzerdefinierten Attributen zu versehen, diese sind allerdings auf eine fixe Anzahl und ein festes Schema beschränkt. Generell hat sich der *SRB* im Praxiseinsatz als relativ unflexibel und problematisch bei der Integration mit Standard-Software herausgestellt. Das Nachfolge-System *iRODS* ermöglicht die Verwaltung beliebiger Metainformationen, die aber hauptsächlich über proprietäre Kommandozeilen-Befehle verwaltet werden. Möglichkeiten zur transparenten Einbindung und einfachen Integration mit Standard-Software in dem Maße, wie sie das ZIB-DMS bietet, stehen hier aber nicht zur Verfügung.

Gfarm und *XtreemFS* sind als verteilte Dateisysteme konzipiert. Hier steht das konsistente Speichern und der effiziente Zugriff auf den Inhalt der Dateien im Vordergrund. Das ZIB-DMS hingegen ist als reines Datenmanagement-System konzipiert, das nur Metadaten und Quellangaben zu Replikaten, nicht aber die Daten selbst speichert. Dies muss über externe Mechanismen und Protokolle erfolgen, wobei das ZIB-DMS bemüht ist einen transparenten Zugriff auf diese Ressourcen durch Bereitstellung entsprechender File-Access-Komponenten zu gewährleisten.

Das Konzept der eigenständigen Selbstorganisations-Komponenten bildet die Grundlage für eine Anbindung des ZIB-DMS an die Job-Platzierungsmechanismen eines Grid-Systems, um die Platzierung von Replikaten, aber auch von Rechenjobs, zu beeinflußen und zu steuern (siehe auch Abschnitt 3.1 in Kapitel 3). Derartige Monitoring-Instrumente auf DMS Ebene fehlen sowohl bei *Globus*, *SRB* und *iRODS*, als auch im *XtreemFS*. Bei *Gfarm* hingegen ist dies ein zentraler Aspekt, wobei eine eigene Scheduler-Komponente verwendet wird.

5 Implementierung von Userspace-Dateisystemen mit FUSE

FUSE ist ein Rahmenwerk zur Implementierung von Dateisystemen im Userspace. Es entstand 2001 als Teil des AVFS-Paketes [71], wurde später aber von Autor M. Szeredi [72] als eigenständiges OpenSource-Projekt ausgegliedert. Ursprünglich für die Linux-Platform entwickelt, existieren inzwischen auch Portierungen für FreeBSD, MacOS X, NetBSD und OpenSolaris. Zur Nutzung der FUSE-Schnittstelle sind neben der nativen C-Bibliothek auch Anbindungen an viele andere Programmiersprachen verfügbar – unter anderem C++, Java, C#, Python, Perl und Ruby.

Allen folgenden Beschreibungen wird die zum Zeitpunkt der Entstehung dieser Arbeit aktuelle FUSE-Version 2.7.0 zu Grunde gelegt. Aus Gründen der Übersichtlichkeit wird in dieser Arbeit ausschließlich die Linux-Variante in Verbindung mit der C-Schnittstelle betrachtet. Die Ausführungen treffen zwar (teilweise) auch auf andere Plattformen oder Programmiersprachen zu; Aufgrund der engen Bindung an betriebssystemspezifische Details gibt es aber viele Abweichungen, die zu erörtern den Rahmen der Arbeit sprengen würde. Leider ist zu diesem Zeitpunkt nur spärlich Dokumentation über die Interna des Rahmenwerks verfügbar, so dass die in diesem Kapitel präsentierte Darstellung der internen Mechanismen durch ein relativ aufwändiges Studium des FUSE-Quelltextes erarbeitet werden musste.

Bevor das Rahmenwerk in den Abschnitten 5.3 und 5.4 detailliert vorgestellt wird, soll in Abschnitt 5.1 der Begriff *Userspace* genauer abgegrenzt werden. Außerdem gibt Abschitt 5.2 einen groben Überblick über die wichtigsten Aspekte bei der Implementierung von Dateisystemen.

5.1 Begriffsklärung: Userspace

Die Mehrheit moderner Prozessoren unterstützt mindestens zwei verschiedene Betriebsmodi. Die Ausführung bestimmter Operationen sowie der Zugriff auf bestimmte systemkritische Ressourcen sind nur im privilegierteren Modus erlaubt [68]. Dieser Mechanismus unterstützt den in Mehrbenutzersystemen notwendigen Schutz von Ressourcen.

Typischerweise werden die zentralen Funktionen des Betriebssystems in diesem privilegierten Modus, dem sogenannten *Kernel-Mode*[1] ausgeführt. Wohingegen gewöhnliche Benutzerprogramme im *User-Mode* und damit nicht-privilegiert ablaufen [74].

Der Begriff *Userspace* bezeichnet den Teil des Speichers, der zur Ausführung von User-Mode Programmen verwendet wird. Im Gegensatz dazu umfasst der *Kernelspace* denjenigen Teil des Speichers, in dem das Betriebssystem ausgeführt wird und seine Dienste zur Verfügung stellt. Man spricht daher auch von User- beziehungsweise Kernelspace Programmen.

Ein Dateisystem, auf das über die Schnittstelle des Betriebssystems zugegriffen wird, dessen Daten und Metadaten aber von einem nicht-privilegierten Prozess im Userspace bereitgestellt werden, nennt man *Userspace Dateisystem* [72]. Mehr hierzu in Abschnitt 5.2.2.

5.2 Implementierung von Dateisystemen

Die meisten Dateisysteme werden, wie in Kapitel 2 bereits erwähnt, für blockorientierte Geräte entwickelt. Der wohl wichtigste Aspekt bei der Implementierung von Dateisystemen ist daher die Zuordnung von Datenblöcken zu Dateien [74]. Hierzu gibt es verschiedene Möglichkeiten. Neben der Verwendung von Datei-Zuordnungstabellen ist die Verwendung von Index-Datenblöcken, sogenannten *inodes*, weit verbreitet.

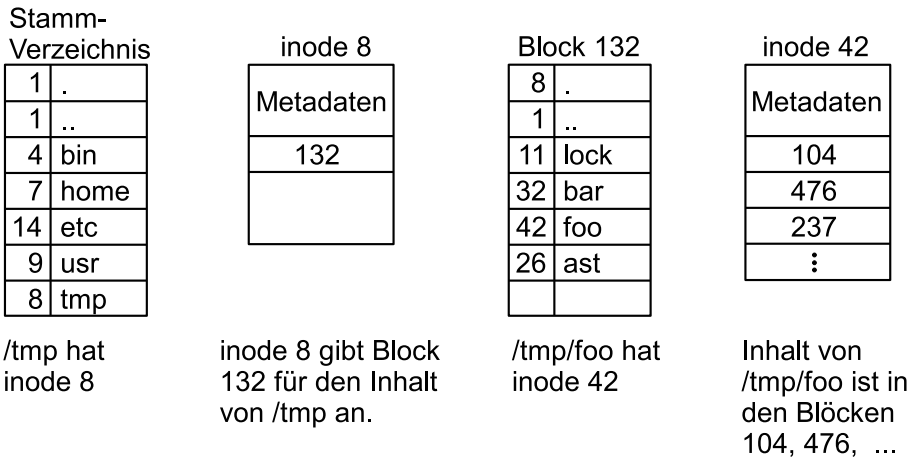

Abbildung 5.1: Schritte beim Zugriff auf */tmp/foo*.

Dabei wird jeder Datei ein Index-Block mit einer eindeutigen ID zugeordnet. Hier werden Metadaten und alle der Datei zugeordneten Datenblöcke festgehalten. Der Zugriff auf

[1] Die zentralen Funktionen eines Betriebssystems, also der Betriebssystem-Kern, werden oft als *Kernel* bezeichnet.

eine Datei bedingt das Auffinden und Lesen des jeweiligen inode-Blocks. Ein Verzeichnis wird in diesem Schema meist als spezielle Datei betrachtet. Der Unterschied zu einer regulären Datei liegt nur im Inhalt der Datenblöcke. Diese enthalten die Verzeichnis-Einträge und die IDs der korrespondierenden inode-Blöcke. Abbildung 5.1 veranschaulicht dieses Schema an einem Beispiel.

Die Manipulation von Dateien und Verzeichnissen erfolgt über eine Sammlung von Systemaufrufen. Das Dateisystem stellt die Funktionlität dieser Operationen bereit. Umfang und Gestalt dieser Schnittstelle ist grundsätzlich Abhängig vom jeweiligen Betriebssystem. Es gibt aber Bestrebungen, hier Standards zu etablieren. Einer der bekanntesten und meist unterstützten ist der vom IEEE-Verband geschaffene POSIX-Standard [30]. Obwohl es sich hierbei um eine Festlegung von Systemaufrufen für Unix-ähnliche Systeme handelt, bieten auch andere Systeme – zumindest teilweise – POSIX-konforme Schnittstellen [74]. Eine Übersicht über die wichtigsten POSIX-Systemaufrufe für Dateien und Verzeichnisse gibt Tabelle 5.1.

Operation	**Datei**	**Verzeichnis**	**Symbolischer Link**
Anlegen	`creat()`	`mkdir()`	`symlink()`
Löschen	`unlink()`	`rmdir()`	`unlink()`
Öffnen	`open()`	`opendir()`	—
Schließen	`close()`	`closedir()`	—
Lesen	`read()`	`readdir()`	`readlink()`
Durchuchen	`lseek()`	`seekdir()`	—
Schreiben	`write()`	—	—
Größe ändern	`truncate()`	—	—
Attribute lesen	`stat()`		`lstat()`
Attribute ändern	`chmod(), chown(), utime()`		
Umbenennen	`rename()`		

Tabelle 5.1: Zentrale POSIX-Dateisystem-Operationen [30].

5.2.1 Traditioneller Ansatz

Traditionell werden Dateisysteme als Teil des Betriebssystems aufgefasst und implementiert. Dies resultiert aus dem Verständnis des Betriebssystems als Mittel zur Ressourcen-Verwaltung [68]. Frühe Systeme realisierten die wenigen unterstützten Dateisysteme als integrale Bestandteile des Betriebssystems. Die Operationen des Dateisystems wurden direkt als Systemaufrufe implementiert. Im Laufe der Zeit wurde Erweiterbarkeit und Interoperabilität immer wichtiger, wodurch auch die Unterstützung einer Vielzahl von verschiedenen Dateisystemen notwendig wurde. Dies gestaltet sich mit dem erwähnten Vorgehen allerdings schwierig und unflexibel. Abhilfe schafft die Etablierung einer

Abstraktionsschicht, die den Zugriff auf Dateisysteme vereinheitlicht und implementierungsspezifsche Details ausblendet – allgemein als VFS (*Virtual Filesystem Switch*) bezeichnet [7, 9]. Viele moderne Betriebssysteme, darunter Unix und dessen Derivate, definieren solch eine Abstraktionsschicht. Durch die Einführung von Datenstrukturen und den zugehörigen Operationen schafft die VFS-Schnittstelle eine Trennung zwischen generischen Dateisystem-Operationen und solchen, die spezifisch für ein Dateisystem sind [64]. Die verwendeten Datenstrukturen orientieren sich dabei oft an dem oben beschriebenen *inode*-Konzept.

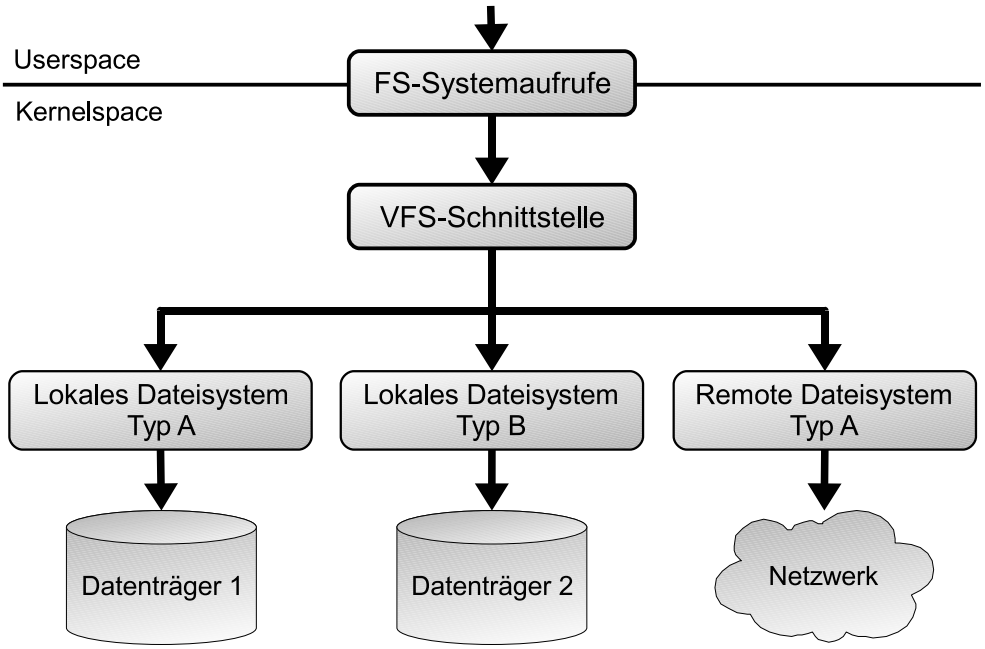

Abbildung 5.2: Aufbau einer Dateisystemarchitektur mit VFS-Schnittstelle.

Abbildung 5.2 zeigt die drei Schichten einer solchen Architektur. Die erste Schicht stellen die Systemaufrufe zur Verwaltung von Dateien dar. Die zweite Schicht ist die VFS-Schnittstelle. Hier werden Dateisystem-Anfragen entegegengenommen und an die jeweils zuständigen Dateisystem-Implementation der dritten Schicht weitergeleitet. Damit ist es möglich mehrere Instanzen verschiedener Dateisysteme transparent zu nutzen. Auch die Erweiterbarkeit eines Systems wird verbessert. Soll ein Betriebssystem um die Unterstützung für ein neues Dateisystem erweitert werden, muss nichts weiter unternommen werden, als eine gegen die VFS-Schnittstelle programmierte Implementation des Dateisystems in den Betriebssystemkern zu integrieren. Die Möglichkeit vieler moderner Systeme, Komponenten in Module auszulagern, die zur Laufzeit (nach)geladen werden können, stellt hierbei eine weitere Erleichterung dar.

5.2.2 Userspace Dateisysteme

Die oben erwähnte Modularisierung vieler moderner Betriebssysteme erleichtert zwar die Integration neuer Dateisysteme, bedingt aber eine Anpassung des Betriebssystems. Dies erfordert einerseits detailliertes Wissen über Systemspezifika und die Verfügbarkeit offener Schnittstellen, andererseits setzt die Installation und Integration des neuen Dateisystems einen voll-privilegierten Zugriff auf das betreffende System voraus. In vielen Fällen ist durch das Fehlen einer dieser Voraussetzungen eine Erweiterung des Betriebsystems nicht möglich oder nur mit hohem Aufwand durchfürbar.

Abhilfe schafft hier die Idee, die Implementation von Dateisystemen in den Userspace zu verlagern. Diese entstammt dem Gestaltungs-Ansatz, den Betriebssystemkern so kompakt wie möglich zu halten. Für gewöhnlich implementiert dieser die gesamte Funktionalität des Betriebssystems in einer monolithischen beziehungsweise schichten-basierten Struktur. Dagegen enthält ein *Microkernel* nur essentielle Funktionen, wie minimales Prozess- und Speichermanagement und Mechanismen zur Interprozesskommunikation [64]. Die restlichen System-Komponenten werden, wie in Abbildung 5.3 dargestellt, als Usermode-Prozesse nach dem Client-Server Modell realisiert. Dateisysteme werden hierbei als Server-Prozesse implementiert. Deren Nutzung erfolgt durch den Austausch von Nachrichten über die Kommunikationswege, die der Microkernel bereit stellt.

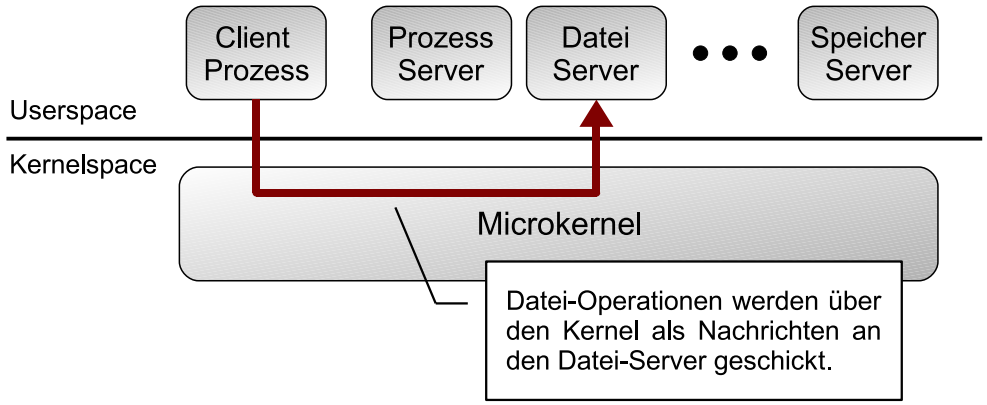

Abbildung 5.3: Client-Server Modell der Microkernel Architektur.

Dieses Schema lässt sich auch auf Betriebssysteme anwenden, die nicht nach der Microkernel-Architektur aufgebaut sind. Durch die einmalige Erweiterung des Systems um eine Komponente, die im wesentlichen den Export der VFS-Schnittstelle in den Userspace und die entsprechenden Kommunikationswege realisiert, werden die oben erwähnten Einschränkungen weitgehend aufgehoben. Da Dateisysteme nun als gewöhnliche Usermode-Programme implementiert werden können, ist keine Anpassung des Betriebssystems mehr erforderlich. Ebenso ist bei Installtion und Einbindung des Dateisystems ein voll-privilegierter Zugriff auf das System nicht mehr zwingend.

5.3 Aufbau

Das FUSE-Rahmenwerk stellt solch eine Betriebssystem-Erweiterung für die Implementierung von Userspace-Dateisystemen dar. Ein FUSE-Dateisystem besteht grundsätzlich aus drei Komponenten:

- Kernelmodul
- Programmbibliothek
- Userspace-Programm

Die beiden erstgenannten Komponenten werden vom Rahmenwerk zur Verfügung gestellt. Die eigentliche Funktionalität des Dateisystems wird durch ein Userspace-Programm realisiert, das gegen die Schnittstelle der FUSE-Bibliothek programmiert wird.

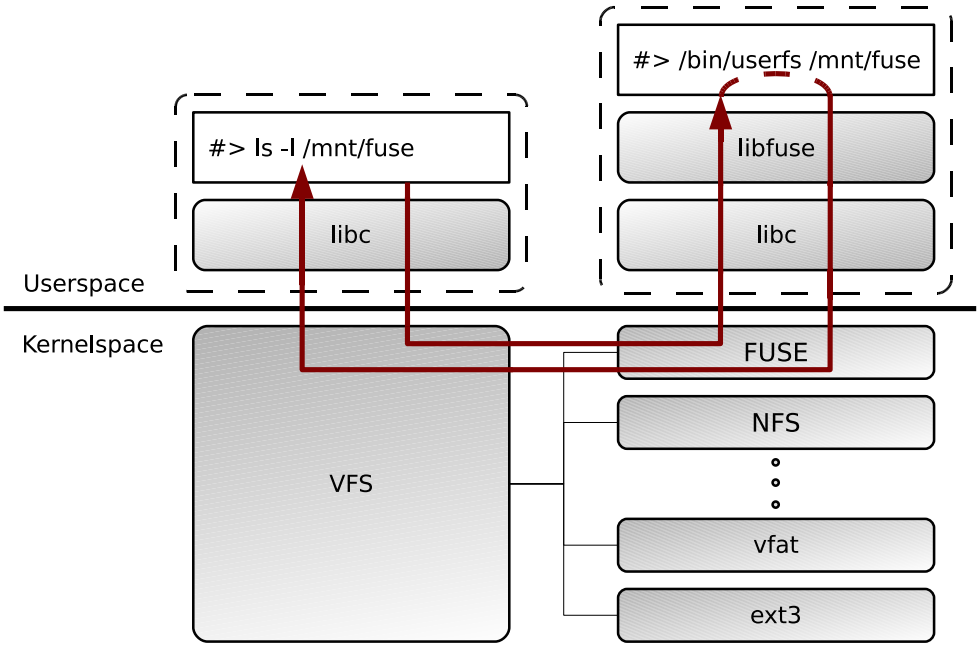

Abbildung 5.4: Aufbau und Kommunikationswege eines FUSE-Dateisystems.

Abbildung 5.4 gibt einen grundlegenden Überblick, wie sich die Kommunikation zwischen der Dateisystem-Schnittstelle des Betriebssystems und dem Userspace-Programm gestaltet. Anfragen an ein beispielsweise unter */mnt/fuse* eingebundenes FUSE-Dateisystem werden zunächst über die VFS-Schicht des Betriebssystems an das hier zuständige FUSE-Kernelmodul gerichtet. Dieses reicht die Anfrage über eine Device-Datei und den Callback-Mechanismus der FUSE-Bibliothek an das Userspace-Programm weiter. Hier wird die Anfrage bearbeitet und die entsprechenden Operationen werden ausgeführt. Die Ergebnisse der Operation gelangen auf gleichem Wege zurück.

5.3.1 Kernelmodul

Das FUSE-Kernelmodul besteht aus drei Teilen. Neben der Implementation der VFS-Schnittstelle steht vor allem die Gerätedatei `/dev/fuse` zur Kommunikation mit dem Userspace im Mittelpunkt. Zudem wird ein Steuerungs-Dateisystem zur Verfügung gestellt, um zur Laufzeit Informationen abfragen und Eingriffe vornehmen zu können.

Das Kernelmodul arbeitet prinzipiell mit den Linux-Versionen der Reihe 2.4[2] und 2.6 zusammen. Seit der Linux-Version 2.6.14 ist es Bestandteil des offiziellen Kernels. Die Unterstützung für Versionen der Reihe 2.4 wurde mit der FUSE-Version 2.6.0 eingestellt. Das Kernelmodul der hier verwendeten aktuellen FUSE-Version 2.7.0 erfordert eine Linux-Version von 2.6.9 oder höher. Zur Verwendung älterer Linux-Versionen ist es möglich, die aktuelle Version der FUSE-Bibliothek mit einem Kernelmodul der FUSE-Versionen vor 2.6.0 zu verwenden.

Gerätedatei

Die Kommunikation zwischen Kernel- und Userspace erfolgt über eine zeichenorientierte Gerätedatei (*Character-Device*) gemäß eines minimalen Protokolls. Die dabei verwendete Protokoll-Dateneinheit `fuse_req` repräsentiert eine einzelne Dateisystemanfrage. Sie enthält alle zur Abarbeitung notwendigen Daten sowie das Ergebnis der Anfrage. Eine Anfrage kann sich in einem von sechs Zuständen befinden:

`INIT` — Die Anfrage wird erstellt und ist noch nicht bereit zur Abarbeitung.
`PENDING` — Die Anfrage ist neu und bereit zur Abarbeitung.
`READING` — Die Anfrage wird in den Userspace übertragen.
`SENT` — Die Anfrage wurde vollständig gelesen und wird bearbeitet.
`WRITING` — Ergebnisse der Anfrage werden vom Userspace übertragen.
`FINISHED` — Die Bearbeitung der Anfrage ist abgeschlossen.

Ein eingebundenes FUSE-Dateisystem wird auch als Dateisystem-Verbindung bezeichnet. Der Gerätetreiber verwaltet eine Liste aktiver Dateisystem-Verbindungen. Eine Verbindung wird durch die `fuse_conn`-Datenstruktur repräsentiert. Diese enthält Dateisystem-Parameter, Daten zur Identifikation unterschiedlicher Dateisystem-Verbindungen und verschiedene Warteschlangen. Einige dieser Warteschlangen dienen zur Prozess-Synchronisation. Andere werden von den Dateisystemanfragen während ihres Lebenszyklus, abhängig vom jeweiligen Zustand, durchlaufen. Abbildung 5.5 zeigt den Lebenszyklus einer Dateisystemanfrage, die dem jeweiligen Zustand entsprechende Warteschlange und die den Zustandsübergang auslösenden Ereignisse.

[2] ab Version 2.4.21.

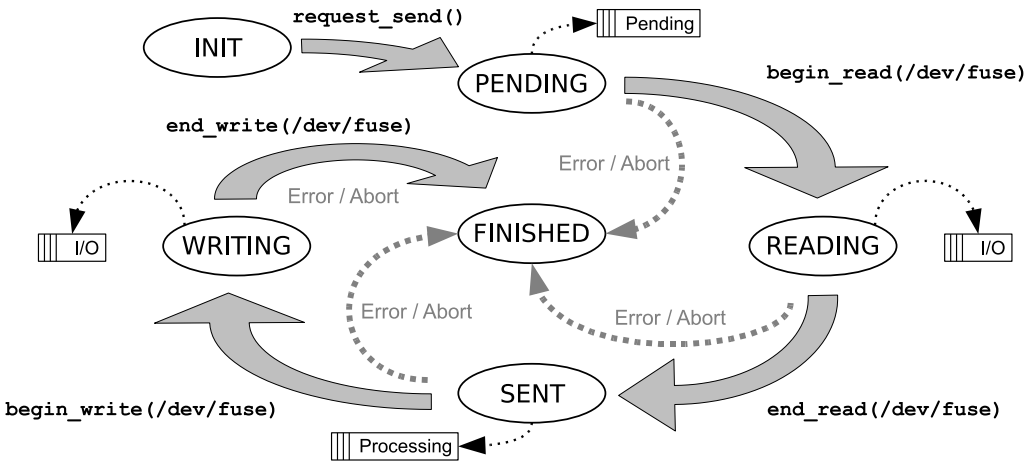

Abbildung 5.5: Lebenszyklus einer FUSE Dateisystemanfrage.

Durch Aufruf der Funktion `request_send()` auf Kernelseite, wird die dabei übergebene Dateisystemanfrage im Zustand `INIT` an die Warteschlange anstehender Anfragen (*Pending*) angehängt und die Anfrage geht in den Zustand `PENDING` über. Das Userspace-Programm nimmt wartende Anfragen durch Lesen von der Gerätedatei entgegen. Sobald die Leseoperation für eine Anfrage beginnt, wird diese in die Warteschlange *I/O* verschoben und wechselt in den Zustand `READING`. Falls die gesmate Anfrage fehlerfrei in den Userspace gelesen werden konnte, wechselt sie in den Zustand `SENT` und wandert in die *Processing*-Warteschlange. Nachdem das Userspace-Programm die Anfrage bearbeitet hat, überträgt es die Ergebnisse durch Schreiben in die Gerätedatei zurück in den Kernelspace. Mit Beginn der Schreiboperation wird die Anfrage erneut in die *I/O*-Warteschlange verschoben und in den Zustand `WRITING` versetzt. Sobald der Schreibvorgang abgeschlossen ist, wird die Anfrage aus der Warteschlange entfernt und ihr Zustand auf `FINISHED` gesetzt. Tritt während einem der genannten Vorgänge ein Fehler auf, oder wird die Anfrage zwischenzeitlich vom System oder vom Benutzer abgebrochen, wird die Anfrage direkt in den Zustand `FINISHED` überführt. Um Anfragen jederzeit unterbrechen zu können, wird für jede Dateisystem-Verbindung eine *Interrupt*-Warteschlange verwaltet. Interrupts werden immer einer Anfrage zugeordnet und haben Vorrang vor deren Abarbeitung.

Die Funktion `request_send()` steht in mehreren Ausprägungen zur Verfügung. Die einfache Variante startet die Bearbeitung der Anfrage, blockiert bis diese den Zustand `FINISHED` erreicht hat und kehrt mit deren Ergebnis zurück. Daneben wird eine ergebnislose und eine nicht-blockierende Version bereitgestellt. Bei Verwendung letzterer kann in der `fuse_req`-Struktur eine Callback-Funktion registriert werden, um Ergebnisse der Anfrage verarbeiten zu können.

VFS-Funktionen

Die Implementation der VFS-Funktionen stellt die Schnittstelle zum Linux-Kernel dar. Die VFS-Systemaufrufe werden hier in FUSE-Nachrichten (`fuse_req`) übersetzt und an den Gerätetreiber weitergeleitet. Größtenteils wird hierzu die blockierende Variante von `request_send()` verwendet. Die nicht-blockierende Variante wird bei asynchronen Lesevorgängen verwendet. Die `FORGET`-Nachricht wird als einzige mit der ergebnislose Variante versendet. Sie wird genutzt, um dem Userspace-Programm zu signalisieren, dass ein VFS-Objekt aus dem Kernel-Cache entfernt wird. Eine detaillierte Beschreibung der Linux-VFS-Schnittstelle würde den Rahmen an dieser Stelle sprengen. Hierzu wird auf [7] verwiesen.

Steuerungs-Dateisystem

Das Kernelmodul stellt ein virtuelles Pseudo-Dateisystem (siehe Abschnitt 2.2.4) *fusectl* bereit. Es wird üblicherweise innerhalb des *sysfs*-Baumes [36] eingebunden. Innerhalb dieses Steuerungs-Dateisystems wird für jede aktive FUSE-Verbindung ein Verzeichnis angezeigt, dessen Name eine eindeutige Zahl ist. Für jede Dateisystem-Verbindung existieren unterhalb dieses Verzeichnisses zwei Dateien:

`waiting`
: Durch Auslesen dieser Datei erhält man die Anzahl abzuarbeitender Dateisystem-Anfragen. Dies umfasst sowohl Anfragen, die auf die Übertragung in den Userspace warten, als auch solche, die bereits übertragen wurden und vom Dateisystem bearbeitet werden. Ist das Dateisystem inaktiv und der Inhalt dieser Datei größer als Null, ist dies ein Indikator für einen Absturz oder Deadlock des Userspace-Dateisystems.

`abort`
: Durch Schreiben beliebiger Zeichen in diese Datei, wird die entsprechende Dateisystem-Verbindung abgebrochen. Das bedeutet, dass alle wartenden sowie eventuelle neue Anfragen mit einer Fehlermeldung abgebrochen werden. Dies bietet die Möglichkeit, abgestürzte oder verklemmte Dateisysteme sauber abzubrechen.

Der Lese- beziehungsweise Schreibzugriff auf diese Dateien ist beschränkt auf den Nutzer, der das Dateisystem eingebunden hat.

5.3.2 Programmbibliothek

Die Funktionalität des FUSE-Rahmenwerks wird auf Userspace-Seite durch eine Programmbibliothek bereitgestellt. Neben der in Abschnitt 5.4 beschriebenen Programmierschnittstelle, stellt diese Funktionen zur Verarbeitung von Kommandozeilenparametern

sowie zur Einbindung eines FUSE-Dateisystems und Mechanismen zur Kommunikation mit dem FUSE-Kernelmodul bereit. Das Userspace-Programm, das die Funktionalität des Dateisystems implementiert, muss gegen diese Bibliothek gelinkt werden. Durch Aufruf dieses Programmes kann das FUSE-Dateisystem verwendet werden:

```
/usr/bin/dateisystemX /mnt
```

Dabei wird der Einhängepunkt (*Mountpoint*), also das Verzeichnis unter dem das Dateisystem erreichbar sein soll, als Parameter übergeben.

Verarbeitung von Kommandozeilenparametern

Beim Auruf eines FUSE-Dateisystems können neben dem Mountpoint verschiedene Kommandozeilenparameter übergeben werden. Diese Parameter lassen sich in drei Kategorien unterteilen: Standard-Bibliotheksoptionen, Mount-Optionen und dateisystemspezifische Optionen. Die Bibliothek unterstützt einen Satz von allgemeinen Optionen, die das Verhalten der Bibliothek steuern und Parameter für das Einbinden eines Dateisystems. Diese können für jedes FUSE-Dateisystem verwendet werden.

Für die Verarbeitung dieser ersten beiden Arten von Parametern bietet die Bibliothek die Funktion `parse_common_options()`, die nur die vorgegebenen FUSE-Optionen verarbeitet, den Rest des übergebenen Argumentvektor aber nicht beachtet. Dateisystemspezifische Optionen steuern das Verhalten des jeweiligen Dateisystems. Die Behandlung dieser Optionen kann manuell durch einen eigens implementierten Mechanismus erfolgen. Die FUSE-Bibliothek bietet aber auch Hilfsfunktionen für die Registrierung und Verarbeitung von programmspezifischischen Kommandozeilenparametern. Die Verarbeitung erfolgt durch Aufruf der Funktion `fuse_opt_parse()`, deren Signatur Listing 5.1 zu entnehmen ist.

```
int fuse_opt_parse ( struct fuse_args *args,          // Argumentvektor
                     void *data,                      // Benutzerdaten
                     const struct fuse_opt opts[],    // Optionsbeschreibungen
                     fuse_opt_proc_t proc);           // Verarbeitungsfunktion
```

Listing 5.1: `fuse_opt_parse()` Signatur.

Diese durchsucht den übergebenen Argumentvektor nach Vorkommen der in den Optionsbeschreibungen festgelegten Kommandozeilenparametern. Auf die Entdeckung eines solchen Parameters kann auf zwei Arten reagiert werden. Entweder wird dessen Wert (oder ein Standardwert) in eine Variable geschrieben oder die übergebene Verarbeitungsfunktion wird gerufen und der Wert dabei übergeben. Dies wird durch entsprechende Belegung der Felder der Datenstruktur `fuse_opt`-gesteuert, die einen gültigen Kommandozeilenparameter repräsentiert.

Einbinden von FUSE-Dateisystemen

Eines der wichtigsten Merkmale von FUSE ist die Schaffung einer sicheren Möglichkeit, Dateisysteme als nicht-privilegierter Nutzer einbinden zu können. Wie fast alle Mechanismen des Rahmenwerks besteht auch das Einhängen aus jeweils einer Operation im Kernel- und Userspace.

```
struct fuse_chan *
    fuse_mount ( const char *mountpoint,       // Einhängepunkt
                 struct fuse_args *args );     // Argumentvektor
```

Listing 5.2: `fuse_mount()` Signatur.

Das Einbinden von Dateisystemen erfolgt auf Systemseite durch den Linux-Systemaufruf `mount()`. Dieser erfordert Superuser-Zugriffsrechte und kann daher nicht direkt gerufen werden. Die Lösung dieses Problems erfolgt durch Einsatz eines ebenfalls im Rahmenwerk enthaltenen Hilfsprogrammes *fusermount*. Bei der Installation wird dieses Programm mit der Benutzerkennung des Superusers (*root*) als Eigentümer und dem *setuid*-Bit versehen. Das bedeutet, dass das Programm bei Ausführung unter der Benutzerkennung *root* statt der des tatsächlichen Benutzers ausgeführt wird und damit dessen privilegierte Zugriffsrechte nutzen kann. Aus Gründen der Sicherheit kann *fusermount* zum Einbinden eines Dateisystems jedoch nicht direkt aufgerufen werden, sondern nur über die Bibliotheksfunktion `fuse_mount()`. Listing 5.2 zeigt die Signatur dieser Funktion. Beim Aufruf wird der Pfadname des Einhängepunktes und eine Liste mit optionalen Argumenten übergeben. Mit diesen Daten wird dann das *fusermount*-Hilfsprogramm als Kindprozess ausgeführt. Dieser lädt zunächst, falls notwendig, das FUSE-Kernelmodul. Anschließend wird die Gerätedatei `/dev/fuse` geöffnet und der `mount()`-Systemcall ausgeführt. Verläuft all dies fehlerfrei, erfolgt dadurch die Einbindung des Dateisystems auf Systemseite und der Dateideskriptor der geöffneten Gerätedatei wird vor Beendigung des Hilfsprogrammes an `fuse_mount()` zurück übertragen. Hier wird ein eindeutig identifizierbarer FUSE-Kommunikationskanal (`fuse_chan`), dessen wichtigster Bestandteil der erhaltene Dateideskriptor ist, erzeugt und zurückgegeben.

```
// Highlevel-API
struct fuse *
    fuse_new ( struct fuse_chan *ch,                  // Kommunikationskanal
               struct fuse_args *args,                // Argumentvektor
               const struct fuse_operations *op,      // Dateisystem Operationen
               size_t op_size,                        // Größe von op
               void *user_data );                     // Benutzerdaten

// Lowlevel-API
struct fuse_session *
    fuse_lowlevel_new ( struct fuse_args *args,              // Argumentvektor
                        const struct fuse_lowlevel_ops *op,  // Dateisystem Operationen
                        size_t op_size,                      // Größe von op
                        void *userdata);                     // Benutzerdaten
```

Listing 5.3: Signaturen der `fuse_new()` Funktionen.

Um das Dateisystem wirklich nutzen zu können, muss es auf Userspace-Seite noch initialisiert und angemeldet werden. Dies erfolgt durch die Funktion `fuse_new()` bei Verwendung der *Highlevel-API*, beziehungsweise `fuse_lowlevel_new()` bei Verwendung der *Lowlevel-API*. Die Signaturen dieser Funktionen sind in Listing 5.3 abgebildet. Der wichtigste Funktionsparameter ist hier die Datenstruktur `fuse_operations` beziehungsweise `fuse_lowlevel_ops`, die Zeiger auf die implementierten Dateisystemfunktionen enthält. Der Argumentvektor enthält Kommandozeilenparameter. Der `user_data`-Zeiger dient zur transparenten Übergabe von Datenstrukturen, die dann innerhalb des Dateisystems zur Verfügung stehen. Bei Verwendung der *Highlevel-API* verwaltet FUSE ein zwischengespeichertes Abbild des Dateisystems innerhalb der Datenstruktur `fuse`. Diese wird durch `fuse_new()` initialisiert und schließlich zurückgegeben. Dabei wird der von `fuse_mount()` erhaltene Kommunikationskanal direkt übergeben. Bei Verwendung der *Lowlevel-API* muss dieser durch Aufruf der Funktion `fuse_session_add_channel()` an die FUSE-Session gebunden werden, die durch den Aufruf von `fuse_lowlevel_new()` initialisiert wird. Auf die Unterschiede zwischen *Lowlevel-* und *Highlevel-API* wird in Abschnitt 5.4 näher eingegangen. Nach Aufruf einer dieser Funktionen ist die Verbindung zum Kernelmodul vollständig etabliert und das Dateisystem kann beginnen, Anfragen zu bedienen.

Kommunikation mit dem Kernelmodul

Die Kommunikation mit dem Kernelspace erfolgt, wie oben beschrieben, durch den Austausch von Protokollnachrichten über die FUSE-Gerätedatei. Dateisystemanfragen vom Kernelmodul müssen von dieser Gerätedatei gelesen, verarbeitet und Ergebnisse anschließend zurück geschrieben werden. Dies wird von den in Listing 5.4 dargestellten verschiedenen Varianten der Funktion `fuse_loop()` erledigt.

```
// Highlevel-API
int  fuse_loop    ( struct fuse *f );    // Singlethreaded
int  fuse_loop_mt ( struct fuse *f );    // Multithreaded

// Lowlevel-API
int  fuse_session_loop    ( struct fuse_session *se );  // Singlethreaded
int  fuse_session_loop_mt ( struct fuse_session *se );  // Multithreaded
```

Listing 5.4: Signaturen der `fuse_loop()` Funktionen.

Wie der Name vermuten lässt, prüfen diese Funktionen in einer interrupt-gesteuerten Endlosschleife, ob neue FUSE-Protokollnachrichten für den entsprechenden Kommunikationskanal vorliegen. Beim Eintreffen neuer Nachrichten werden diese dekodiert und die entsprechenden, bei der Initialisierung des Dateisystems angegebenen Dateisystem-Operationen, aufgerufen. Die Rückgabewerte der Operationen werden dann wieder, in Protokollnachrichten kodiert, in die Gerätedatei geschrieben. Die Schleifenfunktionen stehen grundsätzlich in zwei Varianten zur Verfügung. Eine einfache Variante, die Anfragen nacheinander jeweils einzeln bearbeitet (`fuse_loop()`) und eine Variante, die

mehrere Anfragen parallel bearbeitet (`fuse_loop_mt()`). Wie auch bei den Initialisierungsfunktionen existieren eigene Loop-Funktionen für die *Highlevel-* und *Lowlevel-API*, wobei hier der Unterschied nur im jeweiligen Übergabeparameter liegt. Das Verhalten der Funktionen ist identisch. Beendet werden diese Endlosschleifen durch den Aufruf der Funktion `fuse_exit()` beziehungsweise `fuse_session_exit()`.

5.4 Programmier-Schnittstelle

Das FUSE-Rahmenwerk stellt eine Programmierschnittstelle oder API (*Advanced Programmer Interface*) zur Verfügung, die im Wesentlichen aus den Funktionssignaturen der in Tabelle 5.2 aufgeführten Dateisystem-Operationen besteht. Die in Abschnitt 5.3.2 vorgestellten *fuse_session_loop*-Funktionen lesen Nachrichten von der FUSE-Gerätedatei und rufen die korrespondierenden Operationen. Die Hauptaufgabe bei der Implementierung eines FUSE-Dateisystems ist die Bereitstellung dieser Operationen gemäß der angegebenen Signaturen. Beim Aufruf der FUSE-Initialisierungsfunktion wird, wie oben beschrieben, eine Datenstruktur mit Zeigern auf diese Funktionen übergeben. Dabei ist es prinzipiell nicht erforderlich alle Dateisystem-Operationen mit Funktionen zu belegen. Wird einer Operation kein Funktionszeiger zugewiesen, wird bei deren Aufruf keine beziehungsweise eine Standard-Aktion ausgeführt und eventuell eine Meldung zur fehlenden Implementation dieser Operation ausgegeben. Dieses Verhalten stellt eine große Erleichterung bei der Implementierung dar, da hiermit die inkrementelle Entwicklung eines Dateisystems möglich wird. Für die Realisierung eines minimalen, wenn auch nicht sehr nützlichen FUSE-Dateisystems, ist die Bereitstellung von nur vier Operationen ausreichend: `getattr()`, `open()`, `readdir()` und `read()`. Ausgehend von dieser Basis können weitere Operationen nach und nach hinzugefügt werden.

Die Schnittstelle steht in zwei Ausprägungen zur Verfügung. Diese unterscheiden sich im Wesentlichen durch den jeweiligen Abstraktionsgrad. Die **Lowlevel-Schnittstelle** bietet mit ihren 34 Operationen eine direkte Abbildung des Nachrichtenaustauschs zwischen Kernel- und Userspace. Vom Kernelmodul an den Userspace gesendete Protokollnachrichten werden eins zu eins auf Operationen abgebildet. Dateisystemobjekte werden anhand von logischen Blockindex-Nummern (Inode) identifiziert. Deren Verwaltung liegt in der Verantwortung des Dateisystems selbst, ebenso wie die Abbildung von Inodes auf Pfadnamen und das Caching dieser Informationen. Jede der Funktionen erhält als Übergabeparameter die Inode-Nummer des angefragten Objekts und eine Referenz und damit direkten Zugriff auf die jeweilige Protokollnachricht. Die 37 Operationen der **Highlevel-Schnittstelle** hingegen orientieren sich an den bekannten POSIX Dateisystem-Operationen [30]. Diese sind mit Pfadnamen anstatt Inode-Nummern parametrisiert. Die übergebenen Pfadnamen sind absolut zur Wurzel des FUSE-Dateisystems; Das heißt, bei Anfragen nach einer Datei *foo* an ein unter */mnt* eingebundenes FUSE-Dateisystem wird nicht der Pfad */mnt/foo* sondern */foo* übergeben. Das Rahmenwerk übernimmt standardmäßig die Abbildung von Pfadnamen auf Inode-

Operation	Lowlevel API	Highlevel API	Beschreibung
`init`	√	√	Initialisierung des Dateisystems
`destroy`	√	√	Aufräumen des Dateisystems
`lookup`	√	¬	Abbilden von Dateinamen auf Inodes
`forget`	√	¬	Inode aus Cache entfernen
`mkdir`	√	√	Anlegen von Verzeichnissen
`rmdir`	√	√	Löschen von Verzeichnissen
`mknod`	√	√	Anlegen von Dateien
`create`	√	√	Anlegen und Öffnen von Dateien
`link`	√	√	Anlegen von Hardlinks
`symlink`	√	√	Anlegen von symbolischen Links
`readlink`	√	√	Auslesen von symbolischen Links
`unlink`	√	√	Löschen von Dateien und Links
`rename`	√	√	Umbennen/Verschieben von Dateisystemobjekten
`opendir`	√	√	Öffnen von Verzeichnissen
`readdir`	√	√	Lesen von Verzeichnissen
`releasedir`	√	√	Schließen von Verzeichnissen
`fsyncdir`	√	√	Synchronisieren von Verzeichnissen
`open`	√	√	Öffnen von Dateien
`read`	√	√	Lesen von Dateien
`write`	√	√	Schreiben von Dateien
`flush`	√	√	Cache entleeren
`fsync`	√	√	Dateiinhalt synchronisieren
`release`	√	√	Schließen von Dateien
`truncate`	¬	√	Ändern der Größe von Dateien
`ftruncate`	¬	√	Ändern der Größe geöffneter Dateien
`getattr`	√	√	Lesen von Attributen
`fgetattr`	¬	√	Lesen von Attributen geöffneter Dateien
`setattr`	√	¬	Schreiben von Attributen
`chmod`	¬	√	Ändern von Zugriffsrechten
`chown`	¬	√	Ändern von Eigentümer- und Gruppenkennung
`utime`	¬	√	Ändern der Zeitstempel für Zugriff und Modifikation
`utimens`	¬	√	Ändern der Zeitstempel mit Nanosekunden-Genauigkeit
`setxattr`	√	√	Schreiben von erweiterten Attributen
`getxattr`	√	√	Lesen von erweiterten Attributen
`listxattr`	√	√	Auflisten von erweiterten Attributen
`removexattr`	√	√	Löschen von erweiterten Attributen
`access`	√	√	Prüfen von Zugriffsrechten
`lock`	¬	√	Verwaltung von Dateisperren
`getlk`	√	¬	Auf Dateisperre prüfen
`setlk`	√	¬	Sperren einer Datei
`bmap`	√	√	Direkte Abbildung von logischen auf physikalische Blöcke
`statfs`	√	√	Auslesen der Dateisystemstatistik

Tabelle 5.2: Dateisystem-Operationen der FUSE API.

Nummern und deren Caching, es besteht aber auch hier Möglichkeit, Inode-Nummern dateisystemspezifisch zu verwalten. Allerdings ist kein direkter Zugriff auf die Protokollnachrichten möglich. Tabelle 5.2 zeigt die Operationen beider Schnittstellen. Für den Großteil existieren Entsprechungen für beide Varianten der API. Einige sind nur in der Highlevel-, wenige nur in der Lowlevel-Variante verfügbar. Der höhere Abstraktionsgrad der Highlevel-API wird bei den Operationen für das Setzen von Attributen besonders deutlich. Die hierfür verwendete Lowlevel-Funktion `setattr()` steht in der Highlevel-API nicht zur Verfügung. Stattdessen verteilt sich die Funktionalität auf Operationen `chmod()`, `chown()`, `utime()` und `utimens()`.

Antwortfunktion `fuse_` ...	Erlaubt für	Beschreibung
... `reply_none`	`forget`	Keine Antwort
... `reply_err`	Alle außer `forget`	Antwort mit Statuscode
... `reply_entry`	`lookup, mknod, mkdir, symlink, link`	Antwort mit Verzeichniseintrag
... `reply_readlink`	`readlink`	Antwort mit Ziel eines symbolischen Links
... `reply_open`	`opendir, open`	Antwort mit Open-Parametern
... `reply_create`	`create`	Antwort mit Verzeichniseintrag und Open-Parametern
... `reply_buf`	`read, readdir, getxattr, listxattr`	Antwort mit Datenpuffer
... `reply_iov`	`read, readdir, getxattr, listxattr`	Antwort mit Datenvektor
... `reply_write`	`write`	Antwort mit Anzahl geschriebener Bytes
... `reply_attr`	`getattr, setattr`	Antwort mit Dateiattributen
... `reply_xattr`	`getxattr, listxattr`	Antwort mit erforderlicher Puffergröße für erweiterte Attribute
... `reply_lock`	`getlk`	Antwort mit File-Lock Informationen
... `reply_bmap`	`bmap`	Antwort mit Blockindex
... `reply_statfs`	`statfs`	Antwort mit Dateisystemstatistik

Tabelle 5.3: Antwort-Funktionen der FUSE Lowlevel-API.

Während die Funktionen der Highlevel-Schnittstelle mit Rückgabewerten und In-/Out-Parametern arbeiten, sind die Lowlevel-Operationen ohne Rückgabewert definiert. Um Ergebnisse an den Kernel zurückzuliefern, muss daher innerhalb der Funktionen eine der 14 Lowlevel-Antwortfunktionen gerufen werden. Die Auswahl der Antwortfunktionen hängt dabei vom Datentyp des Ergebnisses ab. Tabelle 5.3 erläutert den Verwendungszweck der Antwortfunktionen und für welche Operationen sie gültig sind.

Beispiel: Listing 5.5 gibt ein Beispiel für die Implementation der `getattr()`-Operation unter Verwendung der Lowlevel-API. Dieses Code-Fragment wurde dem im FUSE-Paket [72] enthaltenen Beispiel-Programm *hello_ll.c* entnommen, das ein minimales Dateisystem im *Hallo Welt* Stil implementiert. In diesem unveränderlichen

Pseudodateisystem existieren nur das Wurzelverzeichnis und eine Datei *hello*, deren Inhalt die Zeichenkette "`Hello World\n`" ist.

```
static void hello_ll_getattr(fuse_req_t req,
                             fuse_ino_t ino,
                             struct fuse_file_info *fi)
{
    struct stat stbuf;
    memset(&stbuf, 0, sizeof(stbuf));
    stbuf->st_ino = ino;
    switch (ino) {
        case 1:
            stbuf->st_mode = S_IFDIR | 0755;
            stbuf->st_nlink = 2;
            fuse_reply_attr(req, &stbuf, 1.0);
            break;

        case 2:
            stbuf->st_mode = S_IFREG | 0444;
            stbuf->st_nlink = 1;
            stbuf->st_size = strlen("Hello World!\n");
            fuse_reply_attr(req, &stbuf, 1.0);
            break;

        default:
            fuse_reply_err(req, ENOENT);
    }
}
```

Listing 5.5: Beispiel einer Lowlevel-Operation.

Dem Verzeichnis wird *Inode 1* zugeordnet, der Datei *Inode 2*. Die Funktion erzeugt eine Datenstruktur `stat()` und setzt dort die Attribute entsprechend der übergebenen Inode. Diese wird dann mit der korrespondierenden Antwortfunktionen `fuse_reply_attr()` an das Kernelmodul gesendet. Wird eine andere Inode-Nummer als 1 oder 2 angefragt, wird eine Fehlermeldung verschickt, die angibt, dass die angeforderte Datei nicht existiert.

```
static int hello_getattr(const char *path, struct stat *stbuf)
{
    int res = 0;
    memset(stbuf, 0, sizeof(struct stat));
    if (strcmp(path, "/") == 0) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
    } else if (strcmp(path, "/hello") == 0) {
        stbuf->st_mode = S_IFREG | 0444;
        stbuf->st_nlink = 1;
        stbuf->st_size = strlen("Hello World\n");
    } else
        res = -ENOENT;

    return res;
}
```

Listing 5.6: Beispiel einer Highlevel-Operation.

Listing 5.6 zeigt die gleiche Operation des selben Dateisystems unter Verwendung der Highlevel-API. Wie man sieht, erfolgt die Unterscheidung zwischen Wurzel-

verzeichnis und Datei hier Anhand des Dateinamens. Die Datenstruktur `stat()` wird hier nicht lokal erstellt, sondern als In-/Out-Parameter übergeben. Nach Setzen der Attribute wird die Funktionen mit dem Rückgabewert *0* (Erfolg) beendet. Wird ein anderer Pfadname als */* oder */hello* angefragt, wird die Funktion mit dem negierten Fehlercode `ENOENT` (Datei nicht gefunden) sofort beendet.

Die Lowlevel-Schnittstelle leistet den Export der VFS-Ebene des Kernels in den Userspace, bietet im Gegensatz zur Highlevel-API darüberhinaus aber keine zusätzlichen Features, die die Implementierung von Dateisystemen weiter vereinfachen würden. Die Nutzung dieser sehr rudimentären Schnittstelle zur Implementierung von Dateisystemen ist daher nur für solche sinnvoll, die deren direkten Zugriff unbedingt erfordern. Meist ist die Verwendung der komfortableren Highlevel-API hierfür zu bevorzugen. Die intendierte Anwendung der Lowlevel-API ist die Schaffung von Anbindungen an andere Programmiersprachen und Schnittstellen mit höherem Abstraktionsgrad. So ist die Highlevel-API der FUSE-Bibliothek intern unter Verwendung der Lowlevel-API implementiert.

```
int fuse_main ( int argc,                              // Anzahl der Argumente
                char *argv[],                          // Argumentliste
                const struct fuse_operations *op,      // Dateisystem-Operationen
                void *user_data );                     // Benutzerdaten
```

Listing 5.7: `fuse_main()` Signatur

Die Highlevel-Schnittstelle bietet einen komfortableren Satz an Schnittstellendefinitionen und Hilfsfunktionen. Die Mehrheit der in [72] verzeichneten FUSE-Dateisysteme nutzt diese Schnittstelle. Desweiteren bietet sie einige sinnvolle Standard-Funktionen für nicht-implementierte Dateisystem-Operationen. Auch die Initialisierung eines FUSE-Dateisystems wird durch die Bereitstellung der Funktion `fuse_main()` vereinfacht. Listing 5.7 ihre Signatur. Beim Aufruf dieser Funktion werden alle in Abschnitt 5.3.2 beschriebenen Schritte zur Initialisierung vorgenommen, so dass es ausreicht, diese Funktion innerhalb der `main()`-Funktion des Userspace-Programms zu rufen.

Beispiel: Das in Listing 5.8 dargestellte Code-Fragment ist dem Beispielprogramm *fusexmp.c* des FUSE-Paketes entnommen, welches eine Art Loopback-Dateisystem implementiert. Es leitet die Anfragen an das FUSE-Dateisystem einfach an die entsprechenden Systemaufrufe weiter. Da die Pfade, die den FUSE-Operationen übergeben werden, absolut zur Wurzel des FUSE-Dateisystems sind, wird dadurch das Systemweite-Wurzelverzeichnis über den Einhängepunkt des FUSE-Dateisystems erreichbar. Listing 5.8 zeigt nur einen Ausschnitt der wichtigsten Elemente des Programms, um ein übersichtliches Beispiel für die Implementierung eines FUSE-Dateisystems zu geben. Der vollständige Quelltext ist in [72] zu finden.

```
static int xmp_getattr ( const char *path, struct stat *stbuf )
{
    int res;
    res = lstat(path, stbuf);
    if (res == -1)
        return -errno;
    return 0;
}

// ... Weitere Operationen ...

static struct fuse_operations xmp_oper = {
    .getattr    = xmp_getattr,
        .               .
        .               .
        .               .
};

int main(int argc, char *argv[])
{
    return fuse_main(argc, argv, &xmp_oper, NULL);
}
```

Listing 5.8: Beispiel für den Aufruf von `fuse_main()`.

Exemplarisch wird wieder die Implementierung der Funktion `getattr()` gezeigt. Hier wird der Linux-Systemaufruf `lstat()`, der Attribute einer Datei ausliest, gerufen. Da dieser wie `getattr()` als Parameter einen Pfadnamen und einen Zeiger auf eine `stat()`-Struktur erwartet, können diese einfach weitergereicht werden. Durch Rufen dieser Funktion wird die Anfrage an das für den Pfad zuständige Dateisystem weitergeleitet. Der Systemaufruf speichert Fehlercodes in der globalen Variable `errno`. Im Fehlerfall wird der negierte Fehlercode zurückgegeben; Andernfalls 0

Funktionszeiger auf diese und weitere Operationen werden dann den Feldern der globalen `fuse_operations`-Datenstruktur zugewiesen. Die `main()`-Funktion des Programms enthält nur eine Zeile: Den Aufruf der Funktion `fuse_main()`. Hier werden nur die Argumentliste und die Funktionszeiger übergeben. Alles weitere wird vom Rahmenwerk erledigt.

Dieses Beispiel illustriert, wie komfortabel sich die Arbeit mit dem FUSE-Rahmenwerk gestaltet. Mit wenigen Zeilen Code kann ein einfaches voll funktionsfähiges Loopback-Dateisystem implementiert werden.

Der Ansatz Dateisysteme im Userspace zu implementieren ermöglicht es, wie in diesem Kapitel beschrieben, neue Dateisysteme ohne Anpassung am Betriebssystem zu schaffen und diese ohne einen voll-privilegierten Zugriff zu erzeugen und und einzubinden. Mit dem FUSE-Rahmenwerk steht ein durchdachtes, komfortables und einfach zu benutzendes Werkzeug für die Realisierung solcher Userspace-Dateisysteme zur Verfügung.

6 Ein FUSE-Dateisystem mit Anbindung an das ZIB-DMS

Ein wesentlicher Bestandteil dieser Arbeit ist die Entwicklung und Implementierung einer FUSE-Dateisystem-Schnittstelle für das ZIB-DMS. Dieses Kapitel beschreibt und dokumentiert den praktischen Teil der Arbeit. Zunächst wird die Situation und der Stand der Implementation zu Beginn der Arbeit betrachtet. Anschließend werden die vorgenommenen notwendigen Änderungen am System im Allgemeinen und an der Directory-View-Komponente im Speziellen erörtert. Hier wird auch eingehend beschrieben, wie die erweiterten Verwaltungs-Konzepte des Systems für hierarchische Dateisysteme umgesetzt werden. Es werden zwei Varianten der FUSE-Schnittstelle entwickelt, deren Implementierung und Anbindung an das System, sowie die Unterschiede der beiden Varianten abschließend beschrieben werden.

6.1 Ausgangssituation

Die Software-Architektur des ZIB-DMS (siehe Abbildung 4.2 auf Seite 25) und dessen geplante Komponenten wurden bereits in Kapitel 4 vorgestellt. Als Basis dieser Arbeit liegt eine grundlegend funktionsfähige, aber unvollständige Implementation dieses Systems vor. Diese besteht im Wesentlichen aus einem Server-Programm und einem separaten Client-Programm in Form einer grafischen Schnittstelle (*GUI*). Für den Metadaten-Katalog (siehe Abbildung 4.7 auf Seite 31) stehen Anbindungen an zwei verschiedene SQL-Datenbanksysteme, *sqlite* und *MySQL*, zur Verfügung. Die Anbindung des *MDC* an ein Overlay-Netzwerk ist noch nicht vollständig umgesetzt und daher nicht funktionsfähig, ebenso wie die Anbindung an den Lucene-Indexer. Die Service-Komponenten *File-Access*, *File-Transfer*, *Storage-Management*, *Assimilation*, *Time-Service*, *Monitoring* und *Notification* sind implementiert. Ebenso einige Optimierungs-Komponenten, wie *Replica-Location*, *Replica-Selection* und *Replica-Placement*. Die Benutzer- und Rechteverwaltung des Systems befindet sich noch in Planung und steht nur in einer rudimentären Test-Implementation zur Verfügung. Es können zwar Benutzer und Gruppen sowie Zugriffsrechte auf Objekte im Medataten-Katalog angelegt und verwaltet werden, die Umsetzung der damit einhergehenden Sicherheitsmechanismen ist aber noch nicht vollständig realisiert. Daher erfolgt der Zugriff auf das ZIB-DMS in der vorliegenden Implementation ohne Beschränkung. Dieser Zugriff erfolgt über eine der sogenann-

ten *Frontend*-Komponenten. Hierbei liegen die *Python*- und *SOAP*-Varianten ebenfalls nur als rudimentäre Test-Implementationen vor. In vollem Umfang sind dagegen die native C++-Variante und die *CORBA*-Anbindung der *ZIB-DMS API* und die NFS-Komponente implementiert.

Die NFS-Server-Komponente stellt den ersten Ansatz zur Schaffung eines völlig transparenten Zugriffs auf den Datenbestand des ZIB-DMS durch Einbindung in einen gewöhnlichen Verzeichnisbaum dar. Diese Form des Zugriffs und die damit einhergehende Abbildung der Funktionalität des ZIB-DMS auf einen hierarchische Verzeichnisbaum hat sich als probate Möglichkeit erwiesen [101, 46]. Jedoch hat die vorliegende Version der NFS-Komponente mit einigen gravierenden Problemen in Entwurf und Implementation zu kämpfen. NFS ist ein sehr umfangreiches, in verschiedenen Versionen verfügbares Protokoll. Es wurde für den Zugriff auf Dateisysteme entfernter Rechner entwickelt. Daher bietet es viele Mechanismen, die für ein Daten-Management-System wie ZIB-DMS wenig oder nicht relevant sind, deren Implementierung aber dennoch teilweise notwendig ist. Die NFS-Komponente implementiert ausschließlich die Version 3 des Protokolls, so dass Neuerungen der aktuellen Version 4 [63] wie beispielsweise die erweiterten Authentifizierungs- und Sicherheitsmechanismen [15, 102] nicht unterstützt werden. Die Kommunikation zwischen Client und Server erfolgt über *Remote Procedure Calls (RPC)* [66], wobei die ausgetauschten Daten gemäß des *External Data Representation (XDR)*-Standards [67] kodiert werden. Die Überführung der internen objektorientierten Datenstrukturen des ZIB-DMS in die komplexe, teils umständliche XDR-Darstellung resultiert in einem hohen Kodierungs- und Dekodierungsaufwand auf der Server-Seite. Moderne, objektorientierte Middleware-Lösungen wie *CORBA* oder *RMI* abstrahieren weitestgehend von der zugrundeliegenden Netzwerk-Kommunikation und bieten einen transparenteren Zugriff auf entfernte Daten und Methoden. Dieser Abstraktionsgrad ist bei RPC-Systemen deutlich niedriger. Beispielsweise muss hier die Anwendung selbst Sorge tragen, dass Aufrufe von nicht-idempotenten Funktionen durch das Auftreten von Netzwerkfehlern nicht mehrfach erfolgen [101]. Die Implementierung eines stabilen und performanten NFS-Servers gestaltet sich daher relativ aufwändig und komplex. Deshalb ist die Verwendung des NFS-Protokolls zur Schaffung einer schlanken Dateisystem-Schnittstelle eher ungünstig. Zudem zeigten sich in der Praxis Defizite bezüglich der Stabilität beim Betrieb der NFS-Komponente und bezüglich der Wartbarkeit ihres Quelltextes. Eine weitere praktische Einschränkung ist, dass es nicht möglich ist, einen herkömmlichen NFS-Server parallel zur NFS-Komponente des ZIB-DMS auf einem Rechner zu betreiben.

Grundsätzlich ist die Directory-View-Komponente verantwortlich für die Abbildung der Funktionalität des Systems in die hierarchische Dateisystem-Struktur. Allerdings wurden weite Teile des Directory-View erst im Zuge der Implementierungsarbeit an der NFS-Komponente umgesetzt, da die bis dahin implementierten Komponenten hauptsächlich den Basic-View nutzten und diese Abbildung in vollem Umfang erst hier erforderlich wurde. Dies führte dazu, dass viele Aspekte und Mechanismen, die konzeptionell dem Directory-View angehören, innerhalb der NFS-Komponente realisiert sind und somit

von anderen Frontend-Komponenten über die ZIB-DMS-API nicht verwendet werden können. Das Konzept verschiedener abgeschlossener Sichtweisen auf den selben Datenbestand ist damit nicht konsequent umgesetzt. Daraus ergibt sich ein weiterer gravierender Mangel im Entwurf der NFS-Komponente. Im Gegensatz zu anderen Komponenten greift diese auf die Dienste des Systems nicht ausschließlich über die Schnittstelle der *ZIB-DMS API* zu. Stattdessen umgeht die NFS-Komponente die Schnittstelle an vielen Stellen, um Funktionen des Directory-View und anderer Komponenten direkt zu rufen. Hauptaufgabe der Schnittstelle ist es, allen *Frontend*-Komponenten einen einheitlichen Zugriff auf das System zu bieten, so dass Änderungen im System entweder durch die Schnittstelle abstrahiert oder durch deren Anpassung allen Zugriffs-Komponenten bekannt gemacht werden. Da sich das System in Entwicklung befindet, wurden im Laufe der Zeit einige notwendige Änderungen an den Schnittstellen und der Semantik verschiedener Komponenten des Systems vorgenommenen. Diese führten aufgrund der "Schichten-Verletzung" zu wachsenden Inkompatibilitäten zwischen der schwer wartbaren NFS-Komponente und dem Rest des Systems.

Zu Beginn dieser Arbeit liegt eine defekte Version der NFS-Komponente vor. Die Anpassungen, die nötig wären, um sie den Veränderungen im System anzupassen, sind umfangreich und werden durch die beschriebenen inhärenten Mängel weiter erschwert. Die einfache Handhabung und die wachsende Stabilität des FUSE-Rahmenwerks, dessen Aufnahme in den Standard-Linux-Kernel sowie zahlreichen Portierungen für andere Betriebssysteme, lassen es zu einer günstigen Alternative für die NFS-Schnittstelle werden. Daher wird die Entscheidung getroffen, die NFS-Komponente aus dem System zu entfernen und diese mit einer Userspace-Dateisystem-Schnittstelle unter Verwendung von FUSE zu ersetzen. Die dabei notwendigen Änderungen am ZIB-DMS und zusätzlichen die Implementierungsarbeiten sind Gegenstand des praktischen Teils dieser Arbeit.

6.2 Vorbereitende Arbeiten

Bevor mit der eigentlichen Implementierung begonnen werden kann, müssen im Vorfeld einige vorbereitende Arbeiten und grundlegende Veränderungen am System vorgenommen werden. Diese stehen zwar teilweise nicht in direktem Zusammenhang mit der Schaffung der FUSE-Komponente, sind aber als Grundlage hierfür notwendig und sinnvoll.

6.2.1 Änderungen am Aufbau des ZIB-DMS

Zunächst wird die NFS-Komponente entfernt. An deren Stelle tritt die *FUSE-Proxy*-Komponente, die das Einbinden des ZIB-DMS-Verzeichnisbaumes in die lokale Verzeichnishierarchie des Systems, das die Server-Instanz ausführt, ermöglicht. Die Realisierung dieser Komponente wird in Abschnitt 6.4 beschrieben. Aufgrund der Erfahrungen mit der

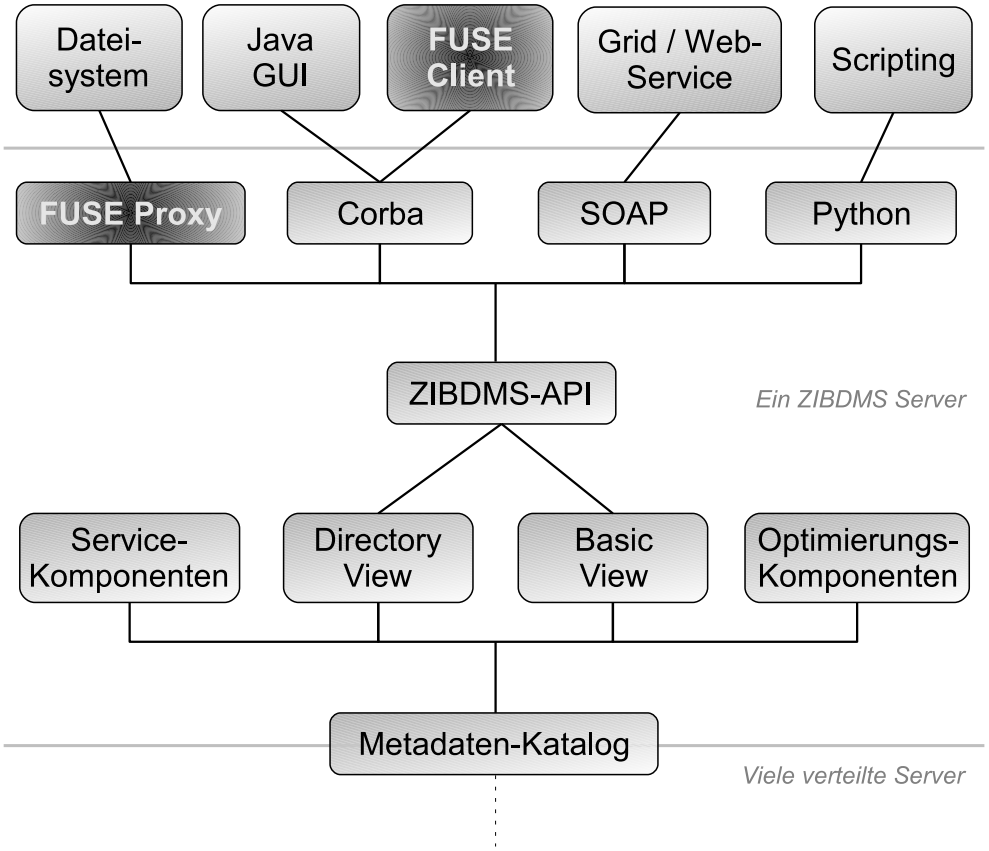

Abbildung 6.1: Veränderte ZIB-DMS Architektur.

NFS-Komponente greift sie allerdings nicht direkt auf die Dienste des Systems zu (vgl. Abbildung 4.2 auf Seite 25), sondern ausschließlich über die ZIB-DMS-API. Damit wird sicher gestellt, dass Veränderungen an Komponenten innerhalb des Systems keine oder zumindest die selbe Auswirkung auf alle Frontend-Komponenten haben. Abbildung 6.1 zeigt den veränderten Aufbau des ZIB-DMS.

Prinzipiell ist es möglich, ein lokal eingebundenes FUSE-Dateisystem unter Verwendung gewöhnlicher Dateiserver-Software für den Netzwerk-Zugriff freizugeben. Damit bleibt zum einen die Möglichkeit ZIB-DMS über NFS zu nutzen erhalten, zum anderen eröffnet sich damit aber auch die Option andere Fileserver-Lösungen, wie *Samba* [96] oder *netatalk* [93] für den entfernen Zugriff zu verwenden. In der aktuellen FUSE-Version treten zwar noch einige Schwierigkeiten bei der Zusammenarbeit mit bestimmten NFS-Server-Implementationen auf, die FUSE-Entwickler arbeiten aber an einer Lösung dieser Probleme.

Für Situationen, in denen ein entfernter Zugriff über Standard-Software nicht möglich oder erwünscht ist, wird zusätzlich ein FUSE-Client geschaffen. Dieses Userspace-Dateisystem ermöglicht das Einbinden des ZIB-DMS auf einem Client-Rechner. Die Kommunikation mit dem Server erfolgt dabei über die CORBA-Anbindung der ZIB-

DMS-API. Die Implementierung des *FUSE-Clients* wird in Abschnitt 6.5 beschrieben.

Da die Funktionalität zur hierarchischen Abbildung der Verwaltungsmechanismen des ZIB-DMS teilweise im Directory-View und teilweise innerhalb der NFS-Komponente implementiert sind, muss diese zunächst vollständig in den Directory-View verlagert werden. An den Stellen, an denen es möglich ist, wird die vorhandene Funktionalität aus der NFS-Komponente übernommen. Viele der Mechanismen müssen jedoch aufgrund ihrer engen Bindung an das NFS-Protokoll und der erwähnten Inkompatibilitäten zu anderen Komponenten völlig neu implementiert werden. Dies wird in Abschnitt 6.3 detailliert beschrieben.

6.2.2 Überarbeitung des Build-Systems

Der Übersetzungsvorgang eines C++-Programmes lässt sich grob in zwei Schritte unterteilen: Die Übersetzung einzelner Quelldateien zu sogenannten *Object-Files* und deren Zusammenfügen zu einem Programm. Ein relativ komplexes System wie das ZIB-DMS setzt sich aus vielen Quelldateien zusammen, deren Übersetzung automatisiert werden muss. Dazu werden die für C++-Software gebräuchlichen Hilfsprogramme *make*, *autoconf*, *automake* und *libtool* – zusammengefasst unter dem Begriff *Build-System* [78] – verwendet. Bei einer großen Anzahl von Quelldateien nimmt die Übersetzung einiges an Zeit in Anspruch. Auf Mehrprozessorsystemen beziehungsweise unter Verwendung verteilter Compiler-Anwendungen wie distcc [47] kann dieser Prozess parallelisiert und damit beschleunigt ablaufen. Da Abhängigkeiten zwischen Quelldateien beachtet werden müssen, erlaubt *make* nur die parallele Übersetzung von Quelldateien eines einzelnen Verzeichnisses. Diese werden dann zu statischen Bibliotheken, sogenannten *Convenience-Libraries*, zusammengefasst. Nachdem die Quelldateien aller Verzeichnisse übersetzt sind, werden in einem abschließenden Linking-Vorgang alle erstellten Convenience-Libraries zu einer Programm-Datei zusammengebunden.

Die Quelltext-Dateien des ZIB-DMS sind in einer stark verschachtelten, tiefen Verzeichnisstruktur abgelegt. Während dies für das Arbeiten der Entwickler ein gewisses Maß an Übersichtlichkeit bringt, wirkt es sich negativ auf den Übersetzungsprozess aus. Manche Verzeichnisse enthalten nur einige wenige Dateien, wodurch die Parallelisierung des Übersetzungsvorgangs erheblich erschwert wird und der Zeitaufwand steigt. Zudem müssen bei einer großen Anzahl von Verzeichnissen ebensoviele statische Bibliotheken erstellt und abschließend zusammengeführt werden. Auch für diesen Vorgang erhöht sich der Zeitaufwand mit wachsender Anzahl der Convenience-Libraries.

Um diese Defizite auszuräumen und um die Implementierung des FUSE-CORBA-Clients zu erleichtern, wird das Build-System und der Aufbau des Quelltext-Verzeichnisbaumes überarbeitet. Für jede Programm-Datei des ZIB-DMS wird ein Verzeichnis auf oberster Ebene angelegt, das den Namen des Programms trägt. Zum Beispiel wird der Quelltext des ZIB-DMS Server-Programms in dem Verzeichnis `zibdmsd` abgelegt, der des

FUSE-CORBA-Clients in dem Verzeichnis `zibdmsfuse`. Die Quelldateien in diesen Verzeichnissen sind nun in einer flacheren Verzeichnisstruktur so organisiert, dass eine logische Gruppierung der Quelltexte zwar erhalten bleibt, eine maximale Verzeichnistiefe von zwei innerhalb des Programm-Verzeichnisses aber nicht überschritten wird und sich möglichst viele Dateien in einem Verzeichnis befinden. Zudem werden Datenstrukturen und Funktionen, die von mehreren Teil-Programmen des Systems genutzt werden, in eine Programm-Bibliothek *libzibdms* ausgelagert. Die zugehörigen Quelldateien werden ebenfalls in einem Verzeichnis auf oberster Ebene abgelegt. Die einzelnen Programme werden dann dynamisch an diese Bibliothek gebunden. Im Gegensatz zum statischen Linking-Vorgang der Convenience-Libraries wird hier nicht der Inhalt der Bibliothek statisch in eine Programmdatei überführt, sondern nur die Bindung zu dieser Bibliothek hinterlegt, die dann bei Ausführung des Programms geladen wird. Dies führt zu einer weiteren Vereinfachung und Verkürzung des Übersetzungsvorgangs. Durch diese Auslagerung wird weiterhin die Verdopplung beziehungsweise das Kopieren von Quelltext-Passagen vermieden und damit die Wartbarkeit des Quelltextes erhöht. Zum selben Zweck werden auch die *IDL*-Definitionen der *CORBA*-Schnittstellen des Systems in einem eigenen Verzeichnis auf oberster Ebene abgelegt.

6.2.3 Verbesserung der Code-Qualität

Um die korrekte Funktionsweise der implementierten Software-Komponenten überprüfen und überwachen zu können, werden im Vorfeld die vorhandenen Testprogramme, sogenannte *Unittests*, ausgebaut und um zusätzliche Tests erweitert. Diese werden unter Verwendung des *cppunit*-Rahmenwerks [84] realisiert und überprüfen, ob das Verhalten der jeweiligen Komponente den Vorgaben entspricht. Mit Hilfe dieser Tests konnten einige Fehler im Quelltext gefunden und beseitigt werden. Die konsequente Erweiterung und Durchführung dieser Tests trägt auf Dauer zur Verbesserung der Qualität des Quelltextes bei und bietet Entwicklern Unterstützung bei der Fehlersuche.

Um den globalen C++-Namensraum zu entlasten und damit Ambiguitäten zu vermeiden, wird außerdem der namespace *zibdms* eingeführt, der alle Deklarationen und Definitionen des ZIB-DMS Quelltextes umschliesst.

6.3 Directory-View

Der Grundgedanke der beiden View-Komponenten ist, wie in Kapitel 4 beschrieben, zwei unterschiedliche, in sich geschlossene, vollwertige Sichtweisen auf den Datenbestand des Systems zu schaffen. Bislang ist jedoch die Kombination von Basic- und Directory-View notwendig, um die volle Funktionalität des Systems nutzen zu können. Die objektorientierte Schnittstelle des Basic-View bietet direkten Zugriff auf die Funktionalität

der Service-Komponenten und allen Verwaltungsmechanismen, außer der Verzeichnis-Hierarchie, die ausschließlich über den Directory-View zugänglich ist. Um die erweiterten Verwaltungsmethoden des Systems auch in der hierarchischen Sicht nutzen zu können, wurden Konzepte zur hierarchischen Abbildung dieser Mechanismen in [101] und [46] eingeführt. Einige dieser Konzepte sind jedoch nicht vollständig realisiert oder bedürfen einer Überarbeitung. Steuerungsmöglichkeiten für die Service-Komponenten fehlen zudem völlig. Der vollständige Ausbau beider View-Komponenten würde allerdings den Rahmen der Arbeit sprengen. Daher werden hier nur die Abbildungen der Verwaltungsmechanismen betrachtet und realisiert. Der Ausbau des Basic-View, sowie die Abbildung der Service-Komponenten im Directory-View müssen in der weiteren Entwicklungsarbeit des Projektes erfolgen.

6.3.1 Änderungen an der Schnittstelle

Die in Abbildung 4.2 auf Seite 33 gezeigte ursprüngliche Schnittstelle des Directory-View definiert Operationen zur Verwaltung von Attributen, die nicht nur auf den System-Attributen der hierarchischen Verwaltungs-Komponente, sondern auf allen Metadaten operieren. Daher liefert die Funktion `getAllAttributes()` eine Liste von Schlüssel-Wert-Paaren variabler Länge zurück. Die Verarbeitung dieser dynamischen Datenstruktur bedeutet einen unnötigen Mehraufwand, da in der hierarchischen Sicht nur die fest vorgegebenen Dateiattribute von Interesse sind. Die Verwaltung der Metadaten erfolgt hier eigentlich mittels virtueller Attribut-Dateien (siehe Abschnitt 6.3.3). Zudem bietet der Basic-View einen Satz an Operationen, die dem selben Zweck dienen. Diese mehrfache Implementierung der selben Funktionalität ist nicht nur unnötig, sondern birgt auch die Gefahr, in Abhängigkeit des gewählten Manipulationsmechanismus unterschiedliches Verhalten zu erzeugen. Desweiteren fehlen die wichtigen Operationen zum Lesen des Ziels eines symbolischen Links und zur Manipulation der Größe einer Datei. Diese Funktionalität wurde bisher über direkten Zugriff auf die entsprechenden System-Attribute realisiert.

Im Zuge der Überarbeitung des Directory-View wird versucht, diese Defizite durch Änderungen an der Schnittstelle zu beheben. Abbildung 6.1 zeigt die Operationen der aktualisierten Schnittstelle. Die `Attribute`-Funktionen werden als veraltet deklariert und sollen weiterhin nicht mehr verwendet werden, bleiben aber vorerst unverändert im Quelltext vorhanden, da ein unmittelbares Entfernen andere Komponenten des Systems beschädigen würde. Die Funktion `getHFNV()` wird ausschließlich von der NFS-Komponente zur Abfrage aller vorhandener HFNs zu einem UFI benutzt und wird daher nicht mehr benötigt, zumal auch der Basic-View Funktionen zur Verfügung stellt, die diese Informationen liefern. Orientiert am POSIX-Standard, werden die Funktionsnamen angepasst und zusätzliche Operationen eingeführt: `stat()`, `chmod()`, `chown()` und `utime()` für das Auslesen und Manipulieren der Dateiattribute, `readlink()` für die Ermittlung des Ziels eines symbolischen Links und `truncate()` zur Manipulation der Dateigröße.

```
/* Verwaltung von Dateiattributen */
void stat     ( const Hfn &path, struct stat *st );
void chown    ( const Hfn &path, uint64_t owner, uint64_t group );
void chmod    ( const Hfn &path, uint64_t mode );
void utime    ( const Hfn &path, const ptime &atime, const ptime &mtime );

/* Verzeichnis- und Datei-Operationen */
vector<Hfn>
     readdir   ( const Hfn &path );
void readdir   ( const Hfn &path, vector<Hfn> &members );
void mkdir     ( const Hfn &path, bool createParents );
void rmdir     ( const Hfn &path, bool recursive );
void create    ( const Hfn &path );
void ln        ( const Hfn &oldpath, const Hfn &newpath );
void unlink    ( const Hfn &path );
void rename    ( const Hfn &oldpath,const Hfn &newpath );
void symlink   ( const Hfn &path, const Hfn &linkinfo );
Hfn  readlink  ( const Hfn &path );
void truncate  ( const Hfn &path, uint64_t filesize );
ssize_t read   ( const Hfn &path, size_t size, uint64_t offset, char *buffer );
ssize_t write  ( const Hfn &path, off_t offset, void *buffer, size_t count );
```

Listing 6.1: Veränderte Directory-View Schnittstelle.

Grundsätzlich ermittelt die Directory-View-Komponente bei Anfragen die zuständige Ebene anhand des HFNs und leitet den Aufruf weiter. Um die hierarchische Darstellung möglichst flexibel zu gestalten und Zusatzinformation an jeder Stelle einblenden zu können, wird der Aufruf der `readdir()`-Operation nicht einfach umgeleitet. Stattdessen werden alle Ebenen (siehe Abbildung 4.8 auf Seite 33) nacheinander abgefragt und die Ergebnisse zusammengefasst. Bislang wurden diese Aufrufe separat durchgeführt und abschliessend die Schnittmenge über den erhaltenen Ergebnissen gebildet, wodurch jede Ebene eine vollständige Abfrage des angefragten Pfades durchführen musste. Dieser Aufwand kann für jede Ebene reduziert werden, indem diese Zugriff auf die Ergebnisse der bereits abgefragten Ebenen erhält. Zu diesem Zweck wird eine zweite Variante der `readdir()`-Operation eingeführt, der nicht nur den Pfad des anzuzeigenden Verzeichnisses, sondern auch eine Liste von Verzeichniseinträgen der vorgeschalteten Ebenen übergeben wird. Somit ist beispielsweise die Collection-Ebene in der Lage, Angaben zur Collection-Mitgliedschaft von den in der Hierarchical-Ebene dargestellten Datenobjekten einzublenden (siehe Abschnitt 6.3.3).

Die Funktion `stat()` wird zum Abruf der für die hierarchische Verwaltung relevanten Metadaten, die Dateiattribute, verwendet. Als Rückgabeparameter wird dabei die ebenfalls im POSIX-Standard definierte Datenstruktur `stat` verwendet. Allerdings sind einige der dort definierten Felder für die Datei-Darstellung im ZIB-DMS gegenstandslos. Daher wird nur die in Listing 6.2 gezeigte Untermenge an Feldern verwendet.

Die Funktionen der Directory-View-Schnittstelle werden durch die ZIB-DMS-API verfügbar gemacht, wobei die Funktionsnamen mit dem Präfix *D_* versehen werden, um sie von den Funktionen des Basic-View (Präfix *B_*) abzugrenzen.

```
struct stat {
    mode_t     st_mode;     // Dateityp und Zugriffsrechte
    nlink_t    st_nlink;    // Anzahl der Hardlinks
    uid_t      st_uid;      // User-ID des Eigentümers
    gid_t      st_gid;      // Gruppen-ID des Eigentümers
    off_t      st_size;     // Größe, in Bytes
    time_t     st_atime;    // Zeitstempel des letzten Zugriffs
    time_t     st_mtime;    // Zeitstempel der letzten Änderung innerhalb der Datei
    time_t     st_ctime;    // Zeitstempel der letzten Änderung der Dateiattribute
};
```

Listing 6.2: Die `stat`-Datenstruktur des Directory-View.

6.3.2 Namensfunktionen

Die Identifikation der Datenobjekte sowie die Ermittlung der zuständigen Ebene im Directory-View erfolgt anhand des HFN. Jede Ebene definiert zu diesem Zweck spezifische syntaktische Regeln, auf deren Basis verschiedene HFN-Typen und Formen der Kodierung zusätzlich notwendiger Informationen definiert werden. Eine detaillierte Beschreibung der verschiedenen Typen, deren Bedeutung und Gestalt wird in Abschnitt 6.3.3 gegeben. Das Erzeugen von HFNs gemäß dieser Regeln erfolgt bislang hardcoded an den relevanten Stellen im Quelltext. Der Mustervergleich zur Erkennung und Prüfung dieser Namen wird mittels einer einzelnen komponenteninternen Funktion `getHfnType()` durchgeführt, die den Typ eines HFN durch einfache String-Vergleiche mit charakteristischen Zeichenfolgen ermittelt. Diese Umsetzungen sind offensichtlich nicht nur schwer wartbar und ineffizient, sie liefern auch keine vollständige, zuverlässige Syntax-Prüfung, da auch syntaktisch ungültige Namen zugelassen werden, solange sie bestimmte Zeichenfolgen enthalten. Es wäre auch wünschenswert, diese Funktionalität an anderen Stellen des Systems nutzen zu können. Dies wird durch die Implementierung innerhalb der Directory-View-Komponente aber verhindert.

Aus diesem Grund wird die gesamte Funktionalität zur Manipulation von HFNs in einem eigens etablierten Sub-Namensraum *zibdms::naming* in die *libzibdms*-Bibliothek ausgelagert, um im gesamten System, also auch in externen Client-Anwendungen, genutzt werden zu können. Hier werden für jeden unterstützten HFN-Typ der neu implementierten Directory-View-Ebenen zentrale, statische Hilfsfunktionen zur Erzeugung, Prüfung und Extraktion nach folgendem Muster eingeführt:

- `is`*HFN-Typ*`()` für die Prüfung und Erkennung eines HFNs vom Typ *HFN-Typ*.
- `create`*HFN-Typ*`()` für die syntaktisch korrekte Erzeugung eines HFNs vom Typ *HFN-Typ*.
- `parse`*HFN-Typ*`()` für die Extraktion der in einem HFN vom Typ *HFN-Typ* kodierten Informationen.

Diese Funktionen verwenden zum Erkennen und Parsen von HFN-Typen nicht einfache manuelle String-Vergleichsoperationen, sondern arbeiten mit *regulären Ausdrücken*.

Ein *regulärer Ausdruck (RegExp)* ist eine nach bestimmten syntaktischen Regeln gebildete Zeichenfolge, die als eine Art Filter- oder Muster-Vorlage für den Vergleich oder die Extraktion von Mengen beziehungsweise Untermengen von Zeichenketten aufgefasst werden kann [21].

Jede reguläre Sprache[1] lässt sich durch einen regulären Ausdruck beschreiben [59]. RegExps stellen somit ein mächtiges und effizientes Werkeug zur Beschreibung und Prüfung der Syntax spezieller HFNs dar. Die Funktionen verwenden dabei die RexExp-Implementation der *boost*-Bibliothek [81]. Die Syntax-Definitionen der verschiedenen HFN-Typen werden als statische vorkompilierte RegExp-Objekte angelegt, wodurch der Verarbeitungsaufwand zur Laufzeit reduziert wird.

Einige der im Folgenden beschriebenen Abbildungs-Ebenen benötigen die Möglichkeit, den vollständigen Pfadnamen eines Datenobjektes zur eindeutigen Identifikation kodiert, in einem hierarchischen Dateinamen als sogenannten *Escaped Path* darstellen zu können. Dies geschah bislang durch die Ersetzung des im ZIB-DMS verwendeten Pfadtrennzeichens / im HFN mit dem Zeichen \. Die Wahl dieses Zeichens ist aber ungünstig, da dieses als Trennzeichen für Pfadnamen in anderen Systemen, unter anderem im CIFS-Netzwerk-Dateisystem, verwendet wird. Dies verhindert den Re-Export des FUSE-Dateisystems des ZIB-DMS über Fileserver-Software wie *Samba* [96]. Daher wird künftig das Zeichen ' bei der Ersetzung verwendet.

6.3.3 Verwaltungsmechanismen

Die Darstellung des ZIB-DMS in einem Verzeichnisbaum basiert auf der Funktionalität der hierarchischen Verwaltungsmechanismen (siehe Abschnitt 4.2.1 auf Seite 26). Die oberste Schicht des Directory-View (siehe Abbildung 4.8 auf Seite 33) ist daher die *Hierarchical*-Ebene. Sie wird bei kumulativen Operationen als Erste angesprochen und gibt somit die grundlegende Struktur vor.

Die darunterliegenden Ebenen stellen die Semantik der restlichen Verwaltungsmethoden durch den Einsatz von *Pseudo-Verzeichnissen* und *Pseudo-Dateien* bereit. Diese virtuellen Verzeichnisse und Dateien, wie sie auch von den in Kapitel 2 beschriebenen Pseudo-Dateisystemen verwendet werden, sind im Gegensatz zu den Objekten der Hierarchical-Ebene in der dargestellten Form nicht im MDC existent. Sie werden vom System zur Laufzeit erzeugt und bilden die Funktionalität der jeweiligen Verwaltungs-Komponente auf den Verzeichnisbaum ab. Die Übergabe von Parametern erfolgt dabei kodiert über den Pfadnamen beziehungsweise durch Schreiben in eine virtuelle Datei. Ergebnisse werden durch Lesen der Pseudo-Verzeichnisse und -Dateien abgerufen.

[1] Reguläre Sprachen bilden die unterste Stufe (Typ-3) der *Chomsky-Hierarchie* für Grammatiken formaler Sprachen. Eine Grammatik ist dann regulär, wenn für alle ihre Abbildungsregeln gilt, dass die linke Seite aus einer einzelnen Variablen und die rechte Seite aus einem einzelnen Terminal-Symbol oder einem Terminal-Symbol gefolgt von einer Variablen besteht. [59].

Zu Beginn der Arbeit ist die Implementation dieser Darstellungs-Ebenen auf die Directory-View- und NFS-Komponente verteilt. Die Funktionalität der *Attribute-File-* und *Querydir*-Schicht für die Metadatenverwaltung und den attributbasierten Zugriff ist teilweise innerhalb der NFS-Komponente realisiert. Die Ebenen *Hierarchical*, *Collection* und *Graph* finden sich im Directory-View, wobei die Collection- und Graph-Abbildungen nicht vollständig implementiert sind. Wie oben erwähnt, ist die Zielsetzung dieser Arbeit der Ausbau des Directory-Views zu einer vollständigen hierarchischen Darstellung der erweiterten Verwaltungsmethoden des ZIB-DMS. Die folgenden Abschnitte beschreiben die Funktionsweise der einzelnen Ebenen und die jeweils vorgenommenen Änderungen an der Implementation anhand der bereitgestellten Operationen. Deren Arbeitsweise wird durch Anwendungsbeispiele verdeutlicht, die als Ein- und Ausgaben von Kommandozeilen-Befehlen, wie sie aus Unix-ähnlichen Systemen bekannt sind, dargestellt. Dabei wird von einem unter dem Pfad */zibdms* eingebundenen ZIB-DMS-System ausgegangen.

Attribut-Dateien

Die Verwaltung der im MDC gespeicherten Metadaten wird in der *Attribute-File*-Schicht über Pseudo-Dateien abgebildet. Für jedes als Datei dargestelltes Datenobjekt in der Verzeichnishierarchie existiert auch eine Attribut-Datei. Diese enthält die Metadaten des korrespondierenden Datenobjekts, wobei jede Zeile genau ein Schlüssel-Wert-Paar in der Form '`Schlüssel = Wert`' enthält. Attribute, denen mehr als ein Wert zugewiesen wurde, werden durch mehrere Zeilen dargestellt. Es werden verschiedene Typen von Attribut-Dateien verwendet, die verschiedene Gruppen von Attributen enthalten. Sie werden durch Anhängen spezifischer Suffixe an den HFN eines Datenobjekts identifiziert:

- `..sysattr` — Diese Attribut-Dateien enthalten die system-internen Attribute einer Datei. Dazu zählen auch Verwaltungs- und Sicherheits-Informationen. Da diese Attribute nicht durch direkten Zugriff, sondern über die vorgesehenen Mechanismen verändert werden sollen, ist die Manipulation dieser Dateien nicht erlaubt.
- `..attr` — Hier werden alle benutzerdefinierten Attribute, also die eigentlichen Metadaten, angezeigt. Auf diesen Typ kann lesend und schreibend zugegriffen werden.
- `..nsl` — Die NSL-Attribut-Datei enthält die URLs aller registrierter Replikate eines Datenobjekts, die durch Manipulation dieser Datei verwaltet werden können.

Da diese Schicht ausschließlich für die Verwaltung von Pseudo-Dateien zuständig ist, implementiert sie nur eine kleine Untermenge der Directory-View-Schnittstelle. Alle verzeichnis- und symlinkbezogenen Operationen sind hier gegenstandslos, ebenso wie die Funktionen zum Erzeugen und Umbenennen von Dateien. Die Dateiattribute der Attribute-Files werden, abgesehen von der Dateigröße, von den korrespondierenden Datenobjekten übernommen, weshalb auch die Prozeduren zur Manipulation der Dateiattribute überflüssig sind. Um die Übersichtlichkeit der Verzeichnis-Hierarchie nicht zu

stark einzuschränken, werden Attribut-Dateien beim Auslesen eines Verzeichnisses nicht ausgegeben, weswegen hier auch die Bereitstellung der kumulativen `readdir()`-Variante entfällt. Somit müssen nur vier Operationen der Directory-View-Schnittstelle bereitgestellt werden, deren in [101] und [46] sukzessiv erarbeitete Implementation, abgesehen von kleineren Anpassungen, übernommen wird:

`stat()`

Zur Anpassung an die veränderte Schnittstelle des Directory-View muss lediglich die Funktion `stat()` implementiert werden. Sie übernimmt die Funktionalität der `getAllAttributes()`-Funktion, liefert aber nur die in der Datenstruktur `stat` (siehe Listing 6.2) definierten Dateiattribute zurück. Diese werden vom korrespondierenden Datenobjekt übernommen. Dazu wird zunächst dessen HFN aus dem übergebenen Pfad ermittelt und verwendet, um die Dateiattribute über die Hierarchical-Ebene abzufragen und zu übernehmen. Die einzige Ausnahme stellt dabei das Attribut *Dateigröße* dar. Dieses enthält die tatsächlichen Größe der Textdarstellung der jeweiligen Metadaten.

`read()`

Die `read`-Prozedur bestimmt anhand des Dateinamens und des verwendeten Attribut-Suffix, welche Metadaten aus dem Katalog gelesen werden müssen, stellt eine entsprechende Anfrage an den MDC und überführt die Ergebnisse in eine lesbare Textform, die in einem Puffer zwischengespeichert wird. Der Inhalt dieses Puffers wird entsprechend der übergebenen `read`-Parameter zurückgeliefert. Das folgende Beispiel zeigt, wie die benutzerdefinierten Attribute der Datei *foo* mit dem Kommandozeilen-Tool *cat* ausgegeben werden können:

```
/zibdms$ cat foo..attr

author = tuschl
context = directory-view
context = zibdms
```

`write()`

Die Manipulation der Metadaten erfolgt über die `write`-Prozedur. Der Inhalt des dabei übergebenen Schreibpuffers wird analysiert, die enthaltenen Attribut-/Wert-Paare extrahiert und mit dem MDC abgeglichen. Auf Anwendungsebene können die Attribute so mit jedem beliebigen Text-Editor bearbeitet werden. Im Stapelbearbeitungsbetrieb kann dies aber auch unter Verwendung der Operatoren zur Ausgabe-Umleitung, wie sie beinahe jeder gängige Kommandozeilen-Interpreter bietet, geschehen. Das folgende Beispiel zeigt, wie durch die Umleitung der Ausgabe mit dem bekannten Befehl *echo* Schlüssel-Wert-Zuweisungen an die Attribut-Datei aus dem vorherigen Beispiel angehängt und somit gesetzt werden können. War ein Schlüssel zuvor schon belegt, wie im Beispiel *context*, wird der neue Wert dem Attribut hinzugefügt. Es fällt auf, dass die Attribut-Datei bei Verwendung

des einfachen Umleitungsoperators > entgegen der Erwartung nicht geleert wird, sondern auch hier die Wert-Zuweisung angehängt wird. Das ist auf die fehlende Implementation der `truncate()`-Funktion zurückzuführen, die bei Verwendung dieses Umleitungsoperators normalerweise vom System zuerst gerufen wird und die Dateigröße auf 0 setzt. Diese veränderte Semantik wurde gewählt, da das System derzeit keine Unterstützung für die Wiederherstellung von gelöschten Metadaten bietet und so ein unbeabsichtigtes Löschen von Metadaten vermieden werden soll.

```
/zibdms$ echo "category = test file" >> foo..attr
/zibdms$ echo "context = fuse" > foo..attr
/zibdms$ cat foo..attr

author = tuschl
category = test file
context = directory-view
context = fuse
context = zibdms
```

Um Attribute von der Kommandozeile aus nicht nur hinzufügen, sondern auch löschen zu können, wird ein zusätzliches Operator-Suffix verwendet. Durch das Anhängen von `.rm` an die Attribut-Dateiendung bei Durchführung der selben Aktion wie im vorherigen Beispiel, wird die angegebene Attribut-Belegung entfernt, sofern sie vorhanden ist. Das folgende Beispiel illustriert diesen Vorgang:

```
/zibdms$ echo "context = directory-view" >> foo..attr.rm
/zibdms$ cat foo..attr

author = tuschl
category = test file
context = fuse
context = zibdms
```

`unlink()`

Die `unlink`-Operation wird normalerweise für das Entfernen von Verzeichniseinträgen und in der weiteren Konsequenz für das Löschen von Dateien verwendet. Da Attribut-Dateien nur virtuell sind und ihre Existenz an die des korrespondierenden Datenobjekts gebunden ist, scheint die Bereitstellung einer solchen Prozedur zunächst zwecklos. Dennoch wird die `unlink`-Funktion implementiert, jedoch mit einer anderen Semantik. Sie bewirkt das Löschen sämtlicher durch die Attribut-Datei repräsentierten Attribute und erfüllt damit die Funktion der fehlenden `truncate()`-Prozedur.

Query-Verzeichnisse

Der attributbasierte Zugriff auf die Datenobjekte des ZIB-DMS wird durch die Abbildung von Suchanfragen in Pseudo-Verzeichnissen, den *Query-Directories (QueryDir)*, bereitgestellt. Solche virtuellen Verzeichnisse enthalten alle Datenobjekte, auf die bestimmte Suchkriterien passen. Diese werden gemäß einer einfachen Abfragesprache, die sich

aus einem oder mehreren elementaren Vergleichstermen der Form '`Schlüssel OP Wert`' zusammensetzt, im Pfadnamen kodiert übergeben. Schlüssel werden im ZIB-DMS immer als Zeichenketten angegeben. Wertangaben können sich auf Zahlenwerte oder Zeichenketten beziehen. Deren Unterscheidung wird standardmäßig vom System durch lexikalische Analyse übernommen. Durch das Einschließen der Wertangabe in doppelte Anführungszeichen (") kann diese aber auch explizit als Zeichenkette deklariert werden. Als Vergleichsoperatoren `OP` stehen der Gleichheitsoperator (==), sowie Bereichsoperatoren (>, <, >=, <=) zur Verfügung. Zudem sind bei der Angabe der Werte Wildcard-Operatoren für exakte oder teilweise Übereinstimmung (*) und für reine Teilübereinstimmung (+) erlaubt. Diese elementaren Terme können mittels der logischen Konjunktions- beziehungsweise Disjunktions-Operatoren (&&, ||) zu komplexeren Anfragen verknüpft und mittels runder Klammern gruppiert werden.

Auch in diesem Fall wird die vorhandene Implementation [46] mit kleineren Anpassungen, bedingt durch die Verlagerung in den Directory-View, beibehalten. Auf den als Dateien dargestellten Ergebnissen der Suchanfragen kann direkt operiert werden, ebenso wie es die Hierarchical-Schicht erlaubt. Daher wird hier die Directory-View-Schnittstelle in vollem Umfang, abgesehen von den Operationen `mkdir()` und `rmdir()`, implementiert. Die einzelnen Funktionen sind dabei aber so aufgebaut, dass sie lediglich den tatsächlichen HFN des angefragten Datenobjektes ermitteln und diesen zur Weiterleitung des Aufrufs an die Hierarchical-Ebene verwenden. Die einzige Ausnahme bildet hier die `readdir()`-Prozedur:

`readdir()`

> Aus dem beim Aufruf übergebenen Pfadnamen wird die Suchanfrage extrahiert und anschliessend lexikalisch und syntaktisch geprüft. Dazu wird das Parser-Rahmenwerk *Spirit*, das Teil der *boost*-Bibliotheken [81] ist, verwendet. Eine detaillierte Beschreibung der Parser-Komponente und der verwendeten Grammatik findet sich in [46]. Da ZIB-DMS separate Namensräume für benutzerdefinierte und System-Attribute verwendet und es so zu Überschneidungen kommen könnte, müssen die Benutzer-Attribute bei der Anfrage durch das Präfix `_u` kenntlich gemacht werden. Die analysierte Anfrage wird dann in die interne `Query`-Objektstruktur (siehe Abschnitt 4.3 auf Seite 30) überführt und die so entstandene Anfrage an den MDC gerichtet. Für jedes in der Ergebnismenge enthaltene Datenobjekt wird ein Verzeichniseintrag erzeugt. Dessen Benennung erfolgt im Regelfall nach dem im HFN enthaltenen Dateinamen des Datenobjektes. Das folgende Beispiel zeigt, wie eine einfache Suchanfrage auf der Kommandozeile durch den Wechsel in ein Query-Verzeichnis gestellt werden kann. Es werden alle Objekte im System abgefragt, deren benutzerdefiniertes Attribut *author* genau den Wert *tuschl* besitzt.

```
/zibdms$ cd "u_author==tuschl"
/zibdms/u_author==tuschl$ ls

foo        fuse.pdf    zibdms.pdf
notes.txt  zibdmsfuse
```

Da die Suchanfrage über den gesamten Datenbestand und unabhängig von der Verzeichnishierarchie durchgeführt wird, kann es häufig dazu kommen, dass zwei Datenobjekten der selbe Dateiname zugeordnet ist. In diesem Fall wird der Verzeichniseintrag aus dem *Escaped-Path* (siehe Abschnitt 6.3.2) des vollständigen HFNs gebildet.

Query-Verzeichnisse werden als globale Entitäten im Verzeichnisbaum behandelt, das heißt sie sind an jeder Stelle der Verzeichnishierarchie verfügbar und werden unabhängig vom angegebenen Pfad verarbeitet. Der Inhalt der Verzeichnisse `/type==pdf` und `/some/path/type==pdf` ist also identisch. Jedoch werden bei Verwendung mehrerer Ebenen von Query-Verzeichnissen, deren Suchkriterien logisch UND-verknüpft. Das QueryDir der untersten Ebene enthält also eine Untermenge der Ergebnisse des darüberliegenden Query-Directories und so weiter. Auf diese Weise ist eine schrittweise Verfeinerung der Anfragen möglich. Das folgende Beispiel veranschaulicht dies. Aus der Ergebnismenge des vorangegangenen Szenarios werden alle als PDF-Dokument deklarierten Objekte extrahiert:

```
/zibdms/u_author==tuschl$ cd u_type==pdf

/zibdms/u_author==tuschl/u_type==pdf$ ls

fuse.pdf    zibdms.pdf
```

Die Formulierung komplexerer Anfragen erfolgt unter Verwendung der Logik- und Gruppierungs-Operatoren. Das folgende Anwendungsszenario gibt ein Beispiel hierfür. Es werden alle im System befindlichen Objekte abgefragt, die als PDF-Dokument deklariert wurden und deren Jahresangabe zwischen 2004 und 2008 liegt:

```
/zibdms$ cd "u_type==pdf && (u_year>=2004 && u_year<2008)"

/zibdms/u_type==pdf && (u_year>=2004 && u_year<2008)$ ls

04451_abstracts_collection.pdf  pham-diplomarbeit.pdf    ZR_07_23.pdf
2003-schuett-iccs.pdf           arsc04-schintke.pdf      zibdms-folien.pdf
2007-schuett-sonar-europar.pdf  ccgrid2004-schintke.pdf  moser-diplomarbeit.pdf
```

Collection

Die Collection-Komponente erlaubt die Bildung von benutzerdefinierten Assoziationen zwischen beliebigen Datenobjekten (siehe Abschnitt 4.2.2 auf Seite 28). Da die zu Beginn der Arbeit vorliegende Implementation der hierarchischen Abbildung dieses Verwaltungsmechanismus unvollständig ist und sich die Darstellungsform im praktischen Betrieb teilweise als ungünstig erwiesen hatte, wird die Collection-Schicht neu implementiert.

Jede Collection wird im Directory-View als virtuelles Verzeichnis dargestellt. Jedes Datenobjekt, das als Mitglied der Collection registriert ist, wird hier durch einen Verzeichni-

seintrag repräsentiert. Um dabei Namenskollisionen zu vermeiden, erfolgt die Benennung anhand der *Escaped-Path*-Darstellung des HFNs. Collection-Verzeichnisse sind flach organisiert, können also keine weiteren Verzeichnisse enthalten. Abbildung 6.2 veranschaulicht dies. Die linke Seite zeigt eine konzeptionelle Darstellung einer Collection und der Verzeichnishierarchie, in der die Datenobjekte angeordnet sind. Auf der rechten Seite wird die Abbildung dieser Collection in die hierarchische Sicht dargestellt.

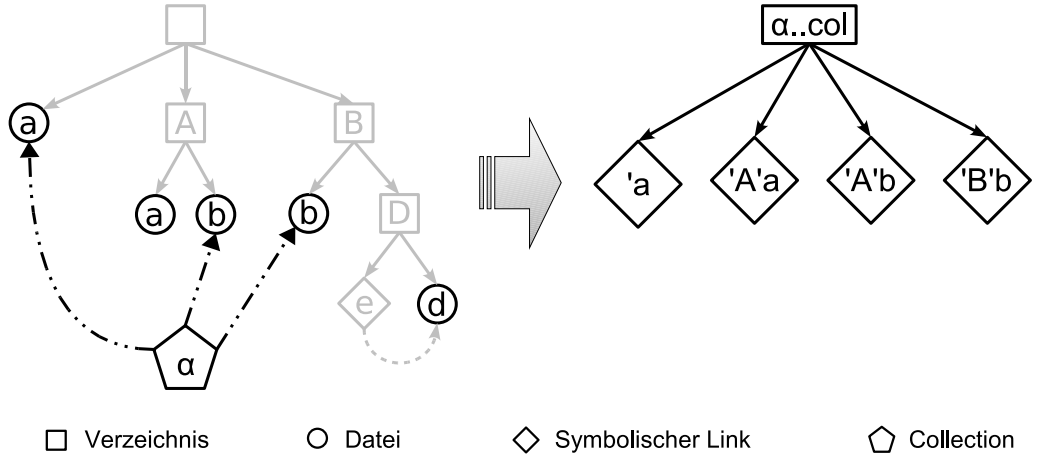

Abbildung 6.2: Hierarchische Darstellung einer Collection

Da die Benennung der Collections selbst in einem von der Verzeichnishierarchie separaten Namensraum erfolgt, kann es auch hier zu Überdeckungen kommen. Um diese zu vermeiden und um Collection-Verzeichnisse identifizieren zu können, werden diese mit dem Suffix `..col` versehen. Insgesamt unterscheidet die Collection-Ebene des Directory-View zwischen vier verschiedenen Typen von Collection-Pfadnamen:

- *Collection-Name*: `/SOME/PATH/COLNAME..col`
  Bezeichnet ein virtuelles Verzeichnis, das alle Mitglieder der Collection `COLNAME` enthält.
- *Collection-Member*: `/SOME/PATH/COLNAME..col/'ESCAPED'PATH`
  Bezeichnet ein Datenobjekt, das Mitglied der Collection `COLNAME` ist und durch den HFN `/ESCAPED/PATH` identifiziert wird.
- *Collection-DirMember*: `/SOME/PATH/FILENAME..{COLNAME}`
  Ein Verzeichniseintrag in einem regulären Verzeichnis, der kennzeichnet, dass das Datenobjekt `/SOME/PATH/FILENAME` Mitglied der Collection `COLNAME` ist.
- *Collection-Maindir*: `/SOME/PATH/..col`
  Bezeichnet ein virtuelles, globales Verzeichnis, das alle existierenden Collection-Verzeichnisse enthält.

Für jeden dieser Typen werden Namensfunktionen, wie in Abschnitt 6.3.2 beschrieben, implementiert. Basierend auf dieser Typisierung wird die Funktionalität der Collection-

Komponente in den implementierten Operationen der Directory-View-Schnittstelle umgesetzt. Die folgenden Abschnitte beschreiben deren Verhalten anhand der jeweils unterstützten Pfad-Typen.

`readdir()`, `stat()`, `readlink()`

Collection-Maindir

Um alle im System vorhandenen Collections anzuzeigen, wird das statische globale *Collection-Maindir* `..col` verwendet. Ein globales Pseudo-Verzeichnis ist, wie auch die Query-Directories, überall in der Hierarchie erreichbar und somit unabhängig von der übrigen Pfadangabe. Es wird beim Auslesen eines Verzeichnisses aber nicht angezeigt.

```
/zibdms$ ls ..col

DA..col     test..col
```

Collection-Name, *Collection-Member*

Durch den Aufruf der `readdir()`-Operation auf einem *Collection-Name*-Pfad, werden die Mitglieder der korrespondierenden Collection ermittelt und als Verzeichniseinträge eines globalen Pseudo-Verzeichnisses dargestellt. Wie alle globalen Pseudo-Verzeichnisse ist auch dieses auf jeder Ebene der Verzeichnis-Hierarchie existent. Während die *Collection-Member* in der ursprünglichen Umsetzung als Pseudo-Dateien abgebildet wurden [46], werden sie hier von der `stat()`-Funktion als symbolische Links ausgewiesen. Die übrigen Dateiattribute werden vom entsprechenden Datenobjekt übernommen. Durch die Verwendung des Symlink-Konzeptes entfällt die Notwendigkeit, eigene `read()`- und `write()`-Operationen in der Collection-Ebene zur Verfügung zu stellen, da Zugriffe auf Anwendungsebene auf dem Ziel-HFN eines Symlinks erfolgen und somit direkt an die Hierarchical-Ebene gerichtet werden. Die `readlink()`-Funktion ermittelt den tatsächlichen HFN dieser Symlink-Abbildung. Da dieser als *Escaped-Path* im Dateinamen des *Collection-Members* kodiert ist, kann dieser Vorgang auf Basis lexikalischer Umwandlung erfolgen, ohne den MDC abfragen zu müssen. Im folgenden Beispiel wird eine Collection *DA* mit vier Mitgliedern angezeigt. Das verwendete Shell-Kommando `ls -l`, wie es auf vielen Systemen zu finden ist, erzeugt eine ausführliche Darstellung des Verzeichnisses, das auch die Ergebnisse der `stat()`- und `readlink()`-Operationen auf den einzelnen Einträgen widerspiegelt.

```
/zibdms$ cd DA..col

/zibdms/DA..col$ ls -l

drwxrwxrwx 2 4096 13:38 .
drwxrwxrwx 6 4096 12:35 ..
lrwxrwxrwx 1   20 13:38 'tuschl'tex'fuse.tex -> /zibdms/tuschl/tex/fuse.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'grid.tex -> /zibdms/tuschl/tex/grid.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'impl.tex -> /zibdms/tuschl/tex/impl.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'main.tex -> /zibdms/tuschl/tex/main.tex
```

Collection-DirMember

Um die Zugehörigkeit eines Datenobjekts zu einer Collection auch außerhalb der Collection-Verzeichnisse anzeigen zu können, wird die kumulierende `readdir()`-Variante implementiert. Diese erhält als Übergabe den angefragten Pfadnamen und eine Liste von Verzeichniseinträgen, die von anderen Schichten des Directory-Views zurückgeliefert wurden. Für jeden Eintrag dieser Liste wird geprüft, ob eine Mitgliedschaft in einer Collection vorliegt und im positiven Fall ein entsprechender *Collection-DirMember*-Eintrag an die Liste angefügt. Dieser Eintrag wird als Symlink umgesetzt und zeigt auf das entsprechende Collection-Verzeichnis. Das folgende Anwendungsbeispiel zeigt, basierend auf dem vorhergehenden Szenario, wie in einem gewöhnlichen Verzeichnis die Mitgliedschaft verschiedener Dateien in der Collection *DA* angezeigt wird.

```
/zibdms/tuschl/tex$ ls

fs.tex           grid.tex         impl.tex..{DA}  zibdms.tex
fuse.tex         grid.tex..{DA}   main.tex
fuse.tex..{DA}   impl.tex         main.tex..{DA}

/zibdms/tuschl/tex$ readlink fuse.tex..{DA}

 -> /zibdms/DA..col
```

`mkdir()`, `rmdir()`

Das Anlegen und Löschen einer Collection wird mit den Operationen `mkdir()` und `rmdir()` realisiert. Beim Entfernen eines gewöhnlichen Verzeichnisses wird geprüft, ob das Verzeichnis leer ist und im gegenteiligen Fall mit einer Fehlermeldung abgebrochen. Diese Prüfung entfällt hier, da das Löschen einer Collection zwar die Collection und die Zuordnungen zu den Datenobjekten, nicht aber diese selbst entfernt. Naturgemäß akzeptieren diese Funktionen nur *Collection-Name*-Pfade. Das nächste Beispiel zeigt das Anlegen und Löschen einer Collection.

```
/zibdms$ mkdir test2..col
/zibdms$ ls ..col

DA..col     test..col     test2..col

/zibdms$ rmdir test2..col
/zibdms$ ls ..col

DA..col     test..col
```

`rename()`

Die `rename()`-Funktion erfüllt zwei unterschiedliche Aufgaben: Das Umbenennen einer Collection und das Hinzufügen eines Datenobjektes zu einer bestehenden Collection. Welche dieser Operationen ausgeführt wird, ist abhängig von der

Parameter-Belegung.

Collection-Name ⇒ *Collection-Name*

Handelt es sich bei den beiden als Parameter übergebenen HFNs um *Collection-Name*-Pfade, wird der Aufruf als Anfrage zur Namensänderung einer Collection interpretiert, wie das folgende Anwendungsbeispiel illustriert.

```
/zibdms$ mv DA..col DiplomArbeit..col
/zibdms$ ls -l DiplomArbeit..col

drwxrwxrwx 2 4096 13:40 .
drwxrwxrwx 6 4096 12:35 ..
lrwxrwxrwx 1   20 13:38 'tuschl'tex'fuse.tex -> /zibdms/tuschl/tex/fuse.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'grid.tex -> /zibdms/tuschl/tex/grid.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'impl.tex -> /zibdms/tuschl/tex/impl.tex
lrwxrwxrwx 1   22 13:39 'tuschl'tex'zibdms.tex -> /zibdms/tuschl/tex/zibdms.tex
```

Gewöhnlicher HFN ⇒ *Collection-Name*

Wird als erstes Argument ein regulärer HFN und als zweites Argument ein *Collection-Name* übergeben, wird dies als Aufforderung verstanden, das über den HFN referenzierte Datenobjekt der Collection hinzuzufügen. Im folgenden Anwendungsszenario wird gezeigt, wie das Datenobjekt *zibdms.tex* im aktuellen Verzeichnis */zibdms/tuschl/tex* zur Collection *DA* mit dem bekannten Unix-Kommando *mv* hinzugefügt werden kann.

```
/zibdms/tuschl/tex$ mv zibdms.tex DA..col
/zibdms/tuschl/tex$ ls -l DA..col

drwxrwxrwx 2 4096 13:39 .
drwxrwxrwx 6 4096 12:35 ..
lrwxrwxrwx 1   20 13:38 'tuschl'tex'fuse.tex -> /zibdms/tuschl/tex/fuse.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'grid.tex -> /zibdms/tuschl/tex/grid.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'impl.tex -> /zibdms/tuschl/tex/impl.tex
lrwxrwxrwx 1   20 13:38 'tuschl'tex'main.tex -> /zibdms/tuschl/tex/main.tex
lrwxrwxrwx 1   22 13:39 'tuschl'tex'zibdms.tex -> /zibdms/tuschl/tex/zibdms.tex
```

Da *Collection-Member* nach dem kodierten HFN der zugehörigen Datenobjekte benannt werden und daher nicht umbenannt werden können und das Entfernen aus einer Collection anderweitig realisiert ist, werden alle anderen Parameter-Konstellationen als ungültig abgewiesen.

`unlink()`

Das Entfernen eines Datenobjektes aus einer Collection wird durch die `unlink()`-Operation ermöglicht. Dies kann entweder direkt unter Verwendung des *Collection-Member*-Pfades oder durch Angabe eines *Collection-DirMember*-HFNs erfolgen. Das folgende Beispiel zeigt, wie unter Einsatz der beiden Varianten zwei Mitglieder aus der im vorangegangenen Anwendungsszenario verwendeten Collection *DA* entfernt werden können.

```
/zibdms/tuschl/tex$ unlink DA..col/'tuschl'tex'impl.tex
/zibdms/tuschl/tex$ unlink zibdms.tex..{DA}
/zibdms/tuschl/tex$ ls -l DA..col

drwxrwxrwx 2  4096 13:39 .
drwxrwxrwx 6  4096 12:35 ..
lrwxrwxrwx 1    20 13:38 'tuschl'tex'fuse.tex -> /zibdms/tuschl/tex/fuse.tex
lrwxrwxrwx 1    20 13:38 'tuschl'tex'grid.tex -> /zibdms/tuschl/tex/grid.tex
lrwxrwxrwx 1    20 13:38 'tuschl'tex'main.tex -> /zibdms/tuschl/tex/main.tex
```

Graph

Während die Darstellung einer Assoziation als globales Pseudo-Verzeichnis relativ naheliegend erscheint, gestaltet sich die Abbildung von beliebigen Relationen bedeutend komplizierter. Insbesondere ist die Umsetzung der Funktionalität der Graph-Komponente über limitierte Schnittstelle des Directory-View eine anspruchsvolle Aufgabe. Das in [46] vorgestellte Abbildungskonzept kann nicht verwendet werden, da sich die Funktionsweise, der Aufbau und die Semantik der Graph-Verwaltungs-Komponente seit dessen Implementierung verändert haben. Deshalb wird die ohnehin unvollständige ursprüngliche Implementation der Graph-Ebene des Directory-View ersetzt.

Ein Graph im ZIB-DMS ist eine Sammlung von beliebig vielen gerichteten Kanten. Jede dieser Kanten kann 1 bis n Start- und Ziel-Knoten haben und mit benutzerdefinierten Attributen belegt werden, wodurch sich unter anderem eine Gewichtung der Kanten ausdrücken läßt. Die ohnedies schwierige Abbildung solch komplexer Gebilde in einer hierarchischen Struktur wird durch die Funktionsweise der vorliegenden Implementation der Graph-Verwaltungskomponente zusätzlich erschwert. Obwohl Kanten als eigenständige Objekte im Metadaten-Katalog gespeichert werden und daher mit Attributen belegt werden können, sieht die Komponente keine Möglichkeit der Benennung der Kanten vor. Stattdessen wird das System-Attribut *NAME* automatisiert mit einer 32-Zeichen langen Textdarstellung des UFI belegt. Die Darstellung dieser kryptischen Bezeichner würde die Übersichtlichkeit der ohnehin komplexen Darstellung bedeutend verschlechtern. Bei Kanten, die über die Directory-View-Ebene angelegt werden, könnte eine Benennung durch Überschreiben des *NAME*-Attributs forciert werden. Dies hätte aber keine Auswirkung auf die Kanten, die über den Basic-View angelegt werden, wie es beispielsweise in der ZIB-DMS-GUI geschieht. Es wurde erwogen, eine Anpassung der Graph-Komponente vorzunehmen. Davon wird aber abgesehen, da die weitreichenden Änderungen im gesamten System, die dies nach sich ziehen würde, den Rahmen der Arbeit sprengen würden. Daher wird das Graph-Konzept unter Abstraktion der Kanten-Objekte mit kleineren Einschränkungen bezüglich der Funktionalität umgesetzt. Das bedeutet, Kanten werden nicht dargestellt, sondern nur die daraus resultierenden Verbindungen zwischen einzelnen Knoten. Damit können auch Graphen angezeigt werden, die mit dem Basic-View angelegt wurden, ohne die Übersichtlichkeit zu beeinträchtigen. Bei der Erzeugung eines Graphen über den Directory-View wird jeder Knoten mit einer eingehenden und einer ausgehenden Kante versehen. Diese können beliebig viele andere

Knoten als Quelle (im Fall der eingehenden Kante) oder Ziel (im Fall der ausgehenden Kante) aufnehmen. Es können also immer nur auf einer Seite einer Kante mehrere Knoten eingetragen werden. Zudem ist es auf diese Art nicht möglich zwei unterschiedliche Kanten mit identischer Quell- und Ziel-Konfiguration anzulegen.

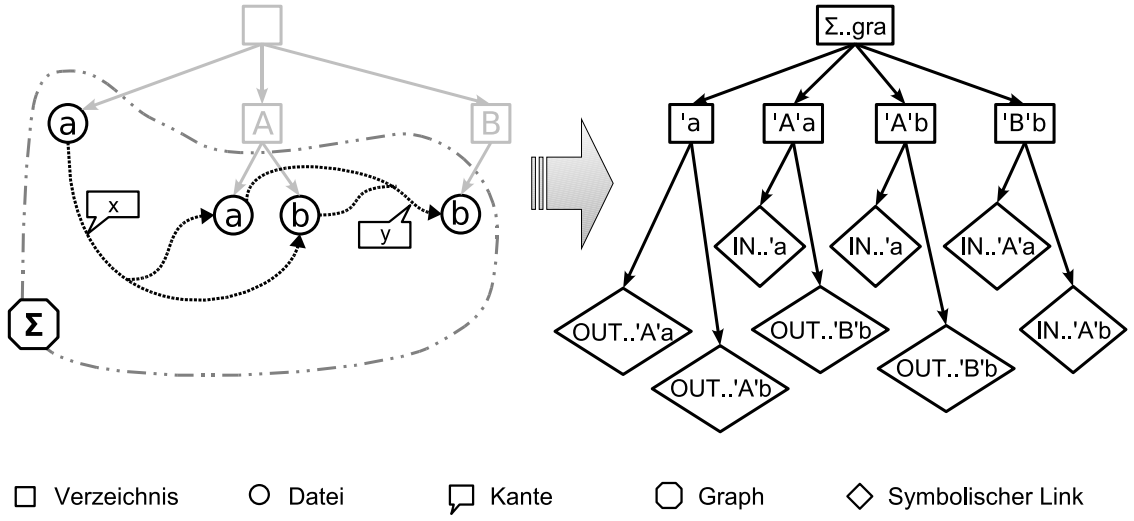

Abbildung 6.3: Hierarchische Abbildung eines Graphen

Ein Graph wird, wie in Abbildung 6.3 illustriert, als globales Pseudo-Verzeichnis dargestellt. Auch die Graph-Komponente verwendet einen separaten Namensraum, weswegen die Erweiterung `..gra` zur Identifikation eines Graph-Verzeichnisses verwendet wird. Dieses enthält für jedes Datenobjekt, das als Quell- oder Ziel-Knoten einer Kante eingetragen ist, ein Unterverzeichnis, das nach dessen *Escaped-Path* benannt ist. Innerhalb dieses virtuellen Verzeichnisses wird jede Verbindung zu einem Datenobjekt als Symlink dargestellt, der ebenfalls nach dessen *Escaped-Path* benannt ist. Dabei wird die Richtung der Verbindung – also eingehende oder ausgehende Kante – durch das Präfix `IN..` beziehungsweise `OUT..` angegeben.

Die Graph-Ebene definiert sechs verschiedene HFN-Typen:

- *Graph-Name*: `/SOME/PATH/GRAPHNAME..gra`
  Bezeichnet ein globales virtuelles Verzeichnis, das alle im Graphen `GRAPHNAME` auftretenden Knoten enthält.
- *Graph-UpperNode*: `/SOME/PATH/GRAPHNAME..gra/'NODE'PATH`
  Bezeichnet ein virtuelles Verzeichnis, das alle Kanten des Graphen `GRAPHNAME` enthält, in denen das durch den HFN `/NODE/PATH` identifizierte Datenobjekt als Quell- oder Ziel-Knoten auftaucht.
- *Graph-LowerNode*: `/SOME/PATH/GRAPHNAME..gra/'NODE'PATH/IN..'ESCAPED'PATH`
  Bezeichnet ein durch den HFN `/ESCAPED/PATH` identifiziertes Datenobjekt von dem eine Kante im Graphen `GRAPHNAME` zum Datenobjekt `/NODE/PATH` führt. Für die umgekehrte Richtung wird `OUT` statt `IN` verwendet.

- *Graph-NodeSubdir*: `/SOME/PATH/GRAPHNAME..gra/'NODE'PATH/..IN`
  Bezeichnet ein Pseudo-Verzeichnis, das verwendet wird, um eine eingehende Kante im Graphen `GRAPHNAME` am Knoten `/NODE/PATH` anzulegen. Für ausgehende Kanten wird das Verzeichnis `..OUT` verwendet.
- *Graph-DirMember*: `/SOME/PATH/FILENAME..GRAPHNAME..IN..(FILE1, ..., FILEn)`
  Ein Verzeichniseintrag in einem regulären Verzeichnis, der kennzeichnet, dass das Datenobjekt `/SOME/PATH/FILENAME` eine eingehende Kante im Graphen `GRAPHNAME` besitzt. Für ausgehende Kanten wird `OUT` statt `IN` verwendet. Als Zusatzinformation werden die ersten n Dateinamen der Quell- beziehungsweise Ziel-Knoten angegeben.
- *Graph-Maindir*: `/SOME/PATH/..gra`
  Bezeichnet ein globales virtuelles Verzeichnis, das alle existierenden Graph-Verzeichnisse enthält.

Auf Basis dieser Typisierung wird die Funktionalität der Graph-Komponente auf die Dateisystem-Operationen abgebildet. In den folgenden Abschnitten wird dies anhand der implementierten Operation und deren untestützten HFN-Typen erläutert und wieder mittels Anwendungsbeispielen illustriert. Dabei wird die in Abbildung 6.4 gezeigte Relation als Basis verwendet.

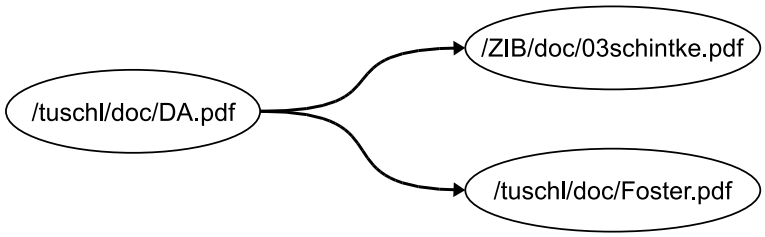

Abbildung 6.4: Beziehungen zwischen den Objekten der Anwendungsbeispiele

`readdir()`, `stat()`, `readlink()`

Graph-Maindir

Um alle im System existenten Graphen anzuzeigen, wird das globale Pseudo-Verzeichnis `..gra` verwendet. Es enthält für jeden Graph einen *Graph-Name*-Verzeichniseintrag.

```
/zibdms$ ls ..gra

CITE..gra     test..gra
```

Graph-Name

Beim Aufruf auf einem *Graph-Name*-HFN werden alle Datenobjekte ermittelt, die als Knoten in dem durch den HFN spezifizierten Graphen auftreten und jeweils ein *Escaped-Path*-Verzeichniseintrag erzeugt. Die von `stat()` zurückgelieferten Datei-

attribute werden von den Datenobjekten übernommen. Die einzige Ausnahme ist dabei der Dateityp, der als Verzeichnis angegeben wird.

```
/zibdms$ ls -l CITE..gra

drwxr-xr-x tuschl users 2 4096 17:23 'tuschl'DA.pdf
drwxr-xr-x tuschl users 2 4096 17:25 'tuschl'doc'Foster.pdf
drwxr-xr-x tuschl users 2 4096 17:28 'ZIB'doc'03schintke.pdf
```

Graph-UpperNode, *Graph-LowerNode*, *Graph-NodeSubdir*

Wird die Operation mit einem *Graph-UpperNode*-HFN gerufen, werden die im HFN kodierten Informationen extrahiert und verwendet, um alle Kanten für die das angegebene Datenobjekt im fraglichen Graphen als Quell- oder Ziel-Knoten eingetragen ist, abzufragen. Für jede Verbindung zu einem Knoten wird dann ein *Graph-LowerNode*-Verzeichniseintrag erzeugt. Dieser wird als Symlink realisiert, der auf den HFN des entsprechenden Datenobjektes zeigt. Somit entfällt auch hier die Notwendigkeit der Bereitstellung von eigenen `read()`- und `write()`-Implementationen. Die beiden statischen Pseudo-Verzeichnisse `..IN` und `..OUT` repräsentieren die eingehende und ausgehende Kante eines Knotens und bleiben immer leer, da sie nur verwendet werden, um Knoten zu der entsprechenden Kante hinzuzufügen (siehe unten). Da diese Verzeichniseinträge mit '.' beginnen, werden sie als versteckte Verzeichnisse abgebildet, wie auch das folgende Anwendungsbeispiel zeigt:

```
/zibdms$ ls "CITE..gra/'tuschl'DA.pdf"

OUT..'tuschl'doc'Foster.pdf     OUT..'ZIB'doc'03schintke.pdf

/zibdms$ readlink "CITE..gra/'tuschl'DA.pdf/OUT..'tuschl'doc'Foster.pdf"

 -> /zibdms/tuschl/doc/Foster.pdf

/zibdms$ ls -a "CITE..gra/'tuschl'DA.pdf"

.   ..IN    OUT..'tuschl'doc'Foster.pdf
..  ..OUT   OUT..'ZIB'doc'03schintke.pdf
```

Graph-DirMember

Die Graph-Ebene verwendet wie auch die Collection-Schicht die kumulierende `readdir()`-Variante, um Informationen zu Relationen zwischen Datenobjekten außerhalb der virtuellen Graph-Verzeichnisse anzuzeigen. Für jedes Element der beim Aufruf übergebenen Liste von Verzeichniseinträgen werden vorhandene Relationen kodiert in *Graph-DirMember*-Einträgen angehängt. Diese beinhalteten den Namen des Graphen, die Richtung der jeweiligen Kante und die Namen der Quell- beziehungsweise Ziel-Knoten. Da diese Liste beliebig lang werden kann, wird nur eine konfigurierbare maximale Anzahl von Namen angezeigt. Der verwendete Standardwert liegt hier bei drei. Sind mehr Knoten eingetragen, wird dies durch Anhängen von '...' an die Namensliste signalisiert.

```
/zibdms/tuschl$ ls

DA.pdf   DA.pdf..CITE..OUT..(03schintke.pdf,Foster.pdf)
doc      slides
tex

/zibdms/tuschl$ readlink "DA.pdf..CITE..OUT..(03schintke.pdf,Foster.pdf)"

 -> /zibdms/tuschl/CITE..gra/'tuschl'DA.pdf

/zibdms/tuschl$ cd ../ZIB/doc

/zibdms/ZIB/doc$ ls

03schintke.pdf                03schintke.pdf..CITE..IN..(DA.pdf)
2004-11-ZIBDMS-dagstuhl.ppt   ccgrid2004-schintke.pdf
pham-diplomarbeit.pdf         witte-diplom-nfs.pdf
ZR_07_23.pdf

/zibdms/ZIB/doc$ readlink "03schintke.pdf..CITE..IN..(DA.pdf)"

 -> /zibdms/ZIB/doc/CITE..gra/'ZIB'doc'03schintke.pdf
```

`mkdir()`, `rmdir()`

Graph-Name

Das Anlegen eines neuen Graphen erfolgt durch den Aufruf der `mkdir()`-Operation mit einem *Graph-Name*-HFN. Analog wird das Löschen eines Graphen inklusive aller Kanten über `rmdir()` abgewickelt.

```
/zibdms$ mkdir test2..gra
/zibdms$ ls ..gra

CITE..gra     test..gra     test2..gra

/zibdms$ rmdir test..gra
/zibdms$ ls ..gra

CITE..gra     test2..gra
```

Graph-UpperNode

Der Aufruf von `rmdir()` mit einem *Graph-UpperNode*-Argument bewirkt, dass das entsprechende Datenobjekt vollständig aus dem Graph entfernt wird. Das bedeutet, dass der fragliche Knoten aus allen Kanten des Graphen, in denen er als Quell- oder Ziel-Knoten auftritt, entfernt wird. Die Kanten selbst bleiben bestehen. Das Erzeugen neuer Knoten wird jedoch nicht mit der `mkdir()`-Prozedur realisiert, sondern mittels der `rename()`-Operation (siehe unten), so dass der `mkdir()`-Aufruf mit einem *Graph-UpperNode*-Argument als ungültig abgewiesen wird.

```
/zibdms$ ls CITE..gra

'tuschl'DA.pdf              'tuschl'doc'Foster.pdf
'ZIB'doc'03schintke.pdf

/zibdms$ rmdir "CITE..gra/'tuschl'doc'Foster.pdf"
/zibdms$ ls CITE..gra

'tuschl'DA.pdf              'ZIB'doc'03schintke.pdf

/zibdms$ ls ./tuschl

DA.pdf      DA.pdf..CITE..OUT..(03schintke.pdf)
doc         slides
tex
```

`rename()`

Graph-Name* ⇒ *Graph-Name

Ein `rename()`-Aufruf mit zwei *Graph-Name*-Argumenten wird als Namensänderung eines Graphen aufgefasst.

```
/zibdms$ ls ..gra

CITE..gra     test2..gra

/zibdms$ mv CITE..gra DA-CITE..gra
/zibdms$ ls ..gra

DA-CITE..gra  test2..gra
```

regulärer HFN ⇒ *Graph-NodeSubdir*

Wie bereits erwähnt, werden die eingehende und die ausgehende Kante eines Knotens für Manipulationen durch die virtuellen `Graph-NodeSubdir`-Unterverzeichnisse repräsentiert. Durch Aufruf der `rename()`-Funktion mit einem regulären HFN als erstes und einem *Graph-NodeSubdir* als zweites Argument, wird das durch den HFN referenzierte Datenobjekt der entsprechenden Kante als Quell- beziehungsweise Ziel-Knoten hinzugefügt. Im folgenden Anwendungsszenario wird das Datenobjekt */ZIB/doc/ZR_07_23.pdf* der ausgehenden Kante des Knotens */tuschl/-DA.pdf* im Graphen *CITE* hinzugefügt.

```
/zibdms$ ls ./ZIB/doc

03schintke.pdf                03schintke.pdf..CITE..IN..(DA.pdf)
2004-11-ZIBDMS-dagstuhl.ppt   ccgrid2004-schintke.pdf
pham-diplomarbeit.pdf         witte-diplom-nfs.pdf
ZR_07_23.pdf

/zibdms$ mv ./ZIB/doc/ZR_07_23.pdf "DA-CITE..gra/'tuschl'DA.pdf/..OUT"
/zibdms$ ls CITE..gra

'tuschl'DA.pdf            'ZIB'doc'03schintke.pdf
'ZIB'doc'ZR_07_23.pdf
```

```
/zibdms$ ls ./ZIB/doc

03schintke.pdf                03schintke.pdf..CITE..IN..(DA.pdf)
2004-11-ZIBDMS-dagstuhl.ppt   ccgrid2004-schintke.pdf
pham-diplomarbeit.pdf         witte-diplom-nfs.pdf
ZR_07_23.pdf                  ZR_07_23.pdf..CITE..IN..(DA.pdf)

/zibdms$ ls ./tuschl

DA.pdf       DA.pdf..CITE..OUT..(03schintke.pdf,ZR_07_23.pdf)
doc          slides
tex
```

regulärer HFN ⇒ *Graph-Name*

Ein neu angelegter Graph enthält zunächst keine Knoten oder Kanten und das entsprechende Graph-Verzeichnis damit keine *Graph-UpperNode*-Einträge. Um in einer solchen Situation initial eine Relation anlegen zu können, wird ebenfalls der `rename()`-Aufruf verwendet. Das durch den ersten HFN-Parameter identifizierte Datenobjekt wird als Knoten in dem Graphen registriert, der durch den *Graph-Name*-Parameter referenziert wird. Das bedeutet, es wird eine eingehende und eine ausgehende Kante auf dem entsprechenden Datenobjekt mit leerer Quell- beziehungsweise Ziel-Knotenangabe angelegt. Dadurch wird in dem korrespondierenden Graph-Verzeichnis ein *Graph-UpperNode*-Eintrag für diesen Knoten angezeigt und Verbindungen können wie oben beschrieben angelegt werden.

```
/zibdms/tmp$ ls

foo     bar

/zibdms/tmp$ mkdir test..gra
/zibdms/tmp$ ls test..gra


/zibdms/tmp$ mv ./foo test..gra
/zibdms/tmp$ ls -l test..gra

drwxr-xr-x 2 4096 17:23 'tmp'foo
```

`unlink()`

Die `unlink()`-Operation wird verwendet, um eine einzelne Verbindung zwischen zwei Datenobjekten aufzuheben. Aus diesem Grund ist als Argument hier nur ein *Graph-LowerNode*-HFN erlaubt.

```
/zibdms$ cd "CITE..gra/'tuschl'DA.pdf"

/zibdms/CITE..gra/'tuschl'DA.pdf$ ls

OUT..'ZIB'doc'03schintke.pdf    OUT..'ZIB'doc'ZR_07_23.pdf

/zibdms/CITE..gra/'tuschl'DA.pdf$ unlink OUT..'ZIB'doc'ZR_07_23.pdf
```

6.4 FUSE-Proxy

Die FUSE-Proxy-Komponente ist als einfache und leichtgewichtige Zugriffskomponente konzipiert. Das bedeutet, sie wurde so entworfen, dass nur die notwendige Logik zur Integration des ZIB-DMS in den lokalen Verzeichnisbaum des Systems enthalten ist. Die Darstellung des Datenbestandes und die Abbildung der Funktionalität des Systems auf die POSIX-Dateisystem-Schnittstelle wird weitestgehend von der oben beschriebenen View-Komponente realisiert[2].

Das in Kapitel 5 detailliert vorgestellte FUSE-Rahmenwerk bietet eine komfortable Grundlage für die Implementierung eines Userspace-Dateisystems. Hierzu stehen zwei Ausprägungen der Programmierschnittstelle zur Verfügung: Die *Lowlevel*- und die *Highlevel*-API, deren Unterschiede in Abschnitt 5.4 auf Seite 55 im Einzelnen beschrieben werden. Während Objekte in Dateisystemen auf Anwendungsebene durch Pfadnamen identifiziert werden, verwendet die VFS-Schicht des Betriebssystems, wie in Abschnitt 5.2 auf Seite 44 beschrieben, intern die *inode*-Abstraktion für den Zugriff auf diese Objekte, wodurch eine Abbildung von *inode*-Bezeichnern auf Pfadnamen notwendig ist. Diese Abbildung muss bei Verwendung der *Lowlevel*-API durch das Userspace-Dateisystem bereitgestellt werden. Grundsätzlich könnte die *HFN-UFI* Abstraktion des ZIB-DMS hierfür verwendet werden. Die Abbildungen der erweiterten Verwaltungsmechanismen des Directory-View verwenden aber virtuelle Pseudo-Dateien und -Verzeichnisse, die ausschließlich anhand der im HFN kodierten Informationen identifiziert werden und denen deshalb kein UFI zugeordnet ist. Für diese Fälle müßte die FUSE-Komponente des ZIB-DMS eine stabile Abbildung von diesen virtuellen HFNs auf virtuelle UFis, sowie Mechanismen zum Zwischenspeichern dieser Informationen bereitstellen. Die *Highlevel*-Schnittstelle hingegen übernimmt die Abbildung der ohnehin nur im lokalen System gültigen *inode*-Bezeichner auf Pfadnamen und kümmert sich um die effiziente Verwaltung und das Caching in bibliotheksinternen Zuordnungs-Tabellen. Die Funktionen der *Highlevel*-API arbeiten daher ebenso wie die der Directory-View-Schnittstelle mit absoluten Pfadnamen, wodurch die Anbindung an das System erleichtert wird. Zudem bietet nur die *Highlevel*-API die Möglichkeit, nur einen Teil der Schnittstellen-Funktionen zu implementieren und für die reslichen Operationen ein durch die Bibliothek definiertes Standardverhalten zu nutzen. Dies ermöglicht nicht nur eine schrittweise Erweiterung der Funktionalität während der Entwicklungsarbeit, es entfällt auch die Notwendigkeit der Implementierung irrelevanter Operationen. Die Nutzung der *Lowlevel*-Schnittstelle, die hauptsächlich zur Schaffung von Anbindungen an neue Programmiersprachen und seltener zur tatsächlichen Implementierung von Dateisystemen verwendet wird, würde daher nur einen relativ hohen Mehraufwand bedeuten und keine nennenswerten Vorteile bieten. Aus diesen Gründen wird hier die *Highlevel*-API genutzt.

[2] Die Bezeichnung *Proxy* soll verdeutlichen, dass es sich um eine leichtgewichtige Zugriffskomponente handelt, die in Anlehnung an das *Proxy*-Entwurfsmuster [23] Dateisystemanfragen ohne tiefgreifende Bearbeitung an die Directory-View weiterleitet.

6.4.1 Implementierung der FUSE-Schnittstelle

Die FUSE-Proxy-Komponente stattet also den ZIB-DMS-Server mit der Funktionalität eines Userspace-Dateisystem durch Implementierung der *Highlevel*-Variante der FUSE-Schnittstelle aus, wie es in Kapitel 5 beschrieben wurde. Dabei werden die einzelnen Operationen mittels der Directory-View-Funktionen der ZIB-DMS-API realisiert. Die FUSE-Schnittstelle wird jedoch nicht vollständig implementiert, sondern nur in dem Umfang, wie er in Listing 6.3 dargestellt ist.

```
namespace zibdms {

    namespace fuse {

        int getattr  ( const char *path, struct ::stat *stbuf );
        int readdir  ( const char *path, void *buf, fuse_fill_dir_t filler,
                       off_t offset, struct fuse_file_info *fi );
        int mkdir    ( const char *path, mode_t mode );
        int rmdir    ( const char *path );
        int mknod    ( const char *path, mode_t mode, dev_t );
        int link     ( const char *file, const char *newfile );
        int symlink  ( const char *file, const char *link );
        int readlink ( const char *path, char *buf, size_t bufsize );
        int unlink   ( const char *path );
        int rename   ( const char *oname, const char *nname );
        int chmod    ( const char *path, mode_t mode );
        int chown    ( const char *path, uid_t user, gid_t group );
        int utime    ( const char *path, struct utimbuf *time_buf );
        int truncate ( const char *path, off_t offset );
        int open     ( const char *path, struct fuse_file_info *fi );
        int read     ( const char *path, char *buf, size_t size,
                       off_t offset, struct fuse_file_info *fi );
        int write    ( const char *path, const char *buf, size_t count,
                       off_t offset, struct fuse_file_info *fi );

    } // namespace fuse
} // namespace zibdms
```

Listing 6.3: Operationen der FUSE-Proxy-Komponente

Dies ist einerseits durch den Funktionsumfang der Directory-View-Komponente bedingt, andererseits sind einige Operationen der Schnittstelle für die Anbindung des ZIB-DMS weitestgehend irrelevant. Da die Funktionen des Directory-View, wie in Kapitel 4 beschrieben, zustandslos arbeiten, sind Operationen zum Öffnen und Schließen von Dateien und Verzeichnissen (zum Beispiel `opendir()` oder `close()`), sowie spezialisierte Varianten von Funktionen, die auf geöffneten Dateisystemobjekten operieren (zum Beispiel `ftruncate()` oder `fgetattr()`), gegenstandslos und werden nicht implementiert. Stattdessen wird das von der FUSE-Bibliothek bereitgestellte Standardverhalten genutzt, das eine reibungslose Funktionsweise sicherstellt (siehe Kapitel 5). Das ZIB-DMS ist nicht für den Einsatz als verteiltes Dateisystem konzipiert. Das Speichern der eigentlichen Daten erfolgt transparent auf externen Speicherquellen, wobei eine *Write-Once*-Semantik verwendet wird. Daher sind auch die Funktionen für direkten I/O-Zugriff, zur Synchronisierung von Datei-Puffern und zum Setzen und Verwalten von Dateisperren (*Locks*) nicht

erforderlich. FUSE bietet auch Unterstützung für *Extended Attributes (xattr)*, wie sie im *XFS*-Dateisystem [70] verwendet werden. Diese könnten prinzipiell zur Darstellung und Verwaltung der Metadaten des ZIB-DMS genutzt werden. Der praktische Nutzen dieser Mechanismen ist allerdings beschränkt, da diese von vielen Standard-Anwendungen[3] bislang nicht unterstützt werden. Zudem sind zur Verwaltung dieser erweiterten Attribute spezielle Kommandozeilen-Tools erforderlich, deren Verfügbarkeit nicht auf jedem System gewährleistet ist. Somit wird von der Verwendung dieser Operationen zur Umsetzung der Metadatenverwaltung abgesehen, zumal hierfür das *Attribute-File*-Konzept des Directory-View verwendet wird.

Die implementierten FUSE-Prozeduren sind relativ einfach aufgebaut, wie am Beispiel der Funktion `readdir()` in Listing 6.4 gezeigt wird. Die Parameter der FUSE-Schnittstelle werden in ZIB-DMS-interne Datenstrukturen umgewandelt und die entsprechende Directory-View Funktion über die ZIB-DMS-API gerufen. Während die Funktionen der FUSE-Schnittstelle mit Status-Rückgabewerten und Fehlercodes arbeiten, verwendet die ZIB-DMS-API C++-Exceptions, um Fehler- oder Ausnahmesituationen zu signalisieren. Diese müssen innerhalb der FUSE-Funktionen gefangen und in Fehlercodes umgewandelt werden, welche dann als Rückgabewerte dienen.

```
int readdir(const char *path,
            void *buf,
            fuse_fill_dir_t filler,
            off_t offset,
            struct fuse_file_info *fi)
{
    int ret = 0;
    vector<Hfn> result;
    vector<Hfn>::iterator it;
    try {
        Hfn pathname = Hfn( path );
        result = zibdmsapi->D_readdir( pathname );
        for( it=result.begin(); it!=result.end(); ++it ) {
            filler( buf, it->c_str(), NULL, 0 );
        }
    } catch ( NotFoundException &nfe ) {
        ret = -ENOENT;
    } catch ( PermissionDeniedException &pde ) {
        ret = -EACCES;
    } catch ( NotDirectoryException &nde ) {
        ret = -ENOTDIR;
    } catch( exception &e ) {
        LOG( fuseLog, ERROR, e.what() );
        ret = -ENOTSUP;
    }
    return ret;
}
```

Listing 6.4: `readdir()`-Funktion der FUSE-Proxy-Komponente

Das FUSE-Rahmenwerk sorgt dafür, dass alle Dateisystem-Operationen mit absoluten Pfaden bezüglich der Wurzel des Userspace-Dateisystems, in diesem Falle der ZIB-DMS-

[3] Beispielsweise werden *Extended Attributes* von den weit verbreiteten Archivierungs-Tools *tar* und *cpio* nicht verarbeitet und damit nicht archiviert [34].

Verzeichnishierarchie, gerufen werden. Somit können die Directory-View-Funktionen direkt mit den angegebenen Pfadnamen gerufen werden. Die einzige Ausnahme stellt hierbei das *linkinfo*-Argument der `symlink()`-Operation dar, das unverändert übergeben wird. Dieses Verhalten ist auf die im POSIX-Standard festgelegte Semantik dieser Operation zurückzuführen, die vorschreibt, dass die Zielangabe eines symbolischen Links als reine Zeichenkette interpretiert werden soll und keine Pfadprüfung erfolgen darf [30]. Dies ist dadurch begründet, dass mit diesem Konzept unter anderem Verweise zwischen verschiedenen Dateisystemen ermöglicht werden sollen und zum Zeitpunkt der Erstellung eines symbolischen Verweises dessen Ziel nicht zwingend existieren muss. Zudem soll es möglich sein, relative Pfade als Zielangaben für Symlinks zu benutzen.

Auch die hierarchische Verwaltungskomponente des ZIB-DMS bietet diese Semantik, wodurch die Erzeugung von symbolischen Verweisen auf Pfade außerhalb der Verzeichnishierarchie des ZIB-DMS möglich ist. Hierbei ergibt sich bei der Verwendung eines absoluten Pfades als Zielangabe allerdings ein Problem. Da die Zielangabe unverändert übergeben wird, bezieht sich deren absoluter Pfad auf das Wurzelverzeichnis des Betriebssystems. Sie wird, wie es die POSIX-Semantik fordert, unverändert gespeichert. Aber auch andere Clients oder Zugriffskomponenten können Symlinks mit absoluten Zielpfadangaben erzeugen, wobei sich der absolute Pfad auf die Wurzel der ZIB-DMS-Hierarchie bezieht. Diese Verweise sollten auch in einem über FUSE eingebundenen ZIB-DMS-Baum gültig bleiben. Beim Auslesen des Ziels eines Symlinks mit der Funktion `readlink()` ergibt sich für die FUSE-Komponente damit das Problem, dass nur anhand der Zielangabe nicht zwischen diesen beiden Fällen unterschieden werden kann.

Da das FUSE-Dateisystem seinen Einhängepunkt (*Mountpoint*) im System-Verzeichnisbaum kennt, wäre ein möglicher Lösungsansatz, beim Aufruf der `symlink()`-Operation mit einer absoluten Zielpfadangabe, die auf ein Objekt außerhalb des Einhängepunktes zeigt, die Zielangabe in einen relativen Pfad zu verwandeln und zu speichern. Das Problem dabei ist, dass der so erzeugte relative Pfad abhängig vom verwendeten Mountpoint ist. Ändert sich der Einhängepunkt wird der Verweis ungültig. Genau dieses Verhalten erwartet ein Benutzer bei der Verwendung einer absoluten Zielangabe jedoch nicht, sondern versucht dies dadurch eventuell sogar zu vermeiden. Daher verfolgt die FUSE-Komponente einen alternativen Ansatz: Die `symlink()`-Operation prüft, ob es sich bei der übergebenen Zielangabe um einen absoluten Pfad handelt und ob dieser auf ein Objekt außerhalb der ZIB-DMS-Hierarchie verweist. Trifft beides zu, wird kein symbolischer Link erstellt, sondern ein *Assimilate*-Prozess auf dem als `file://`-URL angegebenen Ziel angestoßen. Dadurch wird dieses als eigenständiges Datenobjekt im System registriert und der in der URL angebene absolute Pfad als *NSL* gespeichert. In allen anderen Fällen wird die `symlink()`-Funktion des Directory-View gerufen und die Zielangabe unbearbeitet übergeben. In der `readlink()`-Funktion wird der vom Directory-View erhaltenen Zielangabe immer der aktuelle Einhängepunkt des FUSE-Dateisystems vorangestellt. Damit werden sowohl die im ZIB-DMS gespeicherten absoluten Zielangaben, als auch relative Zielpfade korrekt aufgelöst.

6.4.2 Integration in das ZIB-DMS

Die Komponenten des ZIB-DMS werden im Quelltext als sogenannte *Module* umgesetzt. Jedes *Modul* wird als eine von der abstrakten Basisklasse `Module` abgeleitete Klasse repräsentiert. Sie stellt Funktionen zum Initialisieren, Ausführen und Beenden der Komponente bereit, definiert einen eigenen Logging-Kontext und übernimmt das Registrieren und Auswerten von Konfigurations-Optionen. Der ZIB-DMS-Server bietet zudem Mechanismen, um Abhängigkeiten zwischen Modulen zu definieren. Diese werden beim Start des Servers aufgelöst und die einzelnen Module in der daraus abgeleiteten Reihenfolge initialisiert und gestartet. Bei der ordnungsgemäßen Beendigung des Server-Prozesses werden die Module in umgekehrter Reihenfolge angehalten.

Auch die Fuse-Proxy-Komponente wird als eigenständiges Modul realisiert. Das FUSE-Dateisystem wird also als Teil-Komponente des ZIB-DMS-Servers ausgeführt. Dies kann aber nicht einfach durch die in Abschnitt 5.4 beschriebene `fuse_main()`-Funktion erfolgen, da diese zusätzlich zur Initialisierung des Dateisystems verschiedene Operationen ausführt, die das Verhalten und den Kontrollfluss des aufrufenden Prozesses beeinflussen und diesen zu einem Hintergrund-Prozess umwandeln. Diese Seiteneffekte sind für das in Abschnitt 6.5 beschriebene Client-Programm durchaus wünschenswert, nicht jedoch für die Integration in den Server-Prozess. Daher werden die in Abschnitt 5.3.2 beschriebenen notwendigen Schritte zur Initialisierung und Einbindung des Userspace-Dateisystems innerhalb der Initialisierungsfunktion des FUSE-Moduls einzeln ausgeführt. Zunächst wird die Belegung der Konfigurations-Parameter ausgewertet. Durch den Aufruf von `fuse_mount()` wird ein FUSE-Kommunikationskanal für den konfigurierten Einhängepunkt erzeugt. Auf dieser Basis wird mittels der Funktion `fuse_new()` eine neue FUSE-Dateisystem-Objektstruktur angelegt. Hier werden auch die Funktionszeiger auf die implementierten Dateisystem-Operationen übergeben. Abschliessend wird ein neuer Programm-Thread[4] erzeugt und gestartet, der die Funktion `fuse_loop()` ausführt. Damit ist das Userspace-Dateisystem vollständig und beginnt mit der Verarbeitung von Dateisystem-Anfragen. Bei Beendigung des Server-Programms wird die Funktion `done()` des FUSE-Moduls gerufen. Diese beendet durch den Aufruf von `fuse_exit()` die Verarbeitungsschleife und den erzeugten Thread. Durch Aufruf von `fuse_umount()` und `fuse_destroy()` werden die Dateisystem-Einbindung gelöst und die FUSE-Datenstrukturen abgeräumt.

6.5 FUSE-Client

Durch die FUSE-Proxy-Komponente wird es möglich, den Datenbestand des ZIB-DMS in den Verzeichnisbaum des Rechners, der den ZIB-DMS-Server ausführt, einzubinden.

[4] ZIB-DMS nutzt zur Implementierung und Verwaltung von Threads die *boost::thread*-Abstraktion der *boost*-Bibliothek [81], welche in der konkret vorliegenden Implementation mittels POSIX-Threads [11] umgesetzt werden.

Dieser kann wie ein gewöhnliches lokales Dateisystem verwendet und mittels bekannter Standard-Mechanismen für den entfernten Zugriff freigegeben werden. Dies ist jedoch nicht in allen Systemkonfigurationen möglich oder annehmbar. Daher ist eine direktere Möglichkeit für den entfernten Zugriff ohne den Einsatz zusätzlicher Software-Ebenen wünschenswert

Die FUSE-Komponente des Servers greift auf das System vermittelt über die ZIB-DMS-API zu. Diese kann durch die bereitgestellte CORBA-Anbindung (siehe Abschnitt 4.5 auf Seite 34) auch über Rechnergrenzen hinweg verwendet werden. Damit ist es einfach möglich, die Funktionalität der FUSE-Proxy-Komponente in das eigenständige Client-Programm *zibdmsfuse* auszulagern. Die hier implementierten Dateisystem-Operationen entsprechen im Wesentlichen denen der in Abschnitt 6.4 vorgestellten Server-Komponente, nur mit dem Unterschied, dass die Funktionen der ZIB-DMS-API auf einer CORBA-Objekt-Referenz gerufen werden.

Das ZIB-DMS verwendet zur Implementierung der CORBA-Mechanismen die C++-Variante der ORB-Implementation *omniORB4* [94]. Diese stellt einen IDL-Compiler und ein ORB-Laufzeitsystem in Form einer dynamischen Programmbibliothek zur Verfügung. Die aus der IDL-Definition der ZIB-DMS-API erzeugten Quelldateien werden kompiliert und in die *libzibdms*-Bibliothek aufgenommen, ebenso die Hilfsfunktionen zur Umwandlung der internen Datenstrukturen in CORBA-Datentypen. Sowohl das Server-Programm, als auch der FUSE-Client werden gegen diese Bibliothek gelinkt und können diese Funktionalität somit nutzen.

Um Funktionen auf einem entfernten CORBA-Objekt rufen zu können, muss eine Objekt-Referenz auf dieses Objekt vom ORB-Laufzeitsystem bezogen werden. Zum Einen kann dies durch direkte Adressierung des Rechners und des fraglichen Objektes erfolgen, zum Anderen unter Verwendung von CORBA-Diensten, wie dem *Naming-Service* [42] für die Lokalisierung von Objekten anhand von Bezeichnern oder dem *Trading-Service* [40] für ein Auffinden anhand von Objekt-Eigenschaften. Der ZIB-DMS-Sever unterstützt in der aktuellen Version nur die direkte Adressierung, so dass beim Aufruf des *zibdmsfuse*-Clients neben dem gewünschten Einhängepunkt auch die Adresse eines ZIB-DMS-Serversystems angegeben werden muss. Um den Vorgang dieses Verbindungsaufbaus möglichst flexibel und erweiterbar zu gestalten, wird der Erwerb einer CORBA-Objekt-Referenz durch die Factory-Klasse [23] `CorbaZIBDMSAPIFactory` realisiert. Bei deren Instantiierung wird die Adresse des Servers und eine Referenz auf die ORB-Laufzeitumgebung übergeben. Beide Angaben werden innerhalb der Methode `getCorbaZIBDMSAPI()` verwendet, um die Verbindung mit dem ZIB-DMS-Server herzustellen, eine CORBA-Referenz auf das entfernte `CorbaZIBDMSAPI`-Objekt zu erhalten und diese als Rückgabewert zurückzuliefern.

Der zur Behandlung der CORBA-Aufrufe zusätzlich notwendige Code soll so weit wie möglich von der Implementation der Dateisystem-Operationen getrennt werden, um die Übersichtlichkeit und Wartbarkeit des Quelltextes zu verbessern. Dies wird durch den Einsatz eines Singleton-Objektes [23] der Klasse `ZibdmsProxy` erreicht. Diese defi-

```
vector<Hfn>
    ZibdmsProxy::D_readdir( const Hfn& pathname )
    throw ( CORBA::SystemException,
            PermissionDeniedException,
            NotFoundException,
            NotDirectoryException )
    {
        vector<Hfn> ret;
        try {
            C_HfnList_var res;
            m_zibdmsapi->D_readdir( pathname.c_str(), res );
            for( unsigned int i=0; i<res->length(); ++i ) {
                ret.push_back( Hfn(res[i]) );
            }
        } catch( CorbaZIBDMSAPI::PermissionDenied &e ) {
            throw PermissionDeniedException( CORBA::string_dup(e.error_msg) );
        } catch( CorbaZIBDMSAPI::NotFound         &e ) {
            throw HfnNotFoundException( CORBA::string_dup(e.error_msg) );
        } catch( CorbaZIBDMSAPI::NotDirectory     &e ) {
            throw NotDirectoryException( CORBA::string_dup(e.error_msg) );
        }
        return ret;
    }
```

Listing 6.5: `D_readdir()`-Funktion der `ZibdmsProxy`-Klasse

niert eine zur ZIB-DMS-API weitestgehend identische Schnittstelle, so dass die FUSE-Operationen auf dem Proxy-Objekt ebenso operieren können wie es innerhalb der Server-Komponente möglich ist. Bei der Instantierung dieses Objektes wird eine Referenz auf die `CorbaZIBDMSAPIFactory` übergeben, die genutzt wird, um eine CORBA-Referenz auf die entfernte ZIB-DMS-API zu beziehen und gegebenenfalls zu erneuern. Innerhalb der Funktionen der Proxy-Klasse werden die verwendeten ZIB-DMS-Datenstrukturen in CORBA-Datentypen umgewandelt und die korrespondierende Funktion auf dem entfernten Objekt gerufen. Die Ergebnisse dieses Aufrufes werden wiederum in interne Datenstrukturen umgewandelt und zurückgeliefert. Diese Konvertierung umfasst auch eventuell geworfene Exception-Objekte. Listing 6.5 zeigt den Aufbau dieser *Wrapper*-Funktionen am Beispiel der `readdir()`-Prozedur.

Wenn der `zibdmsfuse`-Client gestartet wird, werden zunächst die ORB-Laufzeitumgebung und das FUSE-Rahmenwerk initialisiert. Mit der beim Aufruf übergebenen Server-Adresse wird ein `CorbaZIBDMSAPIFactory`-Objekt instantiiert, welches von dem `ZibdmsProxy`-Singleton verwendet wird, um die Verbindung zu dem ZIB-DMS-Server herzustellen. War der Verbindungsaufbau erfolgreich, wird – wie in Listing 5.8 auf Seite 60 beispielhaft gezeigt – die Funktion `fuse_main()` gerufen. Damit wird das Client-Programm wird zu einem Hintergrund-Prozess und beginnt damit, Dateisystem-Anfragen zu bearbeiten. Die Einbindung wird, wie bei jedem FUSE-Dateisystem, durch Aufruf des Hilfsprogramms `fusermount` gelöst und das Client-Programm damit beendet.

7 Leistungsmessungen

Das ZIB-DMS ist ein experimentelles Forschungs-System, das sich in Entwicklung befindet und daher zum aktuellen Zeitpunkt noch nicht für den Produktiv-Einsatz bereit ist. Um dennoch einen Eindruck über das aktuelle Leistungsverhalten des Systems zu bekommen, werden verschiedene Leistungsmessungen durchgeführt, die in diesem Kapitel beschrieben werden. Dabei wird versucht, den Einfluß der verschiedenen Ebenen der Software-Architektur und der zentralen Komponenten des Systems auf die Gesamtleistung zu ermitteln.

7.1 Messumgebung

Das Erstellen aussagekräftiger Leistungsmessungen von Dateisystemen ist kein triviales Unterfangen. Aufgrund der engen Bindung an das Betriebssystem werden die Messergebnisse durch dessen Zustand und Verhalten stark beeinflußt. Diese Einflüsse lassen sich zum einen schwer abschalten und sind zum anderen in ihrer Gesamtheit schwierig zu reproduzieren. Allerdings nehmen diese Effekte mit wachsendem Zeitverbrauch der einzelnen Dateisystem-Operationen stark ab, so dass signifikante Verfälschungen der Messungen weniger bei verteilten und Netzwerk-Dateisystemen, als bei Dateisystemen für lokale Datenträger auftreten. Umfangreiche Benchmark-Tools wie *bonnie++* [80] oder *Iozone* [89] nehmen ausgiebige Messungen auf Dateisystemen und Speichermedien vor und stellen die dabei erzielten Ergebnisse anhand verschiedender Kennzahlen dar. Viele der Tests, die diese Programme durchführen, sind auf '*vollwertige*' Dateisysteme ausgelegt, so dass diese einige Operationen und Anwendungs-Szenarien verwenden, die vom ZIB-DMS nicht unterstützt werden. Aus diesen Gründen wird eigens ein einfaches Mess-Programm implementiert. Ein beim Aufruf des Programmes übergebener Pfad wird als Zielpfad für die vorgenommenen Tests verwendet. Die Systemaufrufe zum Anlegen und Löschen von Dateien beziehungsweise Verzeichnissen (`mknod()`, `mkdir()`, `unlink()`, `rmdir()`), zum Umbenennen von Dateien (`rename()`) sowie zum Auslesen von Verzeichnissen und Dateiattributen (`readdir()`, `stat()`) werden unterhalb des Zielpfades mit automatisch erzeugten Pfadnamen gerufen. Mit Ausnahme von `readdir()` werden alle Operationen mehrfach in einer Schleife aufgerufen, die mit 1000, 3000, 5000, 7000, 9000 und 10.000 Durchläufen ausgeführt wird. Gemessen wird dabei das Zeitintervall vom Anstoß einer Iteration bis zu ihrer vollständigen Abarbeitung. Die `readdir()`-Operation wird auf ein Verzeichnis angewendet, das der Anzahl der Iterationen entsprechend vie-

le Dateien enthält. Hier wird die verstrichene Zeit vom Aufruf bis zur Rückkehr des Systemaufrufs gemessen. Um den Einfluss der Verzeichnistiefe auf das Zeitverhalten der Operationen zu ermitteln, werden zusätzlich Messungen mit einer fixen Anzahl von 1000 Iterationen und einer variablen Verzeichnistiefe von 0, 1, 2, 6, 10 und 20 durchgeführt. Alle Zeitmessungen werden auf Mikrosekunden genau durchgeführt. Bei der weiteren Verarbeitung werden die Ergebnisse aber in Millisekunden und Sekunden umgerechnet und gerundet.

Die Messungen werden auf einem *Dell Precision 530* Rechnersystem mit einem *Intel XEON 2.2 GHz* Prozessor mit aktiviertem Hyperthreading, 1 GB Hauptspeicher, einer 80 GB SATA Festplatte und einer Gigabit-Ethernet Netzwerkanbindung durchgeführt. Als Betriebssystem kommt eine *GNU/Debian Linux*[1]-Distribution mit einem SMP-Linux-Kernel der Version 2.6.21 und der *binutils*-Version 2.18.0.2008 zum Einsatz. Für die Übersetzung der Quelltexte wurde die *gcc*-Compiler-Suite in der Version 4.1, die *libc* in der Version 2.7, das FUSE-Framework in der Version 2.7.0 und das *SUN Java JDK* in der Version 1.6.0 benutzt. Um die Seiteneffekte der restlichen Systemaktivität auf die Messungen zu minimieren, wurden sämtliche entbehrliche Dienste inklusive aller Protokollierungs-Dienste auf dem System abgeschaltet. Für die netzwerkbasierten Mess-Szenarien wird ein weiteres, identisches Rechnersystem verwendet, das über die Netzwerk-Infrastruktur des *ZIB* über eine Gigabit-Ethernet-Verbindung mit dem anderen Test-System verbunden ist. Auf dem *Server*-Testsystem wird zur Durchführung der NFS-Messungen zusätzlich noch der NFS-Kernel-Server ausgeführt, der zur Standardinstallation der verwendeten Linux-Distribution gehört.

7.2 Vorgenommene Messungen

Innerhalb der dargestellten Umgebung und unter Verwendung der beschriebenen Software werden Messungen gemäß den im Folgenden beschriebenen Szenarien durchgeführt und mehrfach wiederholt. In Anhang A sind sowohl die Ergebnisse der einzelnen Durchläufe, als auch der Mittelwert über allen Messdurchgängen sowie die mittlere Abweichung von diesem Mittelwert für jede Mess-Konstellation aufgelistet. Durch die im Großen und Ganzen geringe mittlere Abweichung sind die Mittelwerte hinreichend aussagekräftig, so dass die hier abgedruckten graphischen Darstellungen der Ergebnisse auf den Mittelwerten basieren.

7.2.1 Vergleich mit dem XtreemFS

Um die Gesamtleistung der aktuellen Version des ZIB-DMS einordnen zu können, werden Vergleichs-Messungen zu dem in Kapitel 3 vorgestellten XtreemFS [28, 27] angestellt.

[1] http://www.debian.org

Zwar ist das XtreemFS anders als ZIB-DMS als verteiltes Grid-Dateisystem konzipiert, die bei den Messungen verwendeten Operationen operieren aber nur auf den Datenobjekten als Ganzes und nehmen keine Manipulation an deren Inhalt vor. Zudem wird der Zugriff auf das XtreemFS ebenfalls mittels FUSE realisiert und die logische Struktur des Datenbestandes in einem Metadaten-Katalog (*MRC*) gespeichert. Insofern kann das XtreemFS für die vorgesehenen Messungen durchaus für einen Vergleich der Leistungsdaten herangezogen werden.

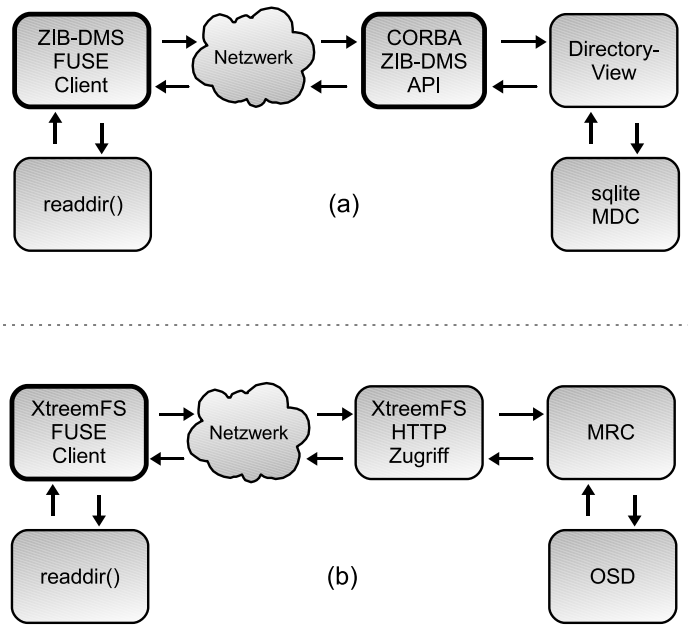

Abbildung 7.1: Schematischer Aufbau der Messungen am ZIB-DMS und XtreemFS: *(a)* Ein über den FUSE-Client eingebundener ZIB-DMS-Server mit *sqlite*-MDC. *(b)* Ein XtreemFS Server mit einer lokaler MRC- und OSD-Instanz ebenfalls über ein FUSE-Dateisystem auf dem Client-System eingebunden.

Der schematische Aufbau des für diese Messungen verwendeten Versuchsaufbaus ist in Abbildung 7.1 dargestellt. Da in der aktuellen Version des ZIB-DMS keine vollständige Implementation der P2P-verteilten MDC-Komponente vorliegt, können die Messungen in Szenario *(a)* nur auf einer einzelnen Server-Instanz mit einem lokalen *sqlite*-MDC durchgeführt werden. Diese wird auf einem Server-System ausgeführt und auf dem Client-System mittels des FUSE-Client-Programms eingebunden, das über die Funktionen der Directory-View-Komponente auf den MDC zugreift. Die Kommunikation erfolgt dabei über die CORBA-Mechanismen der ZIB-DMS-API. Die XtreemFS-Messungen in Szenario *(b)* werden zu Vergleichszwecken ebenfalls auf einer einzelnen Server-Instanz unter Verwendung einer ZIB-internen Testversion des XtreemFS-Server- und Client-Programms durchgeführt. Die Verzeichnishierarchie des XtreemFS wird auf dem Client-Rechner ebenfalls durch ein FUSE-Dateisystem eingebunden. Die Kommunikation mit dem Server, der jeweils eine lokale Instanz der MRC- und OSD-Dienste ausführt, erfolgt

durch den Austausch von serialisierten C-Datenstrukturen über das HTTP-Protokoll.

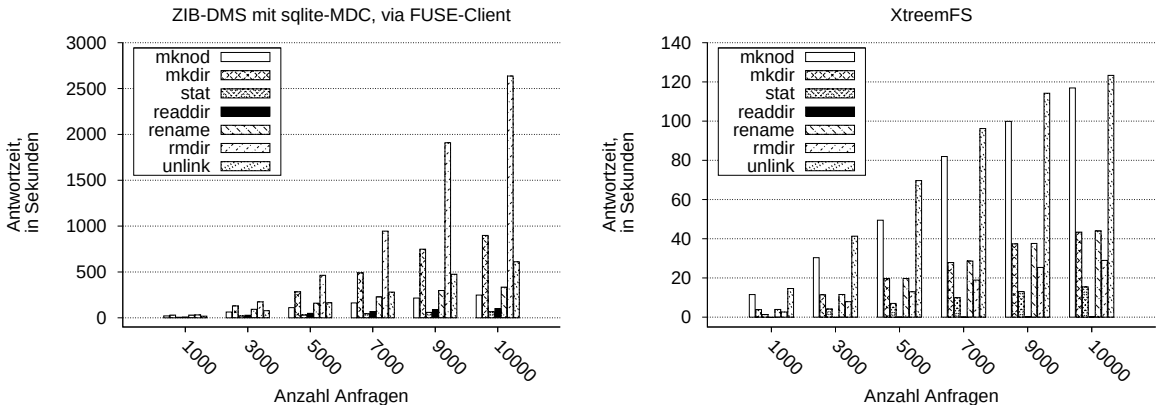

Abbildung 7.2: Ergebnisse der XtreemFS-Vergleichsmessungen

Die gemessene Leistungsfähigkeit der beiden Systeme wird in Abbildung 7.2 separat dargestellt. Hier fällt sofort das deutlich schlechtere Abschneiden des ZIB-DMS ins Auge. Besonders gravierend ist die sich abzeichnende exponentielle Entwicklung des gemessenen Zeitverbrauchs einiger Operationen. Dies wird auch in der direkten Gegenüberstellung der Ergebnisse in Abbildung 7.3 deutlich, die ein exponentielles Wachstum vor allem bei den Funktionen `unlink()` und `rmdir()`, in etwas abgeschwächter Form aber auch bei der Funktion `mkdir()` zeigen. Bei den übrigen Operationen ist ein unterschiedlich starkes, aber lineares Anwachsen zu verzeichnen. Der stärkste Anstieg zeichnet sich hier bei den Funktionen `readdir()` und `rename()` ab. Bei diesen insgesamt großen Unterschieden überraschen die relativ nahe beieinander liegenden Werte der `mknod()`-Funktion. Diese sind durch den unterschiedlichen Aufwand zu erklären, der bei Ausführung dieser Operation auf beiden Systemen entsteht. Da das XtreemFS als vollwertiges Dateisystem nicht nur Metadaten, sondern auch die Daten selbst speichert, wird neben einem neuen Eintrag in dessen MRC-Katalog noch ein entsprechender Abgleich mit dem OSD vorgenommen. Wohingegen das ZIB-DMS hier nur ein neues Datenobjekt im MDC registrieren muss. Somit gestaltet sich diese Operation auf dem XtreemFS etwas aufwändiger, als beim ZIB-DMS.

Das insgesamt wesentlich schlechtere Abschneiden des ZIB-DMS kann zum einen darauf zurückgeführt werden, dass die Entwicklung des Systems noch nicht abgeschlossen ist und die Messungen auf der aktuell verfügbaren experimentellen Version durchgeführt wurden. Zum anderen handelt es sich beim ZIB-DMS um ein sehr umfangreiches und komplexes System. Wie in Kapitel 4 dargestellt, ist die grundlegende Organisation der Datenobjekte im ZIB-DMS eine attributbasierte Struktur. Der Zugriff über eine Verzeichnishierarchie ist nur einer der verschiedenen Verwaltungsmechanismen, die das System bietet. Daher muss die hierarchische Struktur im Gegensatz zum XtreemFS auf die attributbasierte Struktur des MDC abgebildet werden, wodurch ein wesentlicher Mehraufwand entsteht. Die gemessenen Ergebnisse zeigen, dass die aktuelle Umsetzung des ZIB-DMS hier viel Raum für Verbesserungen und Optimierungen lässt. Um diese

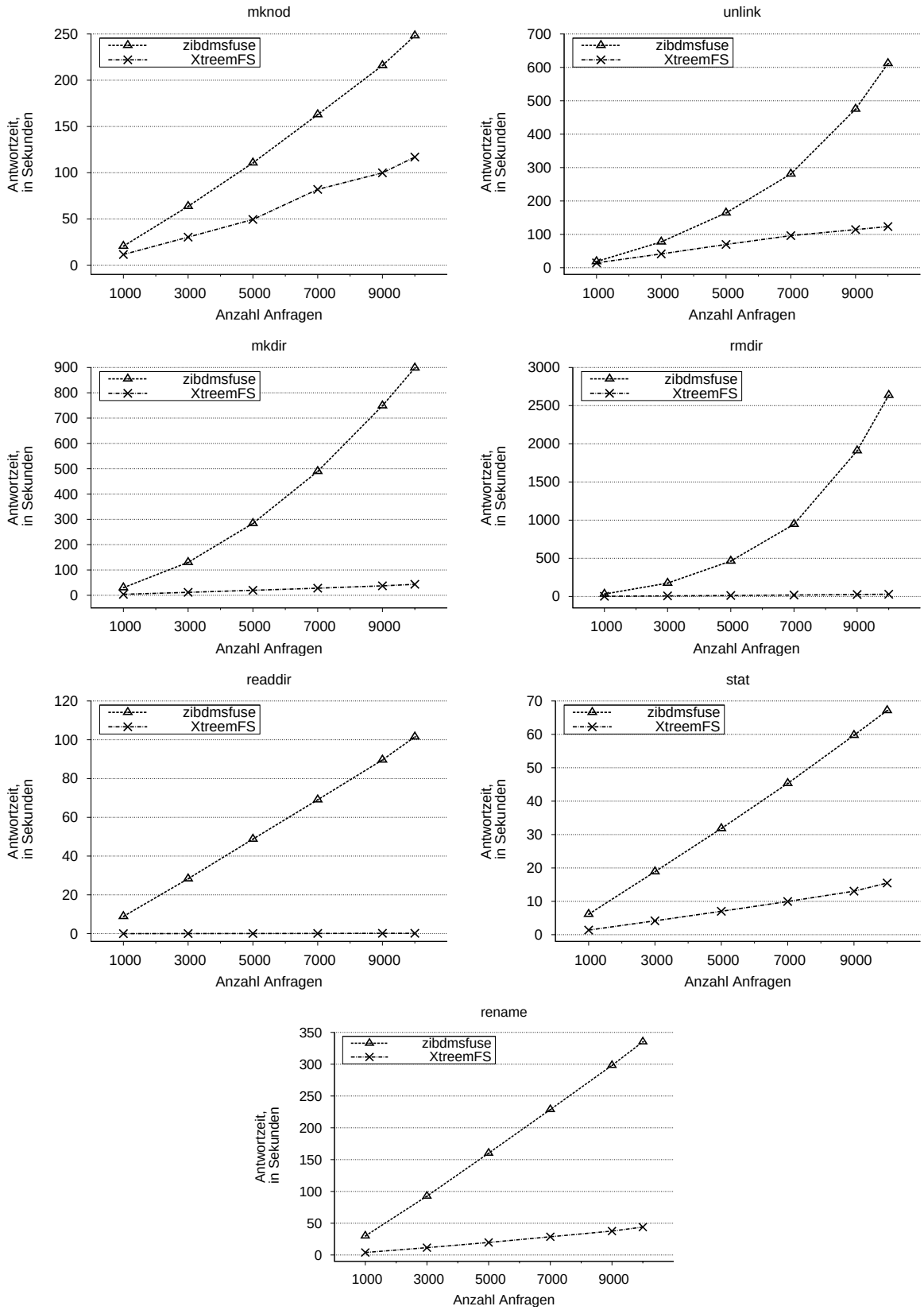

Abbildung 7.3: Gegenüberstellung der XtreemFS-Vergleichsmessungen

Einschätzung zu untermauern, werden im Folgenden weitere Messungen durchgeführt, anhand derer der Einfluß verschiedener Software-Komponenten auf die Gesamtleistung des Systems untersucht werden soll.

7.2.2 Overhead des FUSE-Rahmenwerks

Durch die Verlagerung der Implementation eines Dateisystems in ein eigenständiges Userspace-Programm werden während der Abarbeitung einer Dateisystem-Operation zusätzliche Kontext- und Modewechsel[2] nötig (siehe Abbildung 5.4 auf Seite 48). Dies führt zu einem höheren Zeit-Overhead, als es bei Kernelspace-Dateisystemen der Fall ist. Daher ist zunächst die Auswirkung dieses Overheads auf die Leistungsfähigkeit eines FUSE-Dateisystems im Allgemeinen und auf die des ZIB-DMS im Speziellen von Interesse.

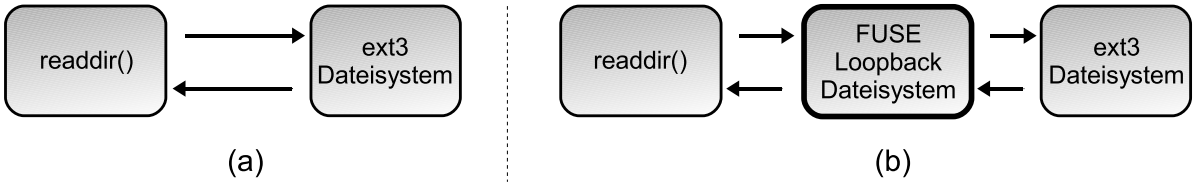

Abbildung 7.4: Versuchsaufbau zur Ermittlung des FUSE-Overheads: *(a)* Die Messungen werden direkt auf einem *ext3*-Dateisystem durchgeführt. *(b)* Die Messungen werden auf einem speziellen FUSE-Dateisystem ausgeführt, das alle Operationen durch die entsprechenden Systemaufrufe auf einem Teilbaum des *ext3*-Dateisystems realisiert.

Um diesen Mehraufwand an Zeit zu messen, wird ein spezielles FUSE-Loopback-Dateisystem implementiert. Dieses realisiert die Dateisystem-Operationen ähnlich wie in dem auf Seite 60 dargestellten Beispiel durch den Aufruf der entsprechenden Systemrufe. Beim Einbinden dieses Dateisystems wird ein Verzeichnispfad übergeben, der dem beim Aufruf einer FUSE-Operation übergebenen Pfad als Präfix vorangestellt wird. Dadurch kann ein Teilbaum der Verzeichnishierarchie über den Einhängepunkt des Loopback-Dateisystems erreichbar gemacht und dieses so einem anderen Dateisystem vorgeschaltet werden.

Bei der Durchführung der Messungen wird zum Vergleich das im Kernel integrierte *ext3*-Dateisystem [13, 77] herangezogen. Abbildung 7.4 zeigt den dabei verwendeten Aufbau.

[2] Das Umschalten des Betriebssystems zwischen zwei Prozessen wird als *Kontextwechsel* bezeichnet. Dabei muss der Zustand des aktuellen Prozesses gesichert und ausgelagert und der des neuen Prozesses geladen und wiederhergestellt werden. Das Umschalten des Betriebssystems in den privilegierten Kernel-Mode und zurück, wie es bei der Ausführung vieler Systemaufrufe geschieht, bezeichnet man als *Modewechsel*. Modewechsel erzeugen im Allgemeinen einen geringeren Zeit-Overhead als Kontextwechsel [64].

In Szenario *(a)* werden die Messungen zunächst auf einem leeren *ext3*-Dateisystem, auf einer separaten Festplatten-Partition, durchgeführt. Diesem wird anschließend in Szenario *(b)* das FUSE-Loopback-Dateisystem vorgeschaltet und die Messungen wiederholt. Vor jedem Messdurchgang wird das *ext3*-Dateisystem neu initialisiert.

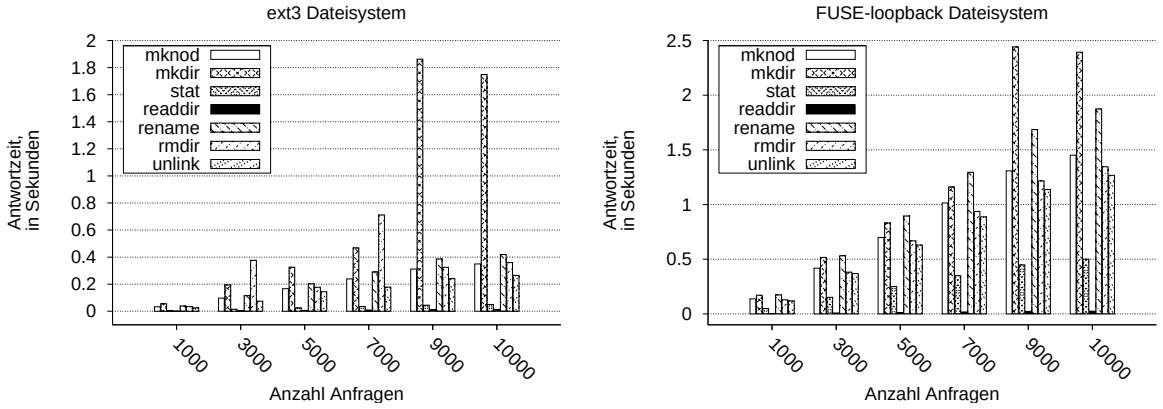

Abbildung 7.5: Ergebnisse der Messungen zum FUSE-Overhead.

In Abbildung 7.5 sind die Ergebnisse dieser Messungen im Einzelnen dargestellt. Die linke Balken-Grafik zeigt die Ergebnisse der Tests auf dem *ext3*-Dateisystem (Szenario *(a)*). Auf der rechten Seite sind die Resultate mit vorgeschaltetem Loopback-Dateisystem (Szenario *(b)*) zu sehen. Jede Operation wird als eigener Balken, gruppiert nach der verwendeten Anzahl von Anfragen, dargestellt. Man erkennt, dass die benötigte Zeit bei beiden Szenarien für fast alle Operationen mit wachsender Anzahl im Großen und Ganzen linear steigt. Die Ausnahme bildet die Operation `mkdir()`. Hier ist bei den Messungen auf dem Loopback-Dateisystem bei einer Anzahl von 9000 Anfragen ein sprunghafter Anstieg zu erkennen. Da ein solcher Sprung auch bei den Messungen auf dem *ext3*-Dateisystem allein auftritt, ist dieses Verhalten aber weniger im FUSE-Rahmenwerk begründet, sondern vielmehr als Charakteristikum der *ext3*-Implementation zu deuten. Dies bestätigt auch der `mkdir()`-Graph in Abbildung 7.6.

Für einen übersichtlicheren Vergleich der Ergebnisse, werden die Resultate der beiden Szenarien in Abbildung 7.6 für jede gemessene Operation als Graphen gegeneinander aufgetragen. Hier fällt zunächst der `rmdir()`-Graph der *ext3*-Messungen in Auge, der bei 3000 und 7000 Anfragen stark ausbricht. Dies ist auf eine schlechte Qualität des verwendeten Mittelwertes zurückzuführen, da die Messungen dieser Funktion auf dem *ext3*-Dateisystem teils relativ stark abweichende Ergebnisse erzeugen. Diese weisen aber – wie den Einzelergebnissen in Anhang A zu entnehmen ist – keine breite Streuung auf, sondern liegen stets im selben höheren oder niedrigeren Wertebereich. Eine eingehende Überprüfung der Messwerkzeuge und der Versuchsanordnung ergab keine Anhaltspunkte, dass es sich hierbei um Messfehler handelt. Dieser Effekt konnte durch vielfache Wiederholung der Messungen reproduziert werden, ein Muster war dabei aber nicht zu erkennen. Die Schwankungen treten nur bei den Messungen auf dem *ext3*-Dateisystem direkt auf, nicht aber bei Messungen in denen andere Software-Ebenen vorgeschaltet

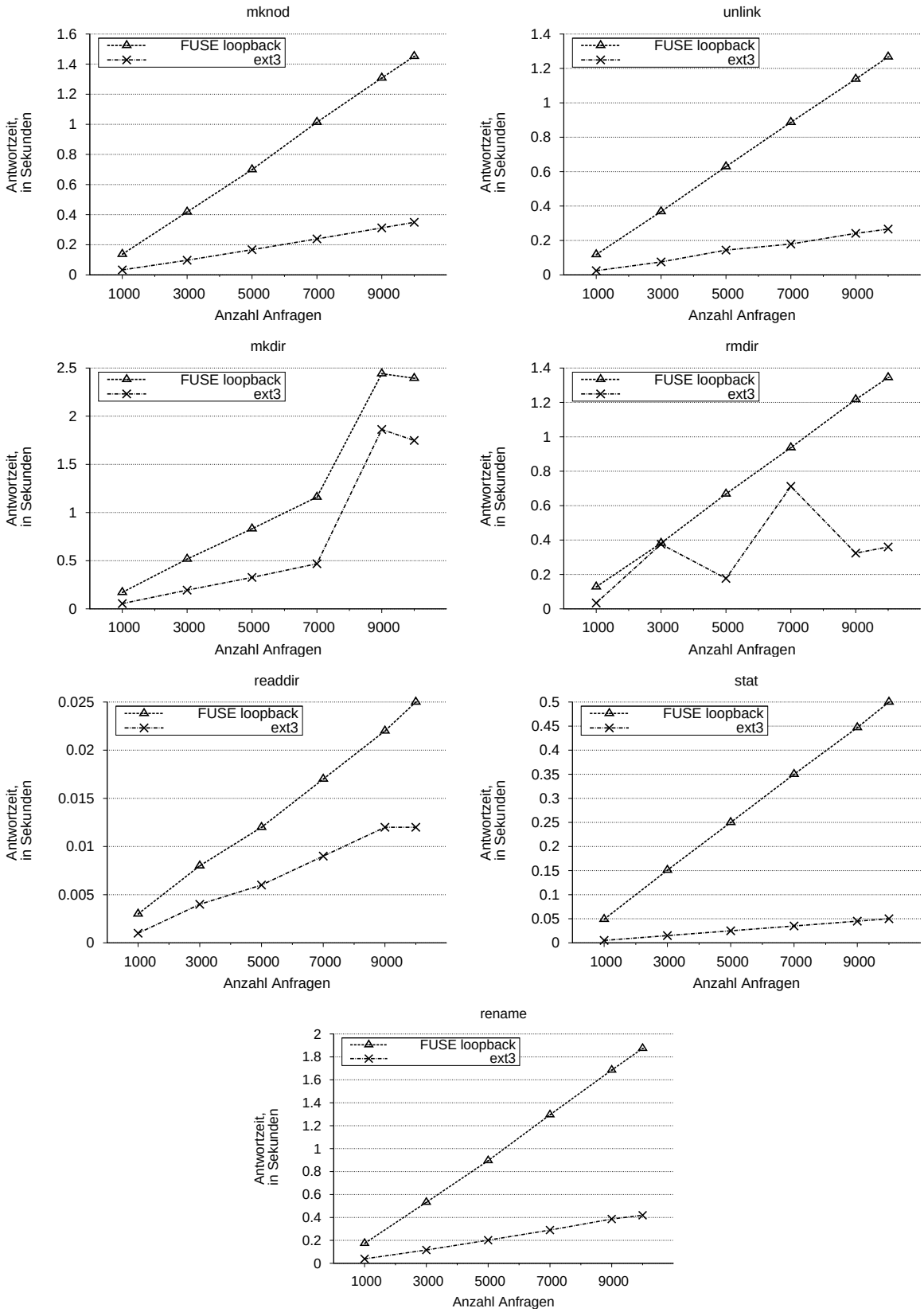

Abbildung 7.6: Gegenüberstellung der Messergebnisse zum FUSE-Overhead

sind. Diese Tatsache und das sporadische Auftreten der Schwankungen legen nahe, dass sie durch einen Optimierungs-Mechanismus dieses Dateisystems ausgelöst werden, der in allen anderen Versuchsanordnungen nicht greift beziehungsweise durch die ohnehin höheren Zugriffszeiten keine signifikanten Auswirkungen hat.

Insgesamt ist zu erkennen, dass der Overhead des FUSE-Rahmenwerks bei allen Operationen mit wachsender Anzahl von Anfragen zunimmt. Um die Signifikanz dieses Anstiegs einschätzen zu können, wurden zusätzliche Messungen auf einem über NFS exportierten *ext3*-Dateisystem durchgeführt. Die dabei erhaltenen Resultate werden in Abbildung 7.7 exemplarisch an den Operationen `mknod()` und `readdir()` zu den vorhandenen Ergebnissen in Relation gesetzt. Hier ist deutlich zu erkennen, dass der Remote-Zugriff über NFS sowohl bei einer Lese- als auch bei einer Schreib-Operation einen wesentlich gravierenderen zeitlichen Mehraufwand erzeugt, so dass der durch FUSE erzeugte Overhead kaum ins Gewicht fällt.

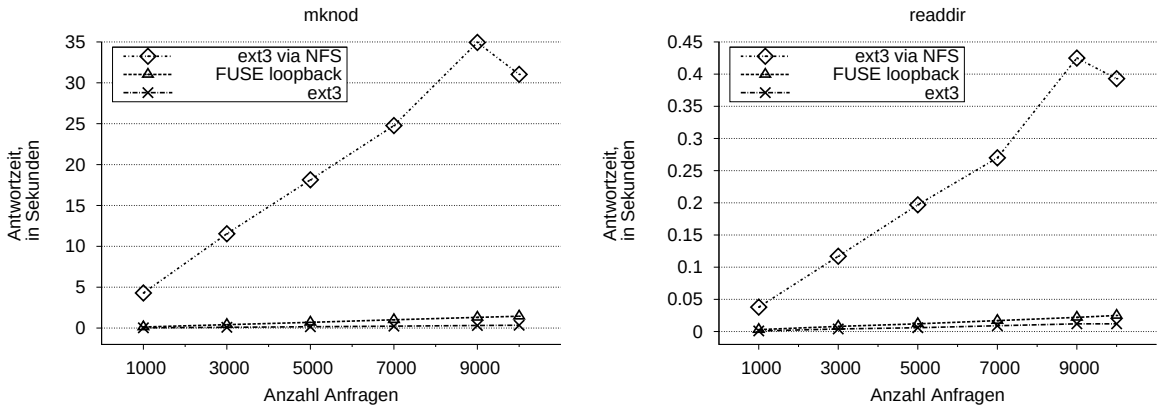

Abbildung 7.7: Vergleich des FUSE-Overheads mit NFS-Messungen

Aus diesen Ergebnissen wird deutlich, dass die Verwendung des FUSE-Rahmenwerks zur Implementierung von Dateisystemen für lokale Datenträger aufgrund der Leistungseinbußen im Vergleich zu Kernelspace-Dateisystemen als nicht besonders günstig anzusehen ist. Für die Realisierung von verteilten oder Netzwerk-Dateisystemen ist es hingegen durchaus gut geeignet, da hier der Overhead der Netzwerk-Kommunikation den des Rahmenwerks um ein Vielfaches übersteigt und so die in Kapitel 5 erörterten Vorteile überwiegen. Dies wird auch durch die relativ kleine Anzahl von Datenträger-Dateisystemen im Kontrast zu dem großen und ständig wachsenden Anteil an netzwerkbasierten Dateisystemen der in [72] geführten Liste bekannter FUSE-Dateisystem-Projekte bekräftigt.

7.2.3 Einfluß des MDC auf die Gesamtleistung

Bei Betrachtung des Aufbaus des ZIB-DMS (siehe Abbildung 6.1 auf Seite 64) wird klar, dass der Metadaten-Katalog als zentrale Komponente ausschlaggebend für die

Gesamtleitung des Systems ist. Maßgeblich ist aber auch die durch die hierarchische Verwaltungskomponente (siehe Abschnitt 4.2.1, Kapitel 4) realisierte Abbildung der attributbasierten Struktur des MDC in einen Verzeichnisbaum. Deshalb werden weitere Messungen zur Bestimmung des Einflusses dieser Komponenten auf das Leistungsverhalten des ZIB-DMS vorgenommen.

Zu diesem Zweck wird eine spezielle Variante der Directory-View-Komponente (siehe Abschnitt 4.4 in Kapitel 4) implementiert. Diese als *Dummy-Directory-View (DummyDV)* bezeichnete Komponente ersetzt die gewöhnliche Directory-View-Komponente und setzt deren Funktionalität[3], ähnlich zu dem in Abschnitt 7.2.2 erörterten Loopback-Dateisystem, durch Aufruf der korrespondierenden Systemaufrufe um, anstatt die Dienste der hierarchischen Verwaltungskomponente zu nutzen. Auch hier wird den dabei übergebenen Pfadnamen ein in der Konfiguration des Server-Programms festgelegtes Verzeichnis vorangestellt, wodurch die Umleitung der über die FUSE-Komponente verwendeten Operationen des Directory-View auf ein gewöhnliches Dateisystem ermöglicht wird. Damit kann zu Testzwecken ein ZIB-DMS-Server ohne Metadaten-Katalog ausgeführt werden. Dessen Leistungsfähigkeit kann dann mit der einer vollwertigen Server-Instanz verglichen und somit der Einfluß der Komponenten auf das Leistungsverhalten bestimmt werden.

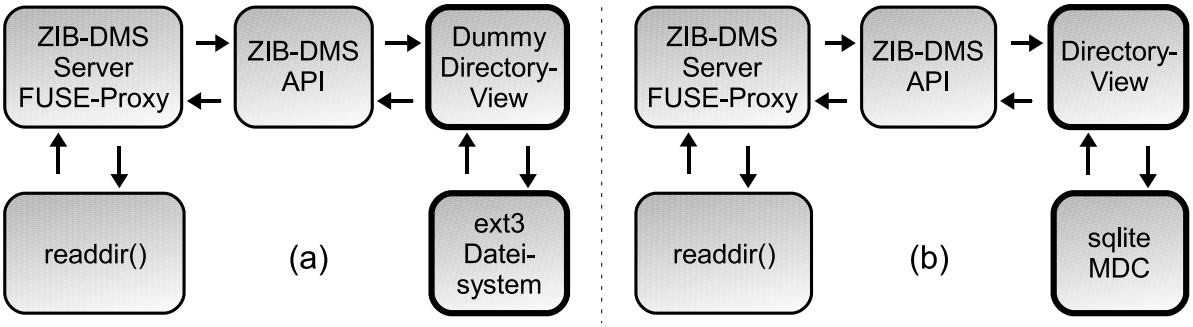

Abbildung 7.8: Versuchsaufbau zur Ermittlung des Leistungsverhaltens des ZIB-DMS-Servers: *(a)* Ein ZIB-DMS-Server ohne MDC, mit Dummy-Directory-View zur Speicherung der Testdateien auf einem *ext3*-Dateisystem. *(b)* Ein ZIB-DMS-Server in Standardkonfiguration mit einem *sqlite*-MDC.

Abbildung 7.8 zeigt die für diese Messungen verwendete Versuchsanordnung. In Szenario *(a)* wird ein ZIB-DMS-Server unter Verwendung der DummyDV-Komponente betrieben, deren Operationen auf ein *ext3*-Dateisystem umgeleitet werden. In Szenario *(b)* hingegen wird die gewöhnliche Directory-View-Komponente mit der *sqlite*-Variante des Metadaten-Katalogs verwendet. In beiden Fällen wird das ZIB-DMS über die FUSE-Komponente des Servers, die über die interne ZIB-DMS-API auf die Funktionen des

[3] Natürlich sind bei Verwendung dieser Komponente die erweiterten Verwaltungsmechanismen des ZIB-DMS nicht verfügbar.

Directory-View zugreift, in die lokale Verzeichnishierarchie eingebunden. Um eine Beeinflussung der Ergebnisse durch Netzwerkverkehr auszuschließen, werden die Messungen lokal auf dem Server-System durchgeführt.

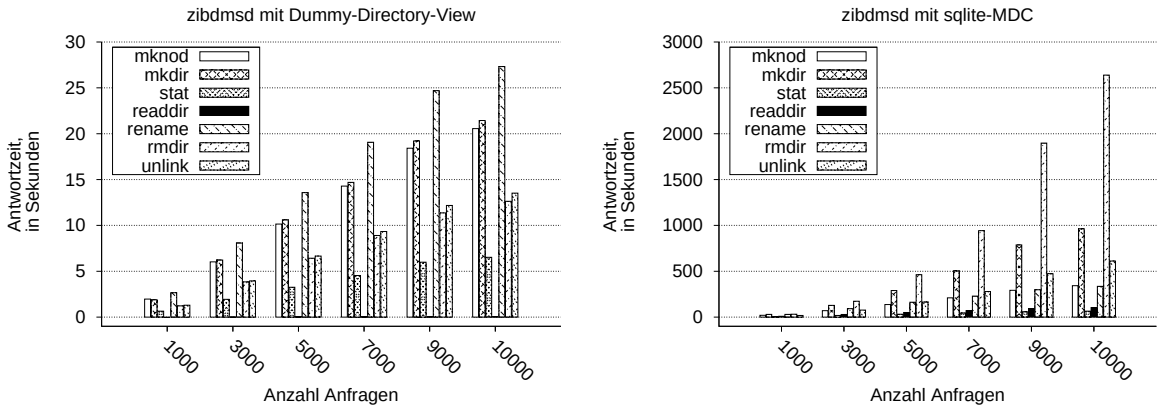

Abbildung 7.9: Ergebnisse der Messungen am ZIB-DMS-Server

Die Resultate sind in Abbildung 7.9 wieder als Balken-Grafiken abgedruckt und veranschaulichen den großen Einfluß des MDC auf die Gesamtleistung des Systems. Bei den auf der rechten Seite dargestellten Werten der Messungen mit dem *sqlite*-MDC zeichnet sich die selbe Entwicklung ab, wie sie auch in Abbildung 7.2 zu sehen ist. Positiv fällt aber auf, dass die Werte bei den hier gezeigten lokalen Messungen nur geringfügig niedriger sind, als bei den in Abbildung 7.5 angestellten Messungen am FUSE-Client. Dies spricht für einen geringen Overhead dieser Form des entfernten Zugriffs (siehe auch Abschnitt 7.2.4). Die linke Grafik in Abbildung 7.9 zeigt die Ergebnisse der Messungen mit dem Dummy-Directory-View. Diese zeigen einerseits einen deutlich geringeren Overhead, wobei die Zeitwerte im Vergleich zu den in Abbildung 7.5 gezeigten Messungen am FUSE-Loopback-Dateisystem um mehr als den Faktor zehn höher liegen, obwohl die Dateisystem-Anfragen analog zu diesem im Großen und Ganzen nur durch das ZIB-DMS hindurch an das *ext3*-Dateisystem weitergereicht werden. Andererseits verläuft die Entwicklung der Ergebnisse hier bei allen Operationen linear und bewegt sich in einem überschaubaren Rahmen, so dass eine Senkung des zeitlichen Mehraufwands, der durch die internen Mechanismen des ZIB-DMS entsteht, durch Optimierungen am Quelltext möglich ist.

Eine wesentlich gravierendere Auswirkung auf das zeitliche Verhalten hat offensichtlich die Verwendung des *sqlite*-MDC. Diese führt zu einem rapiden Anstieg der Zeitwerte bei allen Operationen, wie auch der in Abbildung 7.10 dargestellte direkte Vergleich der Messergebnisse zeigt. Dabei nimmt das Entfernen von Objekten mindestens doppelt so viel Zeit in Anspruch wie das Anlegen. Die auch hier zu beobachtende exponentielle Entwicklung bei den Operationen `unlink()`, `mkdir()` und vor allem `rmdir()` zeigt, dass Operationen die Änderungen an der Verzeichnis-Struktur vornehmen besonders langwierig sind. Dies lässt auf eine suboptimale Abbildung der Verzeichnishierarchie auf die

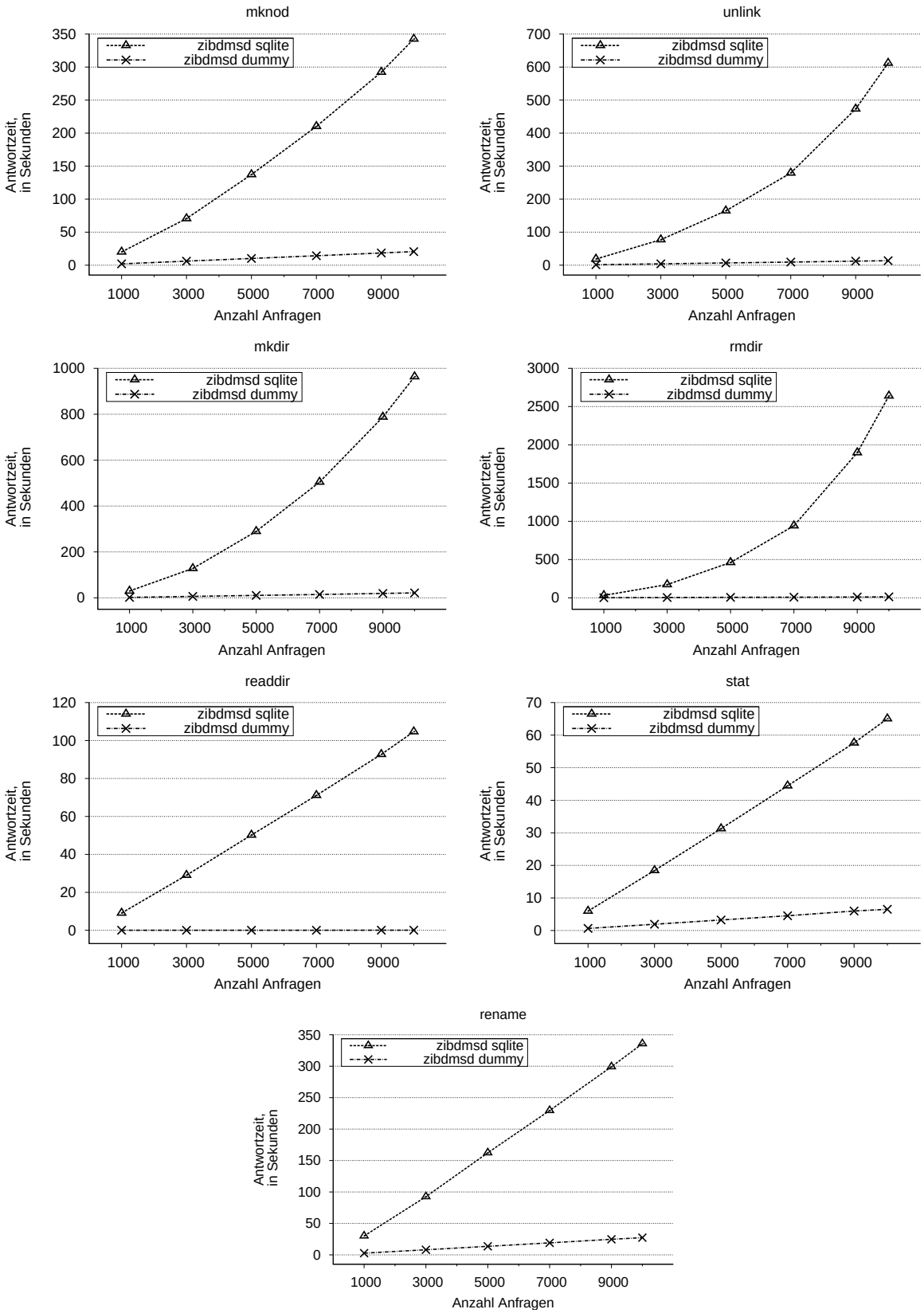

Abbildung 7.10: Gegenüberstellung der Messungen am ZIB-DMS-Server

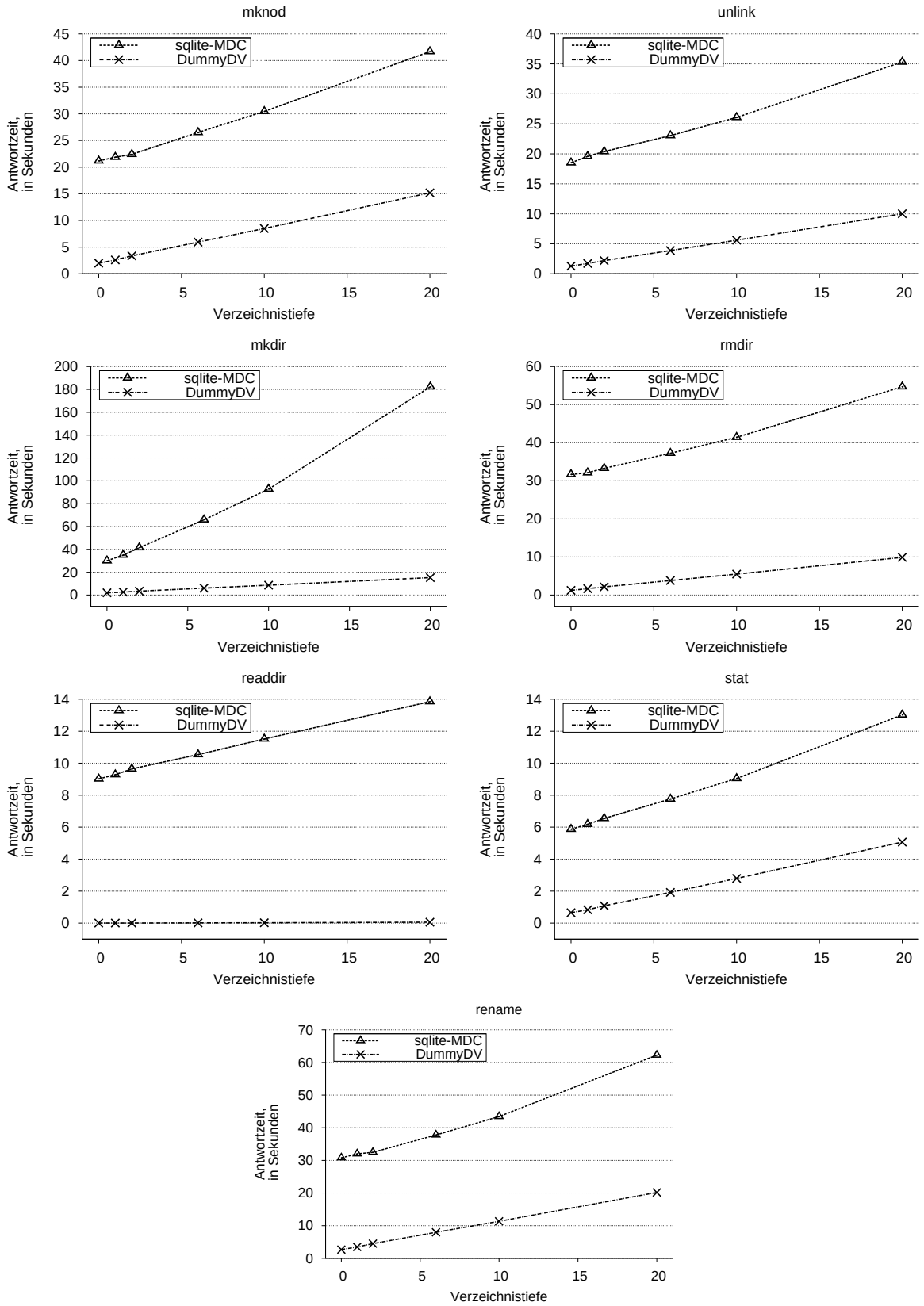

Abbildung 7.11: Gegenüberstellung der Messungen am ZIB-DMS-Server mit steigender Verzeichnistiefe

attributbasierte Struktur des MDC schließen. Das legen auch die in Abbildung 7.11 dargestellten Messungen mit wachsender Verzeichnistiefe nahe. Während bei allen anderen Messungen die verbrauchte Zeit mit wachsender Verzeichnistiefe leicht linear steigt, ist bei den ZIB-DMS Messungen ein stärkerer Anstieg zu verzeichnen. Vor allem bei den Funktionen `readdir()` und `mkdir()` ist ein drastischer Anstieg zu beobachten, wodurch obige Annahme bekräftigt wird. Ein weiterer möglicher Leistungshemmer ist das verwendete SQL-Schema des eingesetzten *sqlite*-MDC. Die attributbasierte Objekt-Struktur des MDC wird auf SQL-Tabellen abgebildet. Das dabei verwendete Schema ist ausschlaggebend für die Leistungsfähigkeit der MDC-Komponente. Ein ungünstig gestaltetes Schema kann erhebliche Preformance-Einbußen zur Folge haben.

Die Ergebnisse dieser Messungen zeigen, welchen hohen Einfluß die MDC-Komponente auf die Gesamtleistung des Systems ausübt. Zwar erzeugen auch die übrigen Komponenten des Systems einen nicht zu vernachlässigenden Overhead, der aber im Verhältnis zu dem der MDC-Komponente wesentlich weniger ins Gewicht fällt. Es zeigt sich so, dass hier ein deutlicher Optimierungsbedarf besteht.

7.2.4 Vergleich der Zugriffs-Varianten

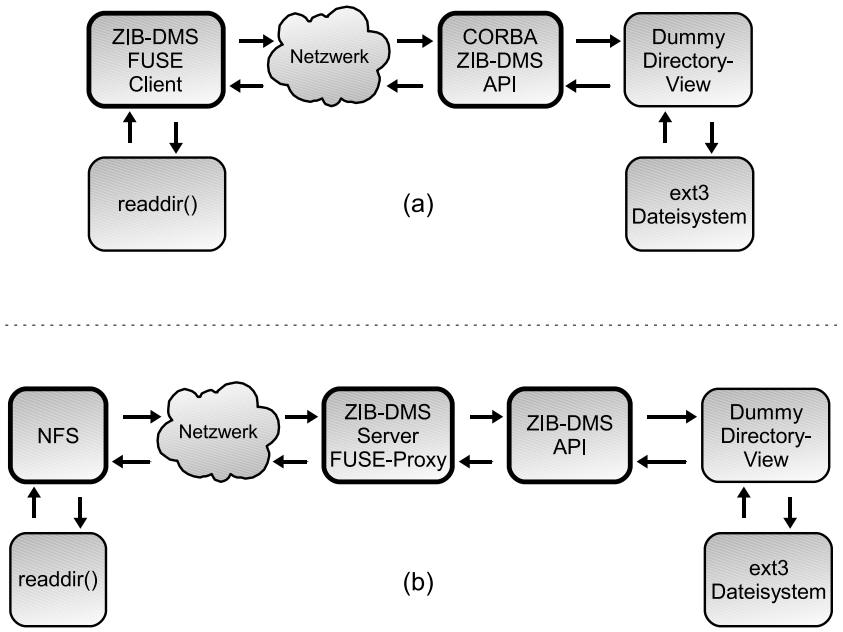

Abbildung 7.12: Versuchsaufbau für die Client-Messungen: *(a)* Einbindung des ZIB-DMS direkt über CORBA durch den FUSE-Client. *(b)* Einbindung des auf dem Server gemounteten ZIB-DMS über NFS.

Wie in Kapitel 6 beschrieben gibt es grundsätzlich zwei Möglichkeiten einen entfernten ZIB-DMS-Server unter Verwendung von FUSE in den Verzeichnisbaum eines Client-Rechners einzubinden: Durch den Re-Export der über die FUSE-Proxy-Komponente auf

dem Server-System eingebundenen Verzeichnishierarchie mittels Standard-Dateiserver-Lösungen oder direkt durch die Verwendung des FUSE-Client-Programms. Welche der beiden Varianten gewählt wird, hängt zunächst von den Gegebenheiten der jeweiligen Systeme und der Präferenz des Benutzers ab. Maßgeblich ist aber auch deren Leistungsfähigkeit. Daher werden Messungen zum Vergleich der Performance beider Zugriffsmöglichkeiten durchgeführt.

Abbildung 7.12 zeigt eine schematische Darstellung der dabei verwendeten Versuchsanordnungen. In Szenario *(a)* wird ein entfernter ZIB-DMS-Server über das FUSE-Client-Programm eingebunden. Die Verbindung wird über die CORBA-Anbindung der ZIB-DMS-API hergestellt. In Szenario *(b)* wird das ZIB-DMS über die FUSE-Proxy-Komponente in den Verzeichnisbaum des Servers eingebunden. Dieser Teilbaum wird dann mittels eines NFS-Servers exportiert und über das Netzwerk auf dem Client-System eingebunden. Um unerwünschte Nebeneffekte weitestgehend auszuschließen, wird in beiden Fällen eine ZIB-DMS Instanz mit dem im vorangegangenen Abschnitt vorgestellten Dummy-Directory-View verwendet, der die bei den Messungen erzeugten Daten auf einem *ext3*-Dateisystem speichert. Um Caching-Effekte zu vermeiden, wird in beiden Fällen das Caching von Dateiattributen deaktiviert.

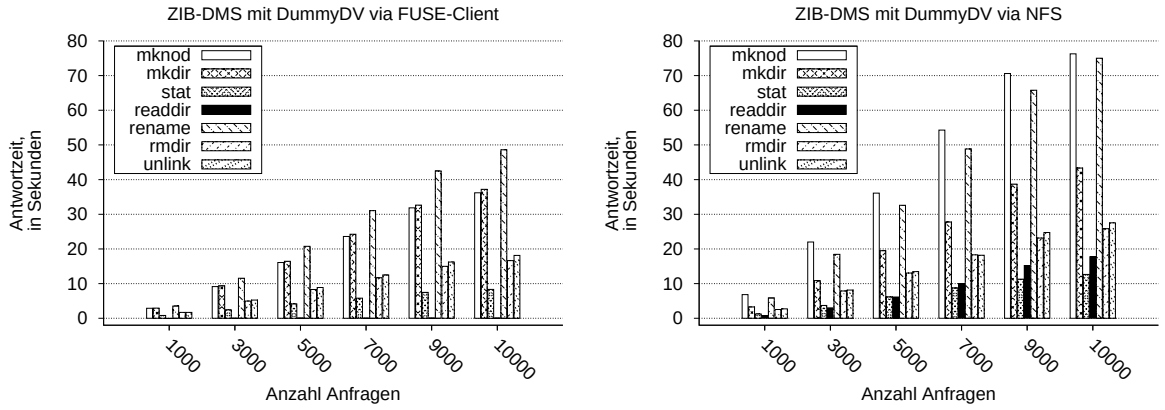

Abbildung 7.13: Ergebnisse der Client-Messungen

Die Ergebnisse dieser Messungen sind in Abbildung 7.13 im Einzelnen und in Abbildung 7.14 vergleichend dargestellt. Wie man sieht, schneiden die Ergebnisse der Messungen über NFS deutlich schlechter ab, als die mit dem FUSE-Client. Dies ist zum einen darauf zurückzuführen, dass der Zugriff über den FUSE-Client wesentlich direkter verläuft. Während hier die Client-Anwendung über das FUSE-Dateisystem und CORBA direkt auf der ZIB-DMS-API operieren, wird bei den NFS-Messungen über einen NFS-Client auf ein vom NFS-Server exportiertes Verzeichnis zugegriffen, das wiederum als Einhängepunkt für das FUSE-Dateisystem des lokalen ZIB-DMS-Servers dient. Somit entsteht eine zusätzliche Software-Ebene, die zeitlichen Mehraufwand bedeutet. Zum anderen verläuft die Zusammenarbeit zwischen FUSE und NFS, wie in Kapitel 6 erwähnt, in der aktuellen Version nicht reibungslos, so dass der zeitliche Mehraufwand auch hier begründet sein könnte.

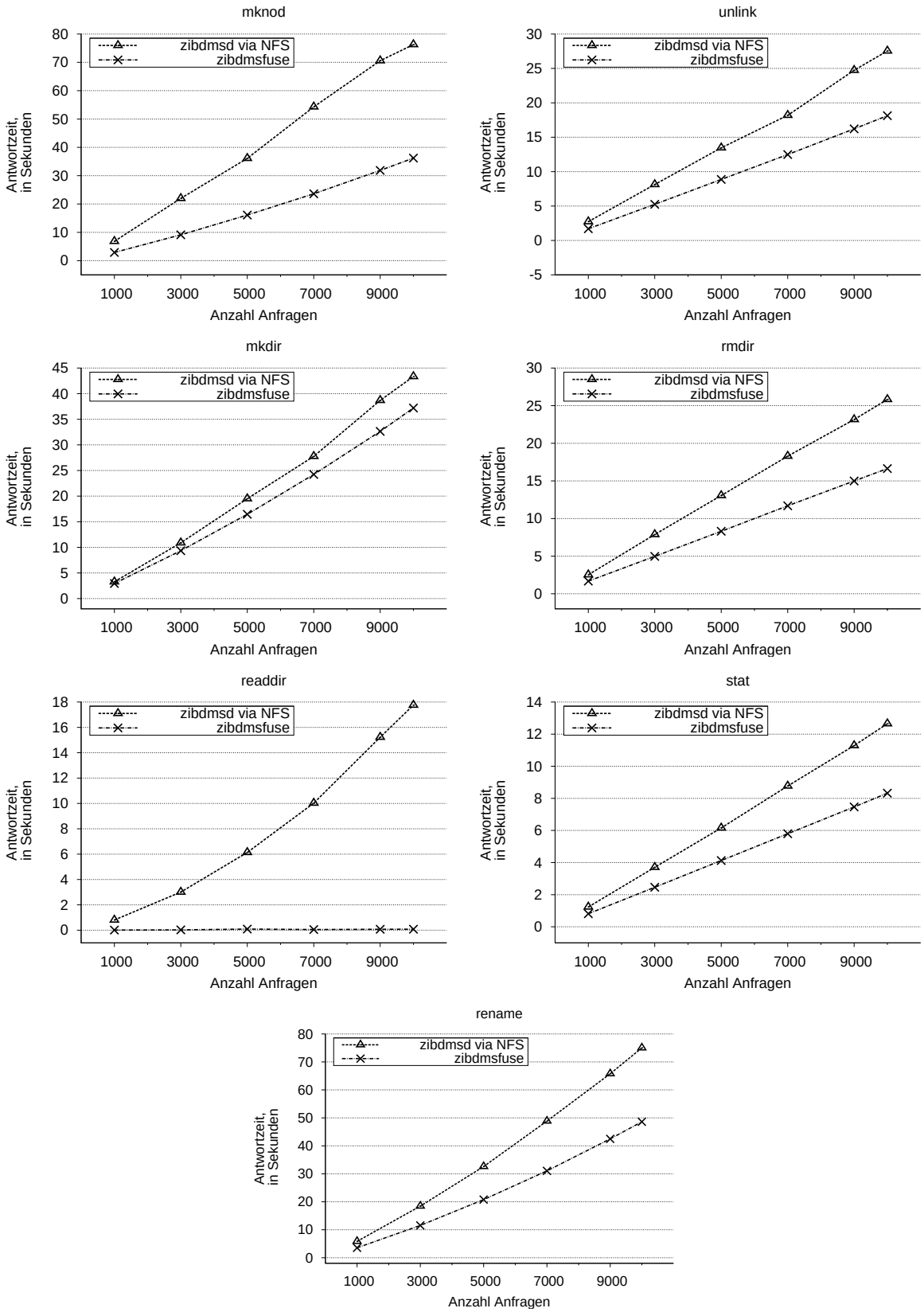

Abbildung 7.14: Gegenüberstellung der Client-Messungen

Mit dem FUSE-Client liegt somit eine performante Möglichkeit für den entfernten Zugriff auf das ZIB-DMS vor. Dies wird vor allem im Vergleich zu den Messungen lokal auf dem ZIB-DMS-Server (siehe Abbildung 7.9 in Abschnitt 7.2.3) deutlich. Die bei dem Zugriff über den FUSE-Client und die CORBA-ZIB-DMS-API gemessenen Zeiten liegen nur leicht über den Werten, die beim lokalen Zugriff über die FUSE-Proxy-Komponente und die native ZIB-DMS-API erzielt wurden. Dies zeigt, dass die Kommunikation über CORBA nur einen sehr geringen Overhead erzeugt, während der Re-Export über eine Dateiserver-Software eine zusätzliche Software-Ebene verbunden mit einem zusätzlichen Overhead erzeugt.

8 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, ein Userspace-Dateisystem zu schaffen, das die Einbindung des Grid-Datenmanagement-Systems *ZIB-DMS* in den Verzeichnisbaum eines Linux-Systems erlaubt. Damit soll eine intuitive, einfache und leicht integrierbare Möglichkeit geschaffen werden, transparent auf den Datenbestand des Systems zuzugreifen und diesen über die Funktionalität der Dateisystem-Schnittstelle zu verwalten. Die vorliegende schriftliche Arbeit dokumentiert und diskutiert die Ergebnisse dieser Bemühungen.

Dabei wurden zunächst die zentralen Konzepte zur Verwaltung von Daten in Dateisystemen, verschiedene Dateisystem-Typen sowie die Vor- und Nachteile dieser Form der Datenverwaltung identifiziert. Weiterführend wurden die speziellen Anforderungen, welche bei der Datenverwaltung in global verteilten Grid-Systemen auftreten, erörtert und einige populäre Grid-Datenmanagement-Lösungen vorgestellt.

Auf Basis dieser grundlegenden Betrachtungen wurde anschließend das *ZIB-DMS* vorgestellt. Es wurden die Gestaltungsprinzipien der auf P2P-Komponenten basierenden Software-Architektur sowie die einzelnen Komponenten des Systems eingehend beschrieben. Besondere Aufmerksamkeit galt den verschiedenen verfügbaren Mechanismen zur Organisation der Daten. Das System bietet neben dem direkten Zugriff auf diese Verwaltungsmechanismen mit dem *Directory-View* eine Darstellung als hierarchische Verzeichnis-Struktur, welche die Basis für die Dateisystem-Darstellung des ZIB-DMS bildet. Desweiteren wurde eine Abgrenzung zu den anderen zuvor vorgestellten Grid-Datenmanagement-Systemen vorgenommen. Im Anschluss wurden grundsätzliche Aspekte zur Implementierung von Dateisystemen erörtert. Dabei wurde der Begriff *Userspace-Dateisystem* und die Motivation zur Schaffung solcher Dateisysteme erklärt. Mit FUSE steht ein durchdachtes, komfortables und einfach zu benutzendes Werkzeug für die Realisierung von Userspace-Dateisystemen zur Verfügung. Dessen Aufbau wurde anschließend vorgestellt, wobei zur Schaffung eines besseren Verständnisses auch die interne Funktionsweise studiert und beschrieben wurde. Die Verwendung des Rahmenwerks und die notwendigen Schritte bei der Implementierung eines FUSE-Dateisystems wurden im weiteren an einfachen Beispielen dargestellt.

Nachdem die für diese Arbeit grundlegende Software vorgestellt wurde, wurde im Hauptteil der schriftlichen Arbeit der Verlauf des praktischen Teils dokumentiert. Es wurden zunächst der Stand des ZIB-DMS zu Beginn der Arbeit und die in dieser Version auftretenden Probleme identifiziert, welche die Motivation zur Schaffung einer Userspace-Dateisystem-Schnittstelle untermauerten. Nach einigen vorbereitenden Arbeiten, wie die

Optimierung des Übersetzungsprozesses und Änderungen am Aufbau des ZIB-DMS, wurde die defekte NFS-Komponente entfernt und mit einer FUSE-Dateisystem-Komponente ersetzt. Die Trennung zwischen Dateisystem-Schnittstelle und Darstellungslogik galt dabei als zentrales Entwurfsziel, da auch andere Zugriffs-Komponenten des Systems die hierarchische Darstellung verwenden und somit Dopplungen im Quelltext sowie eventuelle Inkonsistenzen zwischen verschiedenen Zugriffs-Komponenten vermieden werden können. Da diese Trennung zu Beginn der Arbeit nicht vollständig realisiert war, mussten zunächst weitreichende Änderungen an der *Directory-View*-Komponente vorgenommen werden, die im einzelnen anhand von Anwendungsbeispielen beschrieben wurden. Anschließend wurde die Implementation der FUSE-Komponente dargestellt, die es erlaubt das ZIB-DMS in den Verzeichnisbaum des Servers einzubinden und mittels Standard-Dateiserver-Software für den entfernten Zugriff freizugeben. Um einen direkten entfernten Zugriff zu ermöglichen, wurde desweiteren ein eigenständiges FUSE-Dateisystem implementiert, welches die Funktionalität der FUSE-Komponente auslagert und die Verbindung zu einem ZIB-DMS-Server über die CORBA-Schnittstelle herstellt.

Abschließend wurden verschiedene Leistungsmessungen vorgenommen, deren Ergebnisse grafisch dargestellt und diskutiert wurden. Dabei war hauptsächlich der Einfluß verschiedener Software-Ebenen auf die Gesamtleistung von Interesse. Es zeigte sich, dass der durch das FUSE-Rahmenwerk erzeugte Overhead hier zu vernachlässigen ist, während die Leistungsfähigkeit des Metadaten-Katalogs ausschlaggebend für die Gesamtleistung ist. Für den entfernten Zugriff erwies sich der FUSE-Client als die günstigere Alternative, im Vergleich zu einem lokal eingebundenen und über NFS freigegebenen ZIB-DMS. Diese Variante ist nur dann sinnvoll, wenn der FUSE-Client nicht eingesetzt werden kann oder soll. Die Messungen zeigten, dass einiger Optimierungsbedarf an den verschiedenen Kompontenen des ZIB-DMS, insbesondere aber an der Katalog-Komponente, besteht. Insgesamt wurde im Laufe der Arbeit aber deutlich, dass mit der Userspace-Dateisystem-Schnittstelle des ZIB-DMS eine günstige und einfach zu verwendende Zugriffsmöglichkeit geschaffen wurde. Aufgrund der teilweise komplexen Abbildung der erweiterterten Verwaltungsmechanismen und den fehlenden Steuerungsmechanismen der Service-Komponenten kann sie zwar die übersichtlichere grafische Benutzeroberfläche nicht vollständig ersetzen, ermöglicht dafür aber den automatisierten Zugriff und die Integration mit Standard-Mechanismen und -Software.

In der weiteren Entwicklungsarbeit am ZIB-DMS könnte neben den notwendigen Optimierungen am Metadaten-Katalog auch der Einsatz eines elaborierten Caching-Verfahrens auf Seite der FUSE-Komponenten zur Verbesserung der Leistung beitragen. Zudem könnte, wie in Kapitel 6 beschrieben, der Funktionsumfang des Directory-View um Abbildungs- und Steuerungsmechanismen der Service-Komponenten erweitert und die Einschränkungen der hierarchischen Abbildung der Graph-Komponente durch Reimplementierung der Kanten-Operationen aufgehoben werden.

Abbildungsverzeichnis

Listingverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu und I. Foster: *The Globus Striped GridFTP Framework Aand Server*. In: *Proceedings of Super Computing 2005 (SC05)*, November 2005.

[2] W.E. Allcock, I. Foster und R. Madduri: *Reliable Data Transport: A Critical Service for the Grid*. In: *Building Service Based Grids Workshop, Global Grid Forum 11*, Juni 2004.

[3] C. Baru, R. Moore, A. Rajasekar und M. Wan: *The SDSC Storage Resource Broker*, 1998. San Diego Supercomputer Center.

[4] T. Berners-Lee, R. Fielding und L. Masinter: *Uniform Resource Identifier (URI): Generic Syntax*. RFC 3986 (Standard), Januar 2005. http://www.ietf.org/rfc/rfc3986.txt.

[5] T. Berners-Lee, L. Masinter und M. McCahill: *Uniform Resource Locators (URL)*. RFC 1738 (Proposed Standard), Dezember 1994. http://www.ietf.org/rfc/rfc1738.txt, Obsoleted by RFCs 4248, 4266, updated by RFCs 1808, 2368, 2396, 3986.

[6] D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell und J. von Reich: *The Open Grid Services Architecture, Version 1.5*. Global Grid Forum, Juli 2006.

[7] D. P. Bovet und M. Cesati: *Understanding The Linux Kernel*. O'Reilly, 2. Auflage, 2002.

[8] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte und D. Winer (Herausgeber): *Simple Object Access Protocol (SOAP) 1.1*. World Wide Web Consortium (W3C), 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508, besucht: 8. April 2008.

[9] P. J. Braam: *Linux Virtual File System*. Präsentation, April 1998.

[10] P. J. Braam: *The Coda Distributed File System*. Linux Journal, 50, Juni 1998.

[11] D. R. Butenhof: *Programming with POSIX Threads*. Addison-Wesley, Mai 1997.

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

[13] R. Card, T. Ts'o und S. Tweedie: *Design and Implementation of the Second Extended Filesystem.* In: *Proceedings Of The First Dutch International Symposium On Linux*, 1994.

[14] ECMA: *ECMA-107: Volume and File Structure of Disk Cartridges for Information Interchange.* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, 2. Auflage, Juni 1995. http://www.ecma.ch/ecma1/STAND/ECMA-107.HTM, besucht: 8. April 2008.

[15] M. Eisler, A. Chiu und L. Ling: *RPCSEC_GSS Protocol Specification.* RFC 2203 (Proposed Standard), September 1997. http://www.ietf.org/rfc/rfc2203.txt.

[16] I. Foster: *The Grid: A New Infrastructure for 21st Century Science.* Physics Today, 55:42 – 48, Februar 2002.

[17] I. Foster: *What is the Grid? A Three Point Checklist*, Juli 2002. Argonne National Laboratory and University of Chicago.

[18] I. Foster: *Globus Toolkit Version 4: Software For Service-Oriented Systems.* In: *IFIP International Conference On Network And Parallel Computing*, Band 3779, Seiten 2–13. Springer, 2005.

[19] I. Foster und C. Kesselman (Herausgeber): *The Grid 2: Blueprint For A New Computing Infrastructure.* Morgan Kaufmann, 2. Auflage, November 2003.

[20] R. Freund: *File Systems and Usability – the Missing Link.* Bachelor's Thesis, University of Osnabrück, Cognitive Science, Juli 2007.

[21] J. E. F. Friedl: *Reguläre Ausdrücke.* O'Reilly, 1998.

[22] J. Galbraith: *SSH File Transfer Protocol.* Internet-Draft, Oktober 2004. http://www.snailbook.com/docs/sftp.txt, besucht: 8. April 2008.

[23] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[24] J. Gosling und H. McGilton: *The Java Language Environment.* SUN Microsystems, Mai 1996.

[25] M. Henning und S. Vinoski: *Advanced CORBA Programming with C++.* Addison-Wesley, 1999.

[26] J. H. Howard: *An Overview Of The Andrew File System.* In: *Proceedings Of The USENIX Winter Technical Conference*, Februar 1988.

[27] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti und E. Cesario: *The XtreemFS Architecture.* LinuxTag 2007 Programm, 2007.

[28] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti und E. Cesario: *XtreemFS - A Case For Object-Based Storage In Grid Data Management*, 2007. Zuse Institut Berlin, Barcelona Supercomputing Center, Universitat de Catalunya, NEC HPC Europe GmbH, ICAR-CNR.

[29] Open Systems Interconnection: *ISO/IEC 11578:1996 Information technology – Open Systems Interconnection – Remote Procedure Call*. Technical Standard, 1996.

[30] International Organization For Standardization: *Information technology – Portable Operating System Interface (POSIX)*. IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1990. International standard ISO/IEC 9945. IEEE Std 1003.1-1990 (revision of IEEE Std 1003.1-1988). Part 1. System application program interface (API) [C language].

[31] P. Leach, M. Mealling und R. Salz: *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122 (Proposed Standard), Juli 2005. http://www.ietf.org/rfc/rfc4122.txt.

[32] M. Lutz: *Programming Python*. O'Reilly, 3. Auflage, August 2006.

[33] P. Lyman und H. R. Varian: *How Much Information 2003?*, Oktober 2003. http://www2.sims.berkeley.edu/research/projects, besucht: 8. April 2008.

[34] Manpage: *Linux XFS Compatibility API - ATTR(1)*, November 2000. http://oss.sgi.com/projects/xfs, besucht: 8. April 2008.

[35] M. K. McKusick, W. N. Joy, S. J. Leffler und R. S. Fabry: *A Fast File System For Unix*. ACM Transactions On Computer Systems, 2(3):181–197, August 1984.

[36] P. Mochel: *The sysfs Filesystem*.

[37] R. Moore, A. Rajasekar und M. Wan: *Data Grids, Digital Libraries And Persistent Archives: An Integrated Approach To Publishing, Sharing And Archiving Data*, März 2004.

[38] R. W. Moore, C. Baru, R. Marciano, A. Rajasekar und M. Wan: *Data-Intensive Computing*. In: *The Grid: Blueprint for a New Computing Infrastructure*, Band 1, Kapitel 5, Seiten 105 – 128. Morgan Kaufmann, 2003.

[39] M. Moser: *Platzierung von Replikaten in Verteilten Systemen*. Diplomarbeit, Humboldt Universität zu Berlin, 2006.

[40] OMG: *CORBA Trading Object Service Specification*. The Object Management Group (OMG), Mai 2000.

[41] OMG: *Common Object Request Broker Architecture (CORBA/IIOP)*. The Object Management Group (OMG), März 2004. Kern-Spezifikation.

[42] OMG: *CORBA Naming Service Specification.* The Object Management Group (OMG), Oktober 2004.

[43] Open Group: *DCE 1.1: Remote Procedure Call.* Technical Standard, August 1997.

[44] J. O'Toole und D. Gifford: *Names should mean what, not where.* In: *Proceedings of the 5th workshop on ACM SIGOPS European workshop: Models and paradigms for distributed systemsstructuring.* ACM Press, September 1992.

[45] K. Peter, S. Plantikow, M. Högqvist, C. Grimme und A. Papaspyrou: *Generalizing The Data Management Of Three Community Grids.* Preprint, Dezember 2007. Zuse Institut Berlin (ZIB).

[46] T. A. Pham: *Einbettung neuer Verwaltungsmethoden in die hierarchische Dateisystemsicht.* Diplomarbeit, Technische Universität Berlin, August 2005.

[47] M. Pool: *distcc – A Fast Free Distributed Compiler.* Whitepaper, Dezember 2003. http://distcc.samba.org, besucht: 8. April 2008.

[48] J. Postel und J. Reynolds: *File Transfer Protocol.* RFC 959 (Standard), Oktober 1985. http://www.ietf.org/rfc/rfc959.txt, Updated by RFCs 2228, 2640, 2773.

[49] C. von Prollius: *Chord# in Planet-Lab.* Diplomarbeit, Freie Universität Berlin, Februar 2007.

[50] A. Rajasekar, M. Wan, R. Moore und W. Schroeder: *A Prototype Rule-based Distributed Data Management System*, April 2006. San Diege Supercomputer Center, University of California.

[51] A. Reinefeld, F. Schintke und T. Schütt: *Scalable and Self-Optimizing Data Grids*, Band 6 der Reihe *Annual Review of Scalable Computing*, Kapitel 2, Seiten 30–60. World Scientific, 2004.

[52] A. Reinefeld, F. Schintke und T. Schütt: *P2P Routing of Range Queries in Skewed Multidimensional Data Sets.* ZIB-Report 07-23, Zuse Institut Berlin (ZIB), August 2007.

[53] M. Ripeanu und I. Foster: *A Decentralized, Adaptive, Replica Location Service.* In: *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, Juli 2002.

[54] P. Saint-Andre: *Extensible Messaging and Presence Protocol (XMPP): Core.* RFC 3920 (Proposed Standard), Oktober 2004. http://www.ietf.org/rfc/rfc3920.txt.

[55] F. Schintke: *The ZIB Distributed Data Management System - ZIBDMS.* Präsentation, November 2004.

[56] F. Schintke: *ZIB-DMS - Managing Data in a Distributed Environment.* Präsentation, August 2006.

[57] F. Schintke: *iRODS - iRule Oriented Data System.* Vortrag im Rahmen des DGI, TextGrid Metadata Workshop, Göttingen, März 2007. Zuse Institut Berlin (ZIB).

[58] Florian Schintke, Thorsten Schütt und Alexander Reinefeld: *A Framework for Self-Optimizing Grids Using P2P Components.* In: *14th Intl. Workshop on Database and Expert Systems Applications (DEXA'03)*, Seiten 689–693. IEEE Computer Society, September 2003.

[59] U. Schöning: *Theoretische Informatik - kurzgeasst.* Spektrum Akademischer Verlag, 4. Auflage, 2001.

[60] T. Schütt, A. Merzky, A. Hutanu und F. Schintke: *Remote Partial File Access Using Compact Pattern Descriptions.* Technischer Bericht, Zuse Institut Berlin (ZIB), 2004.

[61] T. Schütt, F. Schintke und A. Reinefeld: *Efficient Synchronization of Replicated Data in Distributed Systems.* In: P. M. A. Sloot, D. Abramson, A. V. Bogdanov, J. J. Dongarra, A. Y. Zomaya und Y. E. Gorbachev (Herausgeber): *Computational Science - ICCS 2003, International Conference Melbourne, Australia and St. Petersburg, Russia, Proceedings, Part I*, Band 2657 der Reihe *Lecture Notes in Computer Science*, Seiten 274–283. Springer, Juni 2003.

[62] T. Schütt, F. Schintke und A. Reinefeld: *A Structured Overlay for Multi-dimensional Range Queries.* In: A. M. Kermarrec, L. Bougé und T. Priol (Herausgeber): *Euro-Par*, Band 4641 der Reihe *Lecture Notes in Computer Science*, Seiten 503–513. Springer, 2007.

[63] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler und D. Noveck: *Network File System (NFS) version 4 Protocol.* RFC 3530 (Proposed Standard), April 2003. http://www.ietf.org/rfc/rfc3530.txt.

[64] A. Silberschatz: *Operating System Concepts.* John Wiley & Sons, 6. Auflage, 2002.

[65] A. Silberschatz, H.F. Korth und S. Sudarshan: *Database Systems Concepts.* McGraw-Hill, 5. Auflage, 2005.

[66] R. Srinivasan: *RPC: Remote Procedure Call Protocol Specification Version 2.* RFC 1831 (Proposed Standard), August 1995. http://www.ietf.org/rfc/rfc1831.txt.

[67] R. Srinivasan: *XDR: External Data Representation Standard.* RFC 1832 (Draft Standard), August 1995. http://www.ietf.org/rfc/rfc1832.txt, Obsoleted by RFC 4506.

[68] W. Stallings: *Operating Systems.* Prentice Hall PTR, 4. Auflage, 2001.

[69] I. Stoica, R. Morris, D. Karger, F. Kaashoek und H. Balakrishnan: *Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications.* In: *Proceedings Of The 2001 ACM SIGCOMM Conference*, Seiten 149–160, 2001.

[70] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto und G. Peck: *Scalability in the XFS File System.* In: *Proceedings Of The 1996 USENIX Conference.* Silicon Graphics, Inc. (SGI), Januar 1996.

[71] M. Szeredi: *AVFS - A Virtual File System.* http://avf.sourceforge.net, besucht: 8. April 2008.

[72] M. Szeredi: *Filesystem in Userspace.* http://www.fuse.org, besucht: 8. April 2008.

[73] M. Szeredi: *SSH Filesystem.* http://fuse.sourceforge.net/sshfs.html, besucht: 8. April 2008.

[74] A. S. Tanenbaum: *Modern Operating Systems.* Prentice Hall PTR, 2. Auflage, 2001.

[75] O. Tatebe, Y. Morita, S. Matsuoka, N. Soda und S. Sekiguchi: *Grid Datafarm Architecture For Petascale Data Intensive Computing.* In: *Proceedings Of The 2nd IEEE/ACM International Symposium On Cluster Computing And The Grid (CCGrid 2002)*, Seiten 102–110, 2002. http://datafarm.apgrid.org, besucht: 8. April 2008.

[76] O. Tatebe, Y. Morita, N. Soda und S. Matsuoka: *Gfarm v2: A Grid File System That Supports High-Performance Distributed And Parallel Data Computing.* In: *Proceedings Of The 2004 Computing In High Energy And Nuclear Physics (CHEP04), Interlaken, Switzerland*, September 2004. http://datafarm.apgrid.org, besucht: 8. April 2008.

[77] T. Ts'o und S. Tweedie: *Planned Extensions to the Linux Ext2/Ext3 Filesystem.* In: *Proceedings of the 2002 USENIX Technical Conference*, Seiten 235–244, 2002.

[78] G. V. Vaughan, B. Elliston und T. Tromey: *GNU Autoconf, Automake and Libtool.* SAMS, 2001.

[79] Website: *AstroGrid-D – German Astronomy Community Grid.* http://www.gac-grid.de, besucht: 8. April 2008.

[80] Website: *Bonnie++ Filesystem Benchmarks.* http://sourceforge.net/projects/bonnie, besucht: 8. April 2008.

[81] Website: *boost C++ Libraries.* http://boost.org, besucht: 8. April 2008.

[82] Website: *C3-Grid – Collaborative Climate Community Data And Processing Grid.* http://www.c3grid.de, besucht: 8. April 2008.

[83] Website: *CLucene Search Engine*. http://clucene.wiki.sourceforge.net, besucht: 8. April 2008.

[84] Website: *cppunit – A C++ Unit Testing Framework*. http://cppunit.sourceforge.net, besucht: 8. April 2008.

[85] Website: *cURL and libcurl*. http://curl.haxx.se, besucht: 8. April 2008.

[86] Website: *CurlFtpFS – A FTP Filesystem Based On cURL And FUSE*. http://curlftpfs.sourceforge.net, besucht: 8. April 2008.

[87] Website: *E2fsprogs: Ext2/3/4 Filesystem Utilities*. http://e2fsprogs.sourceforge.net, besucht: 8. April 2008.

[88] Website: *GfarmFS-FUSE*. http://datafarm.apgrid.org/software/gfarmfs-fuse.en.html, besucht: 8. April 2008.

[89] Website: *IOzone Filesystem Benchmark*. http://www.iozone.org, besucht: 8. April 2008.

[90] Website: *MediGRID – GRID-Computing für die Medizin und Lebenswissenschaften*. http://www.medigrid.de, besucht: 8. April 2008.

[91] Website: *MySQL – Die populärste Open-Source-Datenbank der Welt*. http://www.mysql.de, besucht: 8. April 2008.

[92] Website: *MySQLfs – FUSE Filesystem Using MySQL As A Storage*. https://sourceforge.net/projects/mysqlfs, besucht: 8. April 2008.

[93] Website: *Netatalk - Networking Apple Macintosh through Open Source*. http://netatalk.sourceforge.net, besucht: 8. April 2008.

[94] Website: *omniORB – Free High Performance ORB*. http://omniorb.sourceforge.net, besucht: 8. April 2008.

[95] Website: *Open Grid Forum*. http://www.ggf.org, besucht: 8. April 2008.

[96] Website: *samba – Opening Windows To A Wider World*. http://www.samba.org, besucht: 8. April 2008.

[97] Website: *SQLite Database Engine*. http://www.sqlite.org, besucht: 8. April 2008.

[98] Website: *The Object Management Group (OMG)*. http://www.omg.org, besucht: 8. April 2008.

[99] Website: *World Wide Web Consortium (W3C)*. http://www.w3.org, besucht: 8. April 2008.

[100] Website: *XtreemOS : A Linux-based Operating System To Support Virtual Organizations For Next Generation Grids*. http://www.xtreemos.org, besucht: 8. April 2008.

[101] J. Witte: *Implementierung einer NFS Schnittstelle im Kontext eines Grid-Datenmanagement-Systems*. Diplomarbeit, Humboldt Universität zu Berlin, Januar 2005.

[102] J. Wray: *Generic Security Service API : C-bindings*. RFC 1509 (Proposed Standard), September 1993. http://www.ietf.org/rfc/rfc1509.txt, Obsoleted by RFC 2744.

[103] H. Zimmermann: *OSI Reference Model — The ISO Model Of Architecture For Open Systems Interconnection*. In: *IEEE Transactions On Communications*, Band 28, Seiten 425–432, April 1980.

A Rohdaten der Leistungsmessungen

A.1 Messungen mit steigender Objektzahl

- **ext3 Dateisystem**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	55	1	55	55	55	55	56
3000	194	7	191	218	179	194	189
5000	326	3	312	329	323	341	323
7000	468	3	465	477	450	492	454
9000	1862	2	1814	1851	1839	1912	1896
10000	1748	4	1796	1708	1812	1806	1620

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	33	1	33	34	33	33	33
3000	97	0	97	98	97	97	97
5000	167	0	167	168	167	167	167
7000	239	0	239	239	240	238	239
9000	312	0	311	313	313	312	310
10000	349	0	349	350	349	349	350

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1	0	1	1	1	1	1
3000	4	0	4	4	4	4	4
5000	6	7	7	6	6	6	6
7000	9	11	9	9	8	9	11
9000	12	20	17	11	11	11	11
10000	12	0	12	12	12	12	12

[1] Mittelwert über allen Messdurchgängen, in Millisekunden
[2] Mittlere Abweichung vom Mittelwert, in %

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	39	1	39	39	38	39	39
3000	116	0	116	116	116	115	116
5000	202	0	203	203	201	202	203
7000	290	0	290	290	290	290	291
9000	386	2	379	393	376	390	394
10000	419	1	424	418	418	415	421

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	34	2	35	35	34	34	34
3000	376	59	105	580	104	546	547
5000	177	1	176	181	176	178	176
7000	712	79	1416	256	1384	253	249
9000	324	1	322	330	320	326	324
10000	360	1	359	365	357	362	355

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	5	0	5	5	5	5	5
3000	15	0	15	15	15	15	15
5000	25	0	25	25	25	25	25
7000	35	0	35	35	35	35	35
9000	45	0	45	45	45	45	45
10000	50	1	51	51	50	50	50

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	24	3	25	25	24	24	24
3000	75	0	75	75	75	75	75
5000	144	12	126	164	127	141	164
7000	179	0	179	179	179	179	178
9000	241	5	231	231	231	256	256
10000	266	7	301	256	257	257	257

- **FUSE-Loopback Dateisystem**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	171	7	158	158	181	167	189
3000	517	3	535	521	520	493	514
5000	831	1	831	828	834	839	823
7000	1160	2	1179	1125	1179	1165	1154
9000	2441	2	2408	2406	2429	2440	2523
10000	2394	3	2517	2408	2374	2357	2312

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	137	3	133	134	143	134	142
3000	418	3	403	429	426	403	429
5000	699	3	712	717	672	678	717
7000	1014	0	1013	1018	1011	1021	1009
9000	1308	0	1309	1311	1305	1314	1299
10000	1452	0	1455	1453	1445	1455	1450

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3	0	3	3	3	3	3
3000	8	0	8	8	8	8	8
5000	12	4	13	12	12	12	12
7000	17	3	17	17	17	17	18
9000	22	2	22	22	22	23	22
10000	25	3	25	25	25	24	24

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	174	3	170	170	181	170	180
3000	532	3	514	542	543	512	547
5000	895	3	921	923	861	861	907
7000	1295	1	1292	1301	1280	1307	1294
9000	1685	1	1661	1693	1686	1720	1666
10000	1874	2	1916	1859	1838	1915	1842

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	128	6	119	131	136	119	136
3000	382	4	384	401	365	363	397
5000	668	0	668	670	669	668	663
7000	937	1	936	915	939	949	947
9000	1217	1	1218	1212	1233	1203	1220
10000	1345	1	1326	1378	1358	1328	1337

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	49	1	49	49	50	49	50
3000	151	2	149	152	153	148	154
5000	250	1	252	253	245	247	254
7000	350	4	357	355	356	358	322
9000	447	3	451	460	456	448	418
10000	500	1	497	510	494	502	499

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	118	4	114	114	123	114	123
3000	368	3	372	374	375	344	376
5000	629	1	628	636	620	621	641
7000	887	1	894	881	883	894	883
9000	1138	0	1143	1136	1135	1133	1141
10000	1267	1	1254	1278	1276	1259	1266

- **ext3 Dateisystem via NFS**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3850	7	3847	4065	4029	3329	3981
3000	10596	15	7658	10626	12122	10494	12078
5000	19656	19	15330	22290	15053	22817	22790
7000	27758	13	28971	33091	21821	28086	26822
9000	37284	9	33792	40880	33050	37916	40784
10000	41536	12	33811	43962	37594	46910	45402

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	4293	0	4296	4324	4264	4300	4281
3000	11539	8	12013	12001	12053	12033	9593
5000	18135	22	10326	20077	20053	20093	20125
7000	24775	21	28042	25110	14527	28045	28149
9000	34941	7	36258	36214	30025	36077	36129
10000	31029	27	20976	40150	21385	39873	32763

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	38	0	38	38	38	38	38
3000	117	1	116	117	117	116	117
5000	197	0	198	197	197	197	197
7000	270	0	269	269	272	270	269
9000	425	23	467	600	353	352	353
10000	393	0	394	393	392	395	391

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3375	20	2303	3981	3340	3102	4150
3000	9907	29	4693	11954	8886	11997	12003
5000	13211	42	7988	19994	8019	19998	10055
7000	22574	31	17698	28001	11163	28008	27999
9000	27505	38	14691	36013	14751	36039	36030
10000	31333	28	32238	27751	16611	40022	40044

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2603	13	2650	2002	2671	2668	3024
3000	6487	21	4402	8017	6005	8005	6007
5000	11871	22	10096	14881	7660	13405	13313
7000	16468	21	18691	14158	10696	20107	18688
9000	19033	25	13945	18044	14018	25063	24097
10000	22723	14	20049	20080	26716	26721	20047

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	260	0	260	259	261	260	260
3000	780	0	780	774	783	779	782
5000	1305	1	1324	1298	1302	1298	1304
7000	1825	1	1818	1815	1852	1819	1821
9000	2339	0	2338	2333	2341	2342	2342
10000	2607	1	2599	2589	2641	2600	2604

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2688	10	2823	2803	2833	2808	2175
3000	6642	26	4093	8031	6006	9053	6025
5000	9403	26	6770	10007	6876	10001	13361
7000	15377	25	19270	14003	9527	14050	20033
9000	15854	17	12518	18287	12416	18000	18051
10000	20223	19	21140	25934	13943	20004	20092

- **ZIB-DMS mit DummyDV via NFS**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3282	2	3238	3287	3418	3242	3224
3000	10894	2	10700	10766	11293	10808	10902
5000	19498	2	19218	19823	20197	19114	19137
7000	27765	4	27157	26599	29721	26906	28442
9000	38692	1	38626	38293	39711	38260	38571
10000	43337	2	42350	42252	45202	43338	43543

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	6823	1	6806	6898	6945	6761	6707
3000	21990	3	21644	21886	23079	21903	21438
5000	36109	5	34813	34828	38997	34517	37390
7000	54293	1	54197	54693	55415	53760	53399
9000	70596	1	70740	69577	72536	70381	69745
10000	76284	1	75839	76375	77040	75278	76890

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	801	1	810	812	803	793	786
3000	3006	2	2972	2995	3106	2990	2969
5000	6132	2	6108	6152	6294	6089	6016
7000	10024	2	10111	9737	10311	9994	9967
9000	15225	1	15287	15168	15533	15095	15041
10000	17755	1	17739	17719	18180	17592	17545

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	5854	2	5963	5987	5641	5851	5828
3000	18457	2	18135	18302	19184	18385	18279
5000	32593	4	33132	30330	34167	32844	32494
7000	48850	2	49092	47549	50699	48706	48204
9000	65774	3	66627	62037	68242	66202	65763
10000	75015	2	74892	76117	76364	74405	73295

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2529	2	2508	2540	2627	2502	2469
3000	7881	2	7683	7786	8240	7882	7812
5000	13055	2	12928	13076	13610	12828	12831
7000	18294	2	17989	18664	18997	17976	17843
9000	23158	6	23066	24736	24281	20794	22913
10000	25830	2	25634	25709	26928	25560	25321

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1234	1	1238	1249	1251	1222	1211
3000	3706	3	3572	3624	3898	3748	3686
5000	6156	5	6266	5593	6527	6260	6135
7000	8771	2	8866	8521	9124	8731	8611
9000	11287	2	11354	11063	11731	11201	11084
10000	12652	2	12602	12770	13075	12503	12312

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2729	1	2764	2724	2668	2762	2727
3000	8131	1	8093	8168	8273	8087	8032
5000	13459	2	13505	13005	14011	13470	13304
7000	18188	5	17448	17689	19736	17178	18888
9000	24740	2	24594	24898	25397	24519	24294
10000	27543	2	27475	28032	28065	27201	26941

- **XtreemFS**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3874	3	3963	3976	3686	3819	3924
3000	11443	2	11228	11878	11236	11640	11235
5000	19672	1	19915	19619	19166	19650	20011
7000	27873	1	28497	28033	27291	27658	27887
9000	37383	4	34893	37676	38566	38327	37451
10000	43374	2	42983	45272	43344	43143	42127

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	11548	13	11348	14270	11595	9749	10778
3000	30333	11	29621	29887	36948	27219	27989
5000	49471	8	50875	42730	55656	49318	48776
7000	81938	9	90303	76523	90610	73761	78493
9000	99861	8	110661	93324	109569	94759	90992
10000	116912	11	122154	105287	137797	115715	103609

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	16	3	16	16	16	15	16
3000	34	2	34	35	34	33	34
5000	54	3	54	52	53	55	56
7000	95	44	72	179	74	75	77
9000	187	25	95	223	205	204	210
10000	162	37	239	233	109	111	119

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3954	4	4097	4165	3949	3792	3767
3000	11588	3	11472	12250	11516	11286	11414
5000	19719	3	19986	19782	19171	19154	20502
7000	28640	2	27373	29165	29075	28503	29085
9000	37621	3	35677	37112	38888	39202	37225
10000	43979	2	42392	44419	44121	45487	43477

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2620	3	2641	2751	2554	2496	2660
3000	7909	2	8029	8053	8034	7596	7834
5000	12973	2	12944	13140	13186	12595	12999
7000	18930	3	18441	18793	19809	18554	19051
9000	25393	3	24048	25981	25540	25473	25923
10000	28971	1	28403	29224	29224	29114	28888

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1385	4	1441	1461	1353	1345	1325
3000	4172	3	4034	4340	4134	4036	4316
5000	7009	2	7335	6977	6981	6884	6866
7000	9952	3	9714	9869	10464	9976	9739
9000	13037	2	12865	12872	13637	12880	12933
10000	15476	2	15192	15115	15548	15829	15694

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	14617	17	15523	15276	17021	15545	9721
3000	41342	13	38022	36245	50042	44687	37712
5000	69693	24	58173	75267	91280	79120	44623
7000	96207	21	115718	84205	123909	83812	73392
9000	114215	14	126171	102380	140390	102327	99805
10000	123339	17	130047	111824	161268	112255	101302

- **ZIB-DMS mit DummyDV**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1897	1	1897	1884	1885	1909	1912
3000	6227	1	6156	6209	6259	6289	6221
5000	10606	1	10588	10558	10647	10694	10542
7000	14695	1	14789	14677	14753	14834	14420
9000	19215	1	19027	19016	19194	19450	19388
10000	21445	1	21587	21267	21215	21646	21509

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1971	2	1996	1946	1935	2019	1960
3000	6030	1	5977	6019	6046	6038	6070
5000	10147	0	10143	10147	10216	10153	10077
7000	14290	0	14234	14272	14350	14280	14316
9000	18415	0	18293	18403	18502	18534	18342
10000	20556	0	20464	20492	20586	20506	20732

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	5	0	5	5	5	5	5
3000	13	0	13	13	13	13	13
5000	22	2	22	22	22	21	22
7000	28	2	28	29	29	28	28
9000	39	1	39	38	39	39	39
10000	42	1	41	42	42	42	42

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2662	2	2688	2546	2661	2705	2709
3000	8085	1	7972	8172	8195	8041	8047
5000	13571	1	13364	13686	13734	13581	13490
7000	19072	1	18919	19209	19239	18965	19026
9000	24694	0	24579	24856	24787	24658	24592
10000	27323	0	27271	27343	27334	27302	27365

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1230	0	1231	1227	1227	1230	1233
3000	3846	1	3812	3842	3836	3947	3794
5000	6423	1	6405	6354	6348	6461	6549
7000	8903	2	8830	8828	8746	8797	9313
9000	11368	1	11363	11354	11235	11530	11356
10000	12631	0	12666	12600	12544	12730	12615

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	646	2	659	621	649	660	642
3000	1936	1	1909	1963	1975	1908	1925
5000	3253	2	3171	3285	3312	3320	3178
7000	4539	1	4473	4591	4626	4506	4499
9000	5989	1	5993	5911	5967	6010	6064
10000	6522	1	6515	6584	6619	6360	6533

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1304	1	1307	1277	1324	1312	1299
3000	3949	0	3946	3948	3943	3957	3950
5000	6657	1	6617	6594	6596	6851	6626
7000	9340	1	9358	9230	9206	9421	9483
9000	12178	2	12507	11901	11834	12452	12198
10000	13522	2	13993	13245	13159	13701	13513

- **ZIB-DMS mit sqlite-MDC**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	29798	2	30188	30796	29037	28946	30022
3000	127893	2	123946	133252	128557	127045	126667
5000	289627	2	285608	300338	293175	284938	284076
7000	504553	3	497833	527439	509802	498779	488914
9000	787890	3	786789	824340	790368	773892	764060
10000	963453	2	957412	991355	987252	952032	929214

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	20066	3	20727	20608	19306	19584	20104
3000	70494	1	69823	71749	70143	71432	69321
5000	137255	2	134634	137697	142068	136906	134970
7000	210143	1	206935	207244	212853	212428	211254
9000	292324	2	289938	289665	291481	301702	288834
10000	342530	2	342823	349152	349041	334196	337437

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	9089	1	9065	9230	9104	9011	9036
3000	29024	1	28789	29485	29121	28812	28911
5000	50184	1	49640	50908	50580	49918	49876
7000	71114	1	70538	71874	71359	70746	71055
9000	92718	1	92316	93676	93275	91921	92401
10000	104657	1	103903	105782	105406	104333	103860

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	30058	2	30507	30863	29258	29117	30547
3000	92526	1	92709	94140	92975	91858	90949
5000	162230	1	159013	165558	162915	160944	162722
7000	229516	2	227910	232031	233756	232311	221572
9000	299075	2	299701	297440	306469	303243	288524
10000	335685	2	333436	331402	347153	339398	327035

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	31828	1	31852	32494	31226	31473	32097
3000	173412	2	166796	179086	174408	171719	175050
5000	462395	2	455394	479109	469301	461891	446281
7000	942398	2	937293	966907	954269	933288	920233
9000	1897689	2	1858440	1955522	1923440	1893906	1857137
10000	2638933	2	2599081	2715192	2676797	2624044	2579553

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	5969	1	5973	6059	6018	5876	5917
3000	18433	1	18273	18736	18682	18196	18279
5000	31298	2	30320	32174	31741	30911	31344
7000	44451	3	43741	45844	45597	44437	42637
9000	57614	3	57380	57844	59489	58460	54899
10000	65038	2	64132	64943	67294	65597	63222

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	18510	2	18932	18599	18518	18030	18472
3000	77125	1	76745	77582	78084	76571	76643
5000	164629	1	162644	165554	165031	164340	165577
7000	279140	0	278759	279293	280837	278710	278100
9000	473293	1	470811	477112	475920	473220	469400
10000	612033	1	608460	609677	618587	614477	608965

- **ZIB-DMS mit DummyDV via FUSE-Client**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2947	1	3011	2939	2936	2929	2922
3000	9345	1	9441	9326	9380	9278	9298
5000	16450	0	16561	16414	16508	16335	16434
7000	24237	0	24355	24151	24341	24103	24235
9000	32642	0	32716	32508	32558	32604	32822
10000	37199	0	37306	36969	37253	37086	37380

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	2894	0	2918	2893	2887	2881	2892
3000	9132	1	9224	9116	9090	9091	9141
5000	16095	0	16145	16115	16142	16010	16061
7000	23602	1	23636	23536	23847	23465	23524
9000	31865	0	31808	31723	32108	31814	31871
10000	36199	0	36116	36047	36374	36246	36213

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	9	0	9	9	9	9	9
3000	22	2	23	22	22	22	22
5000	87	115	37	287	37	37	36
7000	50	2	51	50	50	49	51
9000	68	4	67	66	65	71	72
10000	75	4	73	72	79	78	72

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	3546	0	3564	3543	3549	3521	3552
3000	11554	1	11551	11547	11673	11444	11553
5000	20771	1	20774	20759	20993	20602	20725
7000	31054	1	31099	30910	31355	30792	31115
9000	42490	1	42526	42257	42918	42266	42485
10000	48601	1	48852	48341	48823	48274	48714

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1687	3	1670	1734	1643	1636	1754
3000	4976	1	5020	4971	5000	4932	4959
5000	8307	1	8370	8243	8380	8238	8303
7000	11691	0	11721	11626	11683	11653	11771
9000	14990	1	15181	14871	14955	14898	15043
10000	16642	1	16853	16512	16607	16600	16640

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	814	1	825	810	811	809	814
3000	2462	1	2497	2455	2446	2445	2467
5000	4126	0	4155	4130	4129	4098	4119
7000	5796	1	5833	5748	5864	5762	5772
9000	7468	1	7492	7388	7519	7516	7424
10000	8325	1	8372	8212	8388	8381	8274

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	1698	1	1717	1684	1698	1692	1701
3000	5251	0	5284	5227	5264	5237	5241
5000	8871	0	8902	8848	8928	8834	8843
7000	12490	1	12607	12397	12521	12444	12479
9000	16221	1	16303	16066	16296	16203	16236
10000	18132	0	18239	18123	18108	18047	18142

- **ZIB-DMS mit sqlite-MDC via FUSE-Client**

`mkdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	29917	1	29188	29843	30067	29911	30577
3000	130118	3	128006	130824	129879	125936	135945
5000	283755	2	282304	279692	284579	278707	293492
7000	489608	3	482872	484502	486934	474174	519557
9000	748762	2	754277	747060	735357	729236	777881
10000	898096	3	882175	877191	902819	884669	943625

`mknod()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	20576	1	20238	20559	20497	20449	21138
3000	63662	1	62797	63951	62880	63341	65339
5000	110625	1	109593	110643	109659	111113	112118
7000	162864	1	162141	163776	161846	164008	162548
9000	215915	1	217099	215216	213981	218520	214761
10000	248153	1	249665	247218	243865	249949	250067

`readdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	8899	1	8818	8848	8880	8816	9132
3000	28315	1	28187	28302	28270	27841	28975
5000	48746	1	48611	48508	49070	48016	49526
7000	69034	1	68902	68631	69542	68015	70080
9000	89585	2	90469	89241	90471	86298	91447
10000	101487	1	100624	101018	101630	100928	103236

`rename()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	29988	1	29632	29745	29856	30029	30677
3000	92580	1	91930	92247	92684	91335	94703
5000	160137	1	159501	160302	159191	159409	162280
7000	228728	1	227841	228202	229137	226805	231653
9000	298093	1	298378	296656	298776	293363	303290
10000	335168	1	334644	333947	335140	332490	339618

`rmdir()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	31433	2	30851	31424	31508	31029	32352
3000	174350	2	172234	173987	174864	170480	180185
5000	464093	2	462764	460145	461619	457877	478059
7000	945254	2	947333	937128	935834	920535	985442
9000	1909411	2	1897491	1900280	1888710	1884240	1976336
10000	2636695	2	2614489	2618155	2620436	2596906	2733488

`stat()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	6141	2	6064	6091	6071	6043	6434
3000	18872	2	18643	18674	18802	18585	19658
5000	31813	2	31549	31782	31649	31104	32983
7000	45311	1	45153	45016	45949	44411	46025
9000	59721	2	60356	59843	59573	57745	61089
10000	67136	2	66009	66797	66223	66615	70037

`unlink()`

Anfragen	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
1000	18549	1	18312	18468	18530	18409	19026
3000	77441	1	77192	77823	76913	76401	78878
5000	164323	1	164254	164667	164139	162951	165604
7000	280875	0	280949	280381	280905	278898	283241
9000	475177	0	475194	475678	475292	471638	478083
10000	611799	0	611843	614273	611549	607519	613809

A.2 Messungen mit steigender Verzeichnistiefe

- ext3 Dateisystem

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	56	3	56	55	56	55	59
1	56	0	56	56	56	56	56
2	57	0	57	57	57	57	57
6	59	1	58	59	59	59	59
10	61	0	61	61	61	61	61
20	66	1	66	66	65	66	66

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	33	1	33	34	33	33	33
1	34	0	34	34	34	34	34
2	34	1	34	35	34	34	34
6	36	1	36	37	36	36	36
10	38	0	38	38	38	38	38
20	43	0	43	43	43	43	43

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1	45	1	1	2	1	1
1	1	0	1	1	1	1	1
2	1	0	1	1	1	1	1
6	1	45	2	1	1	1	1
10	1	0	1	1	1	1	1
20	1	63	1	2	1	2	1

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	39	0	39	39	39	39	39
1	40	1	40	40	40	41	40
2	41	0	41	41	41	41	41
6	45	1	44	45	45	45	45
10	48	1	48	49	48	48	48
20	58	0	58	58	58	58	58

[1] Mittelwert über allen Messdurchgängen, in Millisekunden
[2] Mittlere Abweichung vom Mittelwert, in %

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	60	87	34	34	165	35	34
1	87	73	35	173	35	155	35
2	35	1	35	36	35	35	35
6	37	1	37	37	37	38	37
10	39	0	39	39	39	39	39
20	105	72	44	183	44	210	44

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	5	0	5	5	5	5	5
1	6	0	6	6	6	6	6
2	6	0	6	6	6	6	6
6	8	0	8	8	8	8	8
10	10	4	10	10	11	10	10
20	15	0	15	15	15	15	15

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	26	7	25	25	29	25	28
1	25	0	25	25	25	25	25
2	26	0	26	26	26	26	26
6	28	5	31	27	27	28	28
10	31	5	30	29	32	29	33
20	39	22	35	56	34	37	34

- FUSE-Loopback Dateisystem

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	193	2	199	197	190	188	191
1	195	1	192	194	198	195	197
2	193	1	192	193	193	193	196
6	204	1	206	202	205	203	204
10	218	1	215	215	223	219	216
20	240	1	241	242	238	237	244

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	171	2	175	176	168	167	168
1	174	3	178	176	179	168	169
2	181	1	179	180	180	182	182
6	191	1	190	189	191	191	192
10	201	1	200	200	203	201	203
20	226	1	225	228	224	223	229

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2	0	2	2	2	2	2
1	2	0	2	2	2	2	2
2	2	0	2	2	2	2	2
6	2	0	2	2	2	2	2
10	2	22	2	3	2	2	2
20	3	0	3	3	3	3	3

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	241	2	250	246	237	237	237
1	250	1	250	247	254	252	249
2	254	0	252	254	253	255	254
6	270	1	271	270	268	273	270
10	287	1	284	284	292	287	288
20	327	1	324	325	333	326	328

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	127	6	136	134	129	117	117
1	133	4	128	130	131	142	132
2	141	2	142	143	143	143	135
6	151	3	141	152	154	153	153
10	157	2	154	162	154	155	162
20	171	2	167	168	174	173	172

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	50	2	52	50	50	49	49
1	52	1	53	52	52	52	53
2	53	2	52	52	52	53	54
6	58	1	59	58	58	58	58
10	64	2	64	62	66	65	64
20	77	2	76	76	78	77	79

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	120	4	125	126	116	115	116
1	130	1	129	130	130	129	132
2	133	1	134	133	134	131	133
6	142	1	139	142	141	142	145
10	149	2	145	148	153	146	153
20	167	2	163	163	169	169	170

- ext3 Dateisystem via NFS

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	3442	25	3151	4081	1866	4065	4049
1	3423	22	3864	3857	1969	3447	3976
2	3861	6	3996	3372	3965	3993	3981
6	6066	2	5933	5893	6166	6067	6270
10	6951	1	6987	6983	6968	6958	6859
20	10719	1	10748	10882	10643	10544	10780

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	4209	4	4289	4297	3901	4272	4284
1	3653	19	4001	4001	2263	4001	3997
2	3710	16	4001	4000	2548	4000	4001
6	7074	12	6214	6004	7325	7900	7929
10	7414	6	7852	7829	7415	6609	7364
20	10731	6	11015	10149	11447	11332	9713

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	38	1	39	38	38	38	38
1	39	0	39	39	39	39	39
2	39	0	39	39	39	39	39
6	41	1	41	40	41	41	41
10	43	1	43	42	43	43	43
20	46	0	46	46	46	46	46

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	3637	26	4090	4064	1746	4153	4131
1	1964	7	2007	2003	1700	2000	2111
2	3314	26	2630	3995	1950	4000	3993
6	4965	12	4371	5543	4613	5862	4438
10	10730	8	9118	11148	11602	10493	11288
20	17975	3	17565	17847	19060	17965	17437

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2032	19	2665	2002	1449	2005	2038
1	2479	18	2679	2683	1610	2694	2727
2	2840	20	2663	2755	3767	2037	2978
6	5092	25	3986	3986	4678	7370	5438
10	6850	8	7810	6461	6459	7017	6503
20	10101	4	9739	10606	9841	9774	10546

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	262	1	260	258	264	262	264
1	296	1	295	293	301	293	297
2	338	1	336	335	345	336	338
6	742	1	746	740	736	739	748
10	4364	0	4364	4364	4366	4364	4364
20	8001	0	8000	8000	8000	8000	8005

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2188	20	2880	2200	1501	2139	2221
1	2531	21	2936	2867	1501	2680	2670
2	2267	25	3393	2065	1824	2027	2027
6	3939	10	3660	3676	3889	4747	3724
10	6338	10	5699	5703	7347	6786	6153
20	10303	6	10737	9585	9792	11160	10240

- ZIB-DMS mit DummyDV via NFS

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	3270	1	3231	3285	3236	3321	3277
1	4598	2	4501	4567	4476	4749	4698
2	6018	1	5999	6088	5945	6046	6011
6	10955	1	10905	11186	10912	10956	10816
10	15905	0	15863	15948	15922	15926	15868
20	28888	1	29059	28817	28784	29198	28581

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	6856	1	6708	6891	6799	6964	6917
1	9034	1	8965	9016	8989	9162	9039
2	11326	2	11073	11342	11147	11585	11483
6	20433	1	20273	20801	20334	20372	20384
10	29256	0	29121	29290	29235	29304	29328
20	52769	0	52482	52892	52603	53137	52733

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	800	3	760	810	805	819	808
1	1001	0	996	1006	996	1007	1000
2	1261	1	1250	1269	1261	1265	1260
6	2175	0	2163	2184	2167	2177	2184
10	3098	0	3079	3093	3103	3121	3095
20	5314	0	5287	5321	5311	5318	5335

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	5858	3	5614	5725	5958	6005	5989
1	7730	2	7603	7695	7587	7911	7855
2	9146	0	9178	9128	9204	9091	9130
6	18161	1	17997	18609	18092	18038	18071
10	26334	1	26043	26166	26934	26319	26209
20	47694	0	47477	48121	47694	47697	47482

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2568	3	2476	2613	2496	2662	2594
1	3610	4	3444	3502	3519	3819	3768
2	4792	2	4730	5004	4777	4767	4682
6	8894	1	8892	9046	8747	8750	9036
10	12704	0	12652	12761	12742	12728	12637
20	23597	1	23672	23453	23815	23771	23275

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1236	4	1152	1239	1244	1281	1263
1	1625	1	1606	1621	1612	1644	1641
2	2141	1	2130	2140	2130	2181	2123
6	5205	1	5169	5324	5173	5186	5172
10	7498	1	7461	7469	7506	7592	7464
20	13900	1	13829	14061	13880	13866	13864

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2710	3	2605	2629	2768	2816	2730
1	3849	2	3860	3878	3682	3931	3894
2	4867	1	4808	5008	4831	4863	4823
6	8851	1	8740	9078	8790	8836	8809
10	12902	0	12877	12945	12904	12925	12858
20	23630	1	23562	23692	23487	23881	23527

- XtreemFS

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	3870	4	3958	4053	3673	3993	3675
1	3909	1	3928	3946	3881	3880	3912
2	3948	5	3877	4153	3682	3840	4188
6	3914	4	3760	4093	4069	3799	3847
10	3958	3	3833	3979	4165	3871	3943
20	4200	2	4348	4205	4079	4134	4235

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	12449	13	14540	10407	12633	10730	13935
1	11376	14	13747	10015	11209	9329	12581
2	10956	13	13452	9457	11256	9483	11131
6	10937	11	12122	9607	12406	9708	10842
10	10871	9	11761	9398	12058	10002	11136
20	12166	13	13733	11073	14512	10742	10768

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	15	6	14	16	16	16	15
1	16	7	15	17	15	16	18
2	16	6	15	17	17	17	16
6	22	3	21	22	22	21	22
10	28	2	28	27	29	28	28
20	43	3	42	45	42	43	43

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	4010	4	4188	4185	3813	4073	3791
1	3878	4	4059	4069	3725	3782	3755
2	3973	3	3806	4117	4050	3825	4066
6	4094	2	4135	4071	4194	3915	4157
10	4088	3	3955	4123	4291	4021	4048
20	4293	1	4302	4319	4176	4319	4349

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2656	4	2684	2816	2676	2576	2526
1	2621	3	2713	2541	2661	2539	2650
2	2649	3	2530	2741	2653	2614	2707
6	2664	3	2576	2743	2787	2586	2630
10	2660	0	2650	2662	2653	2655	2682
20	2845	1	2903	2839	2803	2824	2855

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1413	3	1463	1442	1378	1431	1351
1	1388	3	1435	1440	1359	1347	1357
2	1414	2	1394	1415	1456	1374	1432
6	1451	3	1440	1513	1494	1426	1384
10	1478	3	1413	1464	1559	1470	1483
20	1573	4	1538	1689	1521	1550	1568

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	15908	17	17280	12409	20254	15485	14110
1	12597	14	12449	11188	15116	10242	13989
2	13713	24	15718	8483	17203	11188	15973
6	12602	25	11748	10880	18531	12517	9336
10	12724	18	10905	10211	16698	12813	12991
20	15606	19	11463	14948	14268	16996	20355

- ZIB-DMS mir DummyDV

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1929	1	1935	1938	1921	1906	1943
1	2594	1	2589	2591	2597	2570	2622
2	3357	1	3353	3355	3366	3330	3383
6	6041	1	6006	6036	6115	5992	6056
10	8584	0	8579	8578	8597	8566	8598
20	15236	0	15229	15279	15249	15164	15259

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1978	1	1981	1979	1967	1943	2019
1	2593	1	2589	2601	2591	2569	2617
2	3358	1	3339	3355	3362	3335	3397
6	5943	0	5942	5951	5954	5906	5962
10	8494	0	8518	8494	8508	8442	8510
20	15179	0	15197	15151	15235	15091	15221

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	4	0	4	4	4	4	4
1	5	0	5	5	5	5	5
2	6	0	6	6	6	6	6
6	12	0	12	12	12	12	12
10	22	0	22	22	22	22	22
20	60	1	61	60	60	60	60

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2667	3	2769	2658	2602	2558	2748
1	3460	1	3443	3469	3442	3434	3512
2	4492	1	4451	4510	4483	4491	4526
6	7971	1	8114	7944	7939	7860	7997
10	11330	0	11411	11338	11323	11247	11333
20	20175	0	20143	20156	20228	20137	20213

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1218	1	1218	1219	1206	1211	1235
1	1663	1	1646	1661	1659	1659	1691
2	2137	0	2123	2132	2151	2131	2146
6	3813	0	3787	3815	3814	3827	3824
10	5500	0	5504	5494	5513	5488	5503
20	9891	0	9908	9886	9929	9844	9889

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	653	1	651	653	652	649	661
1	832	0	834	832	834	825	837
2	1083	1	1079	1080	1079	1084	1095
6	1921	0	1926	1921	1923	1909	1924
10	2795	1	2795	2775	2775	2842	2789
20	5066	2	5226	5045	5007	4999	5052

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1280	2	1268	1282	1266	1255	1331
1	1713	1	1703	1721	1707	1705	1731
2	2201	0	2187	2203	2202	2193	2219
6	3867	0	3874	3873	3867	3843	3879
10	5596	1	5582	5551	5559	5678	5611
20	10005	1	10289	9956	9930	9892	9960

- ZIB-DMS mit sqlite-MDC

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	29984	7	29063	32403	32698	27783	27975
1	34936	4	34069	37718	35307	33973	33615
2	41441	4	40015	44552	42400	40245	39994
6	65869	2	65100	67642	67767	64635	64200
10	92760	1	92539	94675	93177	91861	91547
20	182282	0	182773	183636	181513	181808	181682

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	21178	7	20142	22962	22892	19626	20268
1	21865	7	21020	22977	24352	20250	20727
2	22405	6	21508	24425	23307	21328	21456
6	26509	5	25241	28494	27553	25705	25554
10	30452	5	29253	32145	32241	29093	29528
20	41672	2	40969	42977	42441	40935	41036

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	9015	1	9071	9071	8988	9039	8907
1	9283	0	9284	9305	9242	9333	9251
2	9643	0	9684	9702	9592	9627	9609
6	10540	0	10603	10573	10495	10538	10492
10	11513	0	11571	11515	11435	11508	11538
20	13848	0	13905	13866	13779	13841	13848

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	30775	5	30100	32856	32248	29226	29443
1	31996	6	31588	35210	32493	30385	30306
2	32479	4	31818	34705	32871	31430	31571
6	37776	3	36796	39649	39049	36772	36616
10	43425	2	42920	44074	44987	42385	42761
20	62254	3	60868	64235	64412	60401	61352

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	31661	4	31513	33144	33205	30415	30026
1	32104	3	31726	33959	32238	31274	31323
2	33288	3	32875	34997	33521	32703	32344
6	37237	1	37188	37935	37764	36722	36578
10	41421	1	40961	42440	41261	40988	41455
20	54696	1	54468	56138	54556	54248	54072

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	5869	1	5937	5831	5872	5911	5792
1	6176	1	6179	6196	6209	6205	6092
2	6548	0	6606	6531	6557	6532	6512
6	7763	1	7770	7773	7707	7830	7736
10	9045	1	9217	8996	9005	8999	9009
20	13020	0	13107	13024	13022	12912	13035

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	18503	2	18536	18498	18977	18457	18047
1	19574	1	19769	19695	19811	19345	19250
2	20379	2	20098	21023	20468	20223	20083
6	23037	2	22604	23506	23822	22504	22748
10	26067	2	25727	26116	26977	25839	25675
20	35302	1	34968	35745	35903	34884	35011

- ZIB-DMS mit DummyDV via FUSE-Client

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2916	0	2926	2931	2922	2906	2893
1	3624	0	3616	3622	3637	3624	3621
2	4414	1	4405	4439	4407	4370	4447
6	7086	1	7032	7204	7064	7036	7093
10	9655	1	9650	9762	9659	9545	9659
20	16493	0	16438	16599	16483	16425	16518

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	2885	0	2911	2871	2890	2879	2875
1	3602	0	3604	3592	3623	3593	3598
2	4357	1	4348	4399	4349	4333	4358
6	6951	1	6941	7015	6968	6887	6942
10	9592	1	9552	9727	9555	9496	9629
20	16439	1	16341	16589	16392	16414	16459

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	9	0	9	9	9	9	9
1	10	0	10	10	10	10	10
2	11	4	11	11	11	11	12
6	18	2	18	18	19	18	18
10	29	0	29	29	29	29	29
20	70	0	70	70	70	70	70

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	3524	0	3532	3528	3527	3527	3505
1	4488	0	4506	4473	4490	4501	4472
2	5474	0	5459	5503	5481	5446	5480
6	8963	1	8902	9057	8943	8979	8935
10	12401	1	12311	12569	12337	12412	12376
20	21530	1	21377	21670	21637	21323	21641

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1649	3	1638	1744	1633	1622	1606
1	2120	2	2099	2194	2117	2091	2097
2	2620	1	2589	2650	2601	2681	2580
6	4360	1	4366	4373	4448	4288	4324
10	6054	1	5976	6123	6020	6012	6137
20	10527	0	10477	10618	10522	10484	10534

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	807	1	816	802	813	803	803
1	1046	0	1048	1045	1052	1041	1043
2	1303	1	1296	1329	1303	1294	1293
6	2180	1	2187	2214	2176	2155	2167
10	3064	0	3059	3092	3057	3064	3050
20	5382	0	5394	5404	5367	5353	5390

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	1683	1	1702	1678	1691	1672	1672
1	2164	1	2176	2156	2173	2173	2142
2	2652	1	2660	2647	2681	2643	2631
6	4371	1	4350	4434	4371	4333	4366
10	6049	1	6026	6137	6029	6019	6034
20	10569	0	10533	10635	10538	10573	10565

- ZIB-DMS mit sqlite-MDC via FUSE-Client

`mkdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	29749	1	29869	29147	29916	29965	29846
1	36033	1	36166	35535	36217	36228	36017
2	42253	1	41810	41702	42531	42708	42516
6	68375	1	67947	67976	68830	68892	68230
10	97259	1	95803	97141	98294	97542	97516
20	188825	0	188042	187913	189895	189511	188763

`mknod()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	20382	0	20300	20274	20433	20465	20436
1	21509	1	21564	21217	21559	21619	21588
2	22636	1	22465	22459	22676	22815	22765
6	26599	1	26612	26214	26624	26820	26726
10	30708	1	30458	30533	30643	30940	30966
20	43545	1	43318	43199	43664	43907	43639

`readdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	8861	0	8858	8863	8877	8887	8821
1	9220	0	9241	9191	9227	9264	9178
2	9589	0	9585	9569	9628	9623	9542
6	10552	0	10543	10500	10583	10619	10515
10	11523	1	11336	11505	11592	11615	11568
20	13919	0	13901	13885	13961	13987	13862

`rename()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	29889	0	29918	29735	30072	29930	29788
1	31768	1	31889	31401	31818	31953	31780
2	33107	1	33238	32802	33248	33059	33188
6	38976	0	38902	38713	38982	39065	39220
10	44764	1	43771	44737	45122	45174	45016
20	63274	1	62976	62221	63767	63676	63728

`rmdir()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	31303	0	31329	31029	31410	31453	31295
1	32749	1	32827	32298	32805	32987	32826
2	34492	1	34399	33979	34692	34899	34493
6	38488	1	38443	38155	38469	38804	38567
10	43104	1	42560	42834	43349	43523	43253
20	57017	1	56724	56500	57309	57328	57223

`stat()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	6109	0	6113	6120	6120	6136	6055
1	6445	0	6440	6459	6487	6449	6389
2	6840	1	6808	6842	6890	6872	6790
6	8101	1	8060	8103	8095	8225	8022
10	9469	2	9033	9553	9611	9649	9499
20	13774	1	13717	13669	13830	13877	13777

`unlink()`

Tiefe	Mittel[1]	Abw.[2]	Run 1	Run 2	Run 3	Run 4	Run 5
0	18436	0	18362	18411	18327	18566	18516
1	19716	1	19541	19698	19662	19762	19918
2	20604	1	20372	20442	20608	20807	20791
6	23275	0	23203	23197	23228	23365	23381
10	26499	1	25882	26478	26777	26662	26698
20	36224	1	35961	35943	36374	36500	36341