Skalierbarkeit von Cluster-Dateisystemen durch Verteilung der Metadaten

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik

der Universität Karlsruhe (Technische Hochschule)

genehmigte

Dissertation

von

Guido Malpohl

aus Solingen

Tag der mündlichen Prüfung:	9. November 2007
Erster Gutachter:	Prof. Dr. Walter F. Tichy
Zweiter Gutachter:	Prof. Dr. Frank Bellosa

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Noch vor 8 Jahren galten Parallelrechner auf der Basis von Standardprozessoren (Cluster) als Exoten mit geringer Leistungsfähigkeit. Zur Lösung großer wissenschaftlicher Probleme wurden teure Parallelrechner mit Spezialhardware benutzt. Diese Systeme waren, besonders auf Grund der von allen Prozessoren erreichbaren Peripherie (Speicher und Festplatten), aus Anwendersicht leicht handhabbar und stellten sich dem Programmierer als ein geschlossenes System (Single System Image, SSI) dar.

Weil jedoch die Arbeitsplatzrechner einen großen Boom erlebten und daher auch mit große Budgets an der Weiterentwicklung der verwendeten Komponenten gearbeitet wurde, hatten diese bald in ihrer Leistungsfähigkeit die Spezialprozessoren überholt. Doch der eigentliche Grund, warum sich Standardprozessoren auch im Bereich der Hochleistungsrechner durchgesetzt haben, ist ihr -im Vergleich zu den Spezialprozessoren- unschlagbares Preis/Leistungs-Verhältnis und gleichzeitig die Entwicklung von schnellen Netzwerken mit geringer Latenz und hohem Durchsatz, die eine effektive Zusammenarbeit der Einzelrechner ermöglichen.

Dieser Trend zeigt sich auch deutlich in der seit 1993 halbjährlich erscheinenden TOP500 Liste [56]. In dieser Liste, die den Anspruch erhebt, die 500 schnellsten Parallelrechner der Welt zu enthalten, tauchten Cluster erstmals im Jahre 1998 auf. Von diesem Zeitpunkt an, stieg ihr Anteil an den 500 schnellsten Rechnern stetig auf 72.20% (im November 2006), während die Anteile der anderen Architekturen abnahmen (Abbildung 1.1). Doch nicht nur im Höchstleistungsbereich sind Cluster sehr erfolgreich. Auf Grund der geringen Kosten und der relativ einfachen Erweiterbarkeit entstehen auch immer mehr kleine und mittlere Installationen, die nur aus einigen Dutzend Knoten bestehen.

Trotz ihres Erfolgs haben Clustersysteme den großen Nachteil, dass sie sich dem Anwender, entsprechend ihrer Architektur, nur als eine Ansammlung von Einzelrechnern darstellen, die über ein Netzwerk verbunden sind. Der Programmierer muss zur Lösung seiner Rechenprobleme, diese auf die vorhandenen Rechner aufteilen und mittels expliziter Kommunikationsanweisungen eine Zusammenarbeit herbeiführen. Denn obwohl in einem solchen Rechnerbündel eine große Menge an Ressourcen, insbesondere Haupt- und Festplattenspeicher, zur Verfügung stehen, sind diese nicht von jedem Knoten aus erreichbar. Im Gegensatz zu den eingangs erwähnten spezialisierten Parallelrechnern, bieten Clustersysteme keine einheitliche Verwaltung dieser Ressourcen, erschweren dadurch

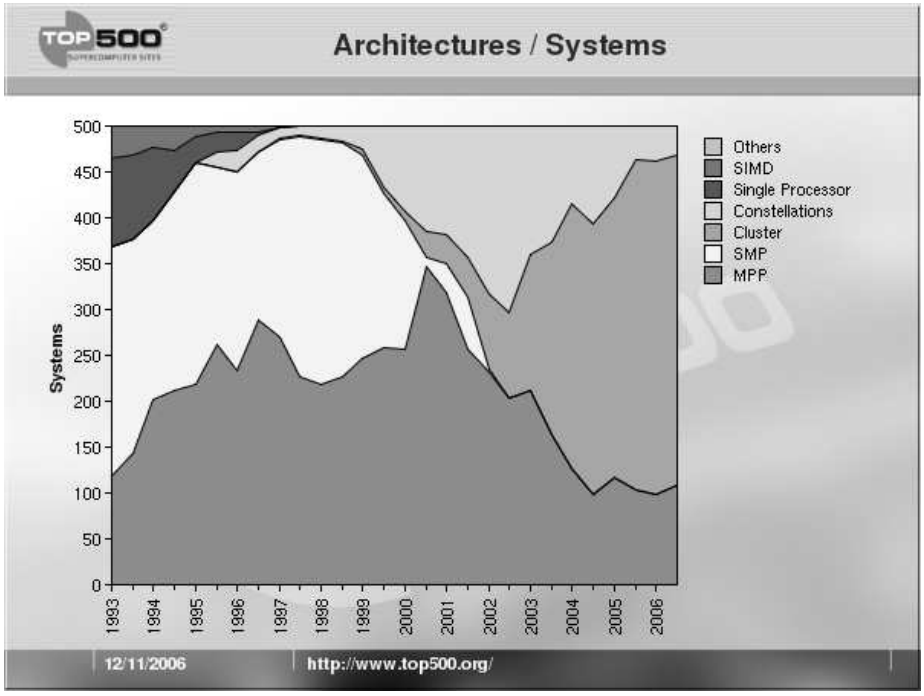


Abbildung 1.1: Top500 Supercomputer: Architekturen [56]

ihre Programmierung und Administration und bleiben aus diesen Gründen unter ihrer potentiellen Leistungsfähigkeit.

Ziel dieser Arbeit war die Weiterentwicklung des parallelen Dateisystems (*Clusterfile*) zu dem Cluster-Dateisystem *CLF* (Cluster Filesystem), welches durch Kooperation der einzelnen Knoten die Skalierbarkeit der Metadatenverarbeitung verbessert, und darüber hinaus von jedem Knoten des Clusters einen einheitlichen Zugriff auf die gespeicherten Daten ermöglicht. Es soll nicht nur von parallelen Programmen genutzt werden, sondern vor allem auch von lokalen, ein-Prozessor-Programmen, die für den Betrieb eines einzelnen Knoten notwendig sind.

1.1 Architektur von *Clusterfile*

In diesem Abschnitt wird zunächst die Architektur von *Clusterfile* vorgestellt, das der Ausgangspunkt für die Arbeiten an *CLF* war und ein typisches Beispiel für die Architektur eines parallelen Dateisystems ist.

Clusterfile [44, 45] wurde als Dateisystem für parallele Programme entworfen und folgt der klassischen Architektur paralleler Dateisysteme, in der zwischen drei verschiedenen Rollen unterschieden wird, welche die einzelnen Knoten im Cluster einnehmen können: *Metadatenmanager*, *Daten-Server* und *Klient*. Die Trennung zwischen Metadatenmanager und Daten-Server unterscheidet diesen Aufbau von den traditionellen Netzwerkdateisystemen, wie beispielsweise NFS [70]. Bei NFS gibt es nur einen einzigen Dienstgeber, der einen Teil seines lokalen Dateisystems an die angeschlossenen Klienten exportiert. Da Anfragen

an das Dateisystem von einem Dienstgeber nur sequentiell verarbeitet werden können, kann er, besonders bei gleichzeitigem Zugriff von mehreren Klienten, zu einem Engpass werden. Diesem Problem treten parallele Dateisysteme durch eine Aufteilung des Dienstes entgegen.

Unter *Metadaten* versteht man im Allgemeinen *Daten über Daten*, bei Dateisystemen sind dies unter anderem die Verzeichnishierarchie und Informationen über die in ihr gespeicherten Dateien. Anders ausgedrückt: Metadaten sind alle Informationen in einem Dateisystem mit Ausnahme des Inhalts der Dateien. Der Metadatenmanager ist somit ausschließlich für die Verarbeitung dieser Verwaltungsinformationen verantwortlich. Es ist wichtig, dass diese Daten in einem konsistenten Zustand gehalten werden, da sonst die Integrität des Dateisystems gefährdet ist. Aus diesem Grund besitzt *Clusterfile*, wie auch die übrigen parallelen Dateisysteme, nur einen einzigen Metadatenmanager, der die an ihn gestellten Anfragen nacheinander bearbeitet.

Der eigentliche Inhalt der Dateien wird auf den Daten-Servern gespeichert, von denen es in einer Dateisystemkonfiguration beliebig viele geben kann. Die Dateien werden dazu in Blöcke gleicher Größe (üblicherweise 64KB) aufgeteilt und reihum auf den Daten-Servern gespeichert. Der Index des ersten Daten-Servers wird variiert, um eine gleichmäßige Lastverteilung und Speicherbelegung zu erreichen. Die Anfragen zum Schreiben oder Lesen von Datenblöcken stellen die Klienten direkt an die Daten-Server.

Darüber hinaus bietet *Clusterfile* auch die Möglichkeit Daten mittels einer flexiblen Funktiondefinition (sogenannte *PITFALLS*) fast beliebig auf die Daten-Server zu verteilen [46, 47]. Auf die flexible Daten-Verteilung wird hier nicht weiter eingegangen, da sie hauptsächlich bei parallele Anwendungen zum Einsatz kommt, während sich diese Arbeit auf den Einsatz des Dateisystems bei Ein-Prozessor-Software konzentriert.

Die dritte Rolle, die Knoten im Cluster bezüglich des Dateisystems einnehmen können, ist die des Klienten. Der Zugriff auf die Dateien erfolgt entweder über eine spezielle Bibliothek, die ähnliche Funktionalität bietet wie die Dateisystem-Schnittstelle des Betriebssystems, mit dem Unterschied, dass den Funktionen der Bibliothek der Präfix „clf_“ vorangestellt wird. In Anwendungen, die *Clusterfile* benutzen möchten, müssen Funktionsaufrufe wie beispielsweise `open()` und `read()` durch die *Clusterfile*-Äquivalente `clf_open()` respektive `clf_read()` ersetzt werden. Auf Basis dieser Bibliothek kann auch über eine MPI-IO Schnittstelle auf *Clusterfile* zugegriffen werden. MPI-IO ist eine auf parallele Anwendungen zugeschnittene Bibliothek, welche MPI-Anwendungen den effizienten Zugriff auf parallele Dateisysteme erlaubt. Sie bildet einen Teil des MPI2-Standards [34], einer Erweiterung von MPI1.1 [83].

Transparent für den Benutzer muss die *Clusterfile*-Bibliothek beim Zugriff auf Dateien zunächst Kontakt mit dem Metadatenmanager aufnehmen, um festzustellen, ob die gewünschte Datei existiert und um die Verteilung der Datei auf den Daten-Servern zu erfragen. Im Anschluss werden die benötigten Daten direkt bei den involvierten Daten-Servern angefordert. Falls bei der Operation mehrere Daten-Server involviert sind, so werden diese gleichzeitig kontaktiert, um eine möglichst schnelle Bearbeitung der Anfrage zu ermöglichen.

Abbildung 1.2 zeigt eine mögliche Konfiguration für einen Cluster mit 5 Knoten: Das Dateisystem wird von allen Knoten benutzt, wodurch alle die Klienten-Rolle annehmen. Die ersten drei Knoten spielen jedoch besondere Rollen. Knoten 1 ist Metadatenmanager und die Knoten 2 und 3 sind Daten-Server. Diese

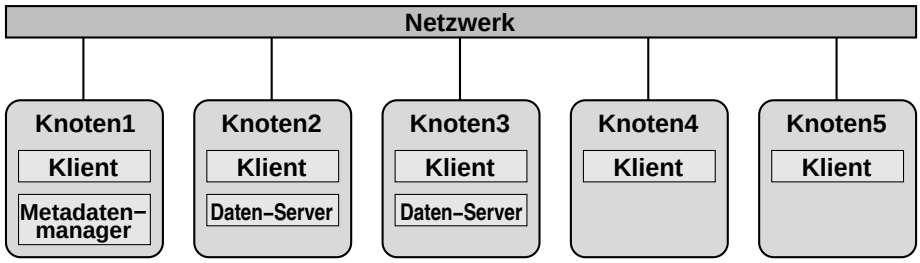


Abbildung 1.2: Beispiel für verschiedene Knotenrollen in *CLF*

drei Knoten müssen über eine lokale Festplatte verfügen, um die anfallenden Daten zu speichern, während die übrigen Knoten keine eigene Festplatte benötigen. Je nach Größe des Clusters und der Ausstattung der einzelnen Knoten können die Dienstgeber auch auf dedizierten Knoten ablaufen, welche selber das Dateisystem nicht benutzen, um eine höhere Leistung zu erreichen.

Das Dateisystem ist vollständig in der Programmiersprache *C* auf der Benutzerebene implementiert. Die Dienstgeber benutzen jeweils die lokalen Dateisysteme auf ihren Knoten, um die anfallenden Daten zu speichern und verwenden eigene Caching-Mechanismen zur Vermeidung von Systemaufrufen. Wie bereits erwähnt, handelt es sich bei der Klientensoftware um eine Bibliothek, welche die Schnittstelle des Betriebssystems imitiert. Die Bibliothek wird an die Programme, die das Dateisystem benutzen, gebunden. Verwenden n Programme auf einem Knoten *Clusterfile*, so werden n Instanzen des Klienten erzeugt. So gesehen ist die Abbildung 1.2 ungenau, da es pro Knoten mehrere „Klienten"-Komponenten geben kann, oder für den Fall, dass kein Programm abläuft, dass *Clusterfile* benutzt, auch gar keine.

Die Kommunikation zwischen den einzelnen Komponenten wurde in *Clusterfile* direkt auf der Basis von TCP Sockets [30] implementiert. Auf Clustern werden zur Kommunikation über das Hochgeschwindigkeitsnetzwerk spezielle Betriebssystemmodule verwendet, die es erlauben, TCP-Verbindungen statt der proprietären Schnittstellen zu verwenden. So ermöglicht beispielsweise *Sockets-GM* [32] die Kommunikation über das Myrinet-Netzwerk mittels Sockets, bei Infiniband [42] ist durch das *Sockets Direct Protocol (SDP)* [66] sogar die transparente Nutzung von *Remote Direct Memory Access (RDMA)* bei der Benutzung der Standard Unix Socket-Schnittstelle möglich.

1.2 Ziele

In dieser Arbeit wurde das Ziel verfolgt ein Clusterdateisystem (*CLF*) zu entwickeln, dessen Anwendungsbereich vor allem die nicht-parallelen Programme sind. Nicht-parallele Programme benutzen in der Regel die Dateisystem-Schnittstelle des Betriebssystems, daher war es eine wichtige Voraussetzung, das Dateisystem unter Linux montierbar zu machen. Dieses Ziel schränkt den Entwurfsspielraum bei der Entwicklung aufgrund der Rahmenbedingungen der zu erfüllenden Schnittstelle ein. Um die Anwendbarkeit von *CLF* in nicht-parallelen Programmen zu verbessern, sollte vor allem die Skalierbarkeit der Metadatenverarbeitung verbessert werden. Die Arbeit wird dabei motiviert durch Schwä-

chen in der Architektur der bisherigen parallelen Dateisysteme, wie sie im Abschnitt 1.1 vorgestellt wurde.

Parallele Applikationen verwenden in einem Dateisystem normalerweise wenige große Dateien, die häufig gleichzeitig von mehreren Rechenknoten gelesen und geschrieben werden [59, 76]. Die bisherigen Arbeiten an *Clusterfile* konzentrierten sich auf diesen Anwendungsbereich [44]. Für die Verwendung in nichtparallelen Anwendungen muss das Dateisystem jedoch in der Lage sein, eine Hierarchie von Tausenden von Dateien jeglicher Größe effizient zu verwalten. Dadurch kann ein einzelner Metadatenmanager, wie er üblicherweise von parallelen Dateisystemen verwendet wird, schnell zu einem Engpass werden, der die Leistungsfähigkeit beeinträchtigt. Der Grund dafür liegt in den häufig notwendigen Anfragen der Klienten: Klienten müssen bei nahezu jeder Operation des Dateisystems den Metadatenmanager kontaktieren um aktuelle Informationen zu erhalten oder eine Änderung vorzunehmen. Eine solche Änderung ist beispielsweise das Anlegen eines Verzeichnisses oder der Zugriff auf eine Datei.

Zunächst muss die Frage beantwortet werden, ob eine Aufteilung der Metadaten und damit auch deren verteilte Verwaltung überhaupt möglich ist, ohne die Rahmenbedingungen der Dateisystemschnittstelle des Betriebssystems zu verletzen. Der in dieser Arbeit vorgestellte Entwurf und die zugehörige Implementierung belegen die folgende These:

These 1: *In einem Cluster-Dateisystem ist es möglich, die Verwaltung der Metadaten automatisch auf mehrere Knoten zu verteilen und gleichzeitig die Schnittstelle des Linux-Betriebssystems zu erfüllen.*

Für die automatische Verteilung der Metadaten ist es notwendig, die Verwaltung innerhalb des Metadatenmanagers den neuen Anforderungen anzupassen und eine Datenstruktur zu entwerfen, die eine Verteilung zulässt. Weiterhin muss ein Betriebssystemmodul für den Klienten entworfen werden, welches es erlaubt, *CLF* wie jedes andere Dateisystem auf einem Knoten zu montieren. Damit können Anwendungen nicht nur mittels der Bibliothek, sondern auch über die Dateisystemschnittstelle des Betriebssystems auf *CLF* zugreifen [29, 30].

Auch wenn die grundsätzliche Machbarkeit der Verteilung gezeigt wurde, ist noch offen, ob die Verteilung auch die gewünschten Verbesserungen bei der Skalierbarkeit zeigt. Aufgrund vielfacher Abhängigkeiten zwischen den einzelnen Objekten eines Dateisystems, wird die Verteilung zu einem erhöhten Kommunikationsbedarf zwischen den Verwaltungsknoten führen. So betrifft beispielsweise die Umbenennung einer Datei sowohl die Metadaten der betroffenen Datei, wie auch die Metadaten des Verzeichnisses in dem diese Datei abgelegt ist. Werden die Metadaten beider Objekte von unterschiedlichen Knoten verwaltet, müssen diese bei der Bearbeitung der entsprechenden Anfrage zusammen arbeiten. Diese Zusammenarbeit erfordert zusätzliche Kommunikation, die eine erhöhte Belastung des Kommunikationsnetzwerks und eine erhöhte Bearbeitungszeit zur Folge haben kann.

Diese Arbeit weist nach, dass sich trotz der beschriebenen Probleme bei der Verteilung der Metadaten, folgende These belegen lässt:

These 2: *Die Verteilung der Metadaten auf mehrere Cluster-Knoten steigert bei typischen ein-Prozessor-Anwendungen die Leistungsfähigkeit der Metadatenverwaltung im Vergleich zur Verwendung eines zentralen Metadatenmanagers.*

Um diese These nachzuweisen wurde zunächst das Verhalten der ein-Prozessor-Anwendungen untersucht und anhand von Studien ein Benchmark entwickelt, der in der Lage ist, die Metadatenleistung eines Clusterdateisystems zu vermessen. Der Benchmark läuft gleichzeitig auf mehreren Klienten des Clusters ab und ist somit in der Lage die Skalierbarkeit der Metadatenverarbeitung zu untersuchen, indem er die Anzahl der aktiven Klienten variiert und den erreichten Durchsatz vergleicht.

1.3 Gliederung

Kapitel 2 gibt zunächst einen Überblick über Dateisysteme im Allgemeinen und verteilte Dateisysteme im Speziellen und stellt anschließend die verwandten Arbeiten aus dem Bereich der Cluster-Dateisysteme vor. Dabei steht insbesondere die Metadatenverwaltung im Mittelpunkt der Betrachtung.

In Kapitel 3 werden zunächst die Grundlagen dieser Arbeit gelegt. Dabei handelt es sich um Linux-Dateisysteme, wobei besonderes Augenmerk auf das Linux-Dateimodell und die VFS-Schnittstelle (Virtual Filesystem Switch), sowie die Implementierung von Dateisystemen geworfen wird.

Die im Rahmen dieser Arbeit entwickelte Netzwerkschnittstelle *Paradis-Net* wird in Kapitel 4 vorgestellt. Sie stellt eine wichtige Vorarbeit für die Implementierung der komplexen Kommunikationsmuster dar, die bei der Zusammenarbeit mehrerer Knoten benötigt werden.

Der Entwurf und die Implementierung von *CLF* werden in Kapitel 5 behandelt. Dort werden die einzelnen Komponenten (Klient, Daten-Server, Metadatenmanager und Metadaten-Surrogat), sowie deren Architektur und die verwendeten Kommunikationprotokolle beschrieben.

Die Effektivität des Dateisystems wird in Kapitel 6 untersucht. Anhand von Benchmarks werden unterschiedliche Konfigurationen von *CLF* untereinander und mit dem bekannten Dateisystem NFS verglichen.

Ein abschließendes Fazit und einen Ausblick bietet Kapitel 7.

Kapitel 2

Verwandte Arbeiten

In diesem Kapitel wird zunächst einen Überblick über Dateisysteme und deren Aufgaben im Allgemeinen gegeben. Dabei werden lokale Dateisysteme und verteilte Dateisysteme, sowie deren besondere Eigenschaften und Anwendungsgebiete betrachtet. Der letzte Abschnitt dieses Kapitels beschäftigt sich mit den eng verwandten Arbeiten zur Metadatenverwaltung bei Cluster-Dateisystemen.

2.1 Aufgaben von Dateisystemen

Massenspeichergeräte wie Festplatten-, CD-ROM- und Diskettenlaufwerke besitzen eine Blockstruktur, das heißt aus Sicht des Rechners lassen sich die Daten nur als ganze Datenblöcke lesen oder schreiben. Die Hardware der Speichergeräte präsentiert sich als großer linearer Bereich bestehend aus nummerierten Blöcken. Dateisysteme verwalten diese Blöcke und bieten gleichzeitig einen flexiblen und effizienten Zugriff auf den Speicherplatz. Sie sind traditionell ein Teil des Betriebssystems (Abbildung 2.1) und vermitteln zwischen den Anwendungsprogrammen und den Gerätetreibern.

Der unterhalb des Dateisystems angeordnete Gerätetreiber dient zur Ansteuerung des verwendeten Massenspeichergerätes und bietet nach oben eine einfache Schnittstelle, die das Gerät als zusammenhängenden, linearen Bereich darstellt, der in Blöcke gleicher Größe aufgeteilt wird. Auf diese Blöcke kann

Abbildung 2.1: Einordnung von Dateisystemen in die Systemarchitektur

über die Schnittstelle des Gerätetreibers lesend und schreibend zugegriffen werden.

Die Schnittstelle des Gerätetreibers ist offensichtlich nicht dazu geeignet die Anforderungen eines modernen Rechnersystems zu befriedigen. Die zusätzliche Funktionalitäten, die ein Dateisystem erbringt, werden in den folgenden Abschnitten beschrieben. Dazu gehören unter anderem die *Dateiabstraktion* (Abschnitt 2.1.1) und der *Namensraum* (Abschnitt 2.1.2).

Neben der in Abbildung 2.1 vorgestellten Anordnung gibt es noch eine wichtige Variante, bei der das Dateisystem nicht Teil des Betriebssystems ist, sondern als Benutzerebenenbibliothek zur Verfügung gestellt wird. Diese Variante wird dann verwendet, wenn ein Dateisystem für eine Spezialanwendung entworfen wird und die normierte Schnittstelle des Betriebssystems zu beschränkt ist um die besonderen Anforderungen der Anwendung zu erfüllen. Beispiele für solche Dateisysteme sind beispielsweise das *Google Dateisystem* [36] und *Clusterfile* [44]. Im weiteren Sinne kann auch jede Datenbank als Benutzerebenen-Dateisystem angesehen werden.

Die Benutzerebenen-Dateisysteme besitzen den Vorteil, dass eine Anwendung die speziellen Möglichkeiten eines Dateisystems nutzen kann, wie beispielsweise die beliebige Aufteilung von Daten auf Daten-Server bei *Clusterfile* oder die *record append*-Operation des Google-Dateisystems, die das atomare Anfügen von Datensätzen an eine Datei ermöglicht. Ein Betriebssystem-Dateisystem kann so spezielle Funktionalität nur in sehr begrenztem Umfang anbieten. Auf der anderen Seite bietet es aber eine einheitliche und normierte Schnittstelle (siehe zum Beispiel *POSIX* [21]). Die einheitliche Schnittstelle erlaubt es einem Anwendungsprogramm „das Dateisystem“ zu verwenden, ohne die spezielle Dateisystemimplementierung zu kennen, welche letztendlich die Funktionalität bereitstellt.

Auf diese Weise ist es möglich, dass ein Programm mit den verschiedensten Dateisystemen zusammen arbeiten kann, ohne neu übersetzt werden zu müssen. Im Gegensatz dazu muss eine Anwendung bei Verwendung von Benutzerebenen-Dateisystemen im Allgemeinen mit einer Bibliothek gebunden werden und auf die Funktionen des Dateisystems hin angepasst werden. Dies stellt bei diesen Anwendungen jedoch normalerweise keine Einschränkung dar, weil sie so entworfen wurden, dass sie ohnehin nur mit dem speziellen Dateisystem zusammen arbeiten können und andere Programme keinen Zugriff auf das Dateisystem benötigen.

Im Bereich der parallelen Dateisysteme gibt es Normierungsbestrebungen für Schnittstellen in der Benutzerebene. So bietet MPI-IO [34] eine einheitliche Schnittstelle für den Zugriff auf (parallele) Dateisysteme, die auch Benutzerebenen-Dateisysteme sein können. Sie ermöglicht unter anderem die Definition von *Sichten* auf eine Datei die jedem Prozess einer MPI-Anwendung die Ausschnitte der Datei, für die sie zuständig sind, als linearen Bereich darstellen. Mit Hilfe von MPI-IO ist eine Entkopplung von Anwendung und Dateisystem möglich, die mit der von Betriebssystem-Dateisystemen vergleichbar ist.

Die Ein-Prozessor-Anwendungen, die im Mittelpunkt dieser Arbeit stehen, benutzen jedoch in der Regel Dateisysteme, die über die Schnittstelle des Betriebssystems zugreifbar sind. Der Rest dieses Kapitel beschäftigt sich daher hauptsächlich mit den Dateisystemen, die innerhalb des Betriebssystems implementiert sind, oder zumindest die Standardschnittstelle des Betriebssystems erfüllen. Die folgenden Abschnitte beschreiben das Modell von lokalen Dateisys-

temen und die Technologie, die bei der Implementierung zum Einsatz kommt. (siehe auch [43])

2.1.1 Dateiabstraktion

In einem Dateisystem werden Daten verschiedener Benutzer (bzw. Prozesse) gespeichert, die inhaltlich große Unterschiede aufweisen. Daher wird der Speicherplatz in logische, inhaltlich zusammenhängende Einheiten aufgeteilt, die *Dateien* genannt werden. Eine Datei stellt eine linear adressierbare Folgen von Bytes dar, die durch das Dateisystem auf den zugrunde liegenden physikalischen Speicher abgebildet werden. *Dateien* werden über einen Namen referenziert.

Neben den in der *Datei* enthaltenen Daten speichert ein Dateisystem auch Verwaltungsinformationen, die so genannten *Metadaten*, die üblicherweise in einem separaten Bereich des Speichermediums abgelegt werden. Jede Datei besitzt eine Metadatenstruktur, in der ihre Attribute, wie Dateigröße, Erstellungs-, Änderungs- und Zugriffszeitpunkte, Besitzer und Zugriffsrechte abgelegt sind. Zu diesen Attributen gehören auch die Informationen zum Speicherort des Dateiinhalts. In traditionellen UNIX-Systemen wird diese Struktur *Inode* genannt.

2.1.2 Namensraum

Dateisysteme verwalten ihre Dateien mit Hilfe eines Namensraums, der es ermöglicht Dateien zu eindeutig identifizieren und zu verwalten. Jede Datei hat einen Namen, der als Zeichenkette definiert wird. Die Dateinamen werden über spezielle Dateien (die so genannten *Verzeichnisse*) zu einer baumartigen Struktur verbunden. Verzeichnisse stellen Behälter für weitere Verzeichnisse oder normale Dateien dar und bilden dabei die inneren Knoten des Baumes. Der Pfad von der Wurzel des Baumes bis zu einem Dateiknoten identifiziert die Datei innerhalb des Namensraums eindeutig.

In den frühen UNIX-Dateisystemen wurde der Inhalt der Verzeichnisse linear in einer Datei abgelegt. Daher war wie Dateisuche eine sequentielle Operation, deren Aufwand mit der Größe eines Verzeichnisses skalierte. Neuere Dateisysteme verwenden *B-Bäume* zur Verwaltung der Verzeichnisse, um eine effizientere Suche zu ermöglichen. XFS [24] und JFS [5] verwenden einen *B-Baum* pro Verzeichnis, während ReiserFS [55] den gesamten Namensraum des Dateisystem in einem einzigen *B-Baum* verwaltet.

2.1.3 Abbildung auf das Medium

In den verschieden Dateisystemen kommen unterschiedliche Datenstrukturen bei der Abbildung von Dateiblöcken auf Blöcke des Datenspeichers zur Anwendung. In dem Dateisystem *FAT* [23], das in MS-DOS zum Einsatz gekommen ist, werden die Blöcke einer Datei als eine verkettete Liste verwaltet. Der Nachteil dieser Strukturierung ist, dass wahlfreie Zugriffe auf eine Datei nicht effizient durchgeführt werden können, weil auf der Suche nach einem Dateiblock die Liste sequentiell durchlaufen werden muss.

In den traditionellen UNIX-Dateisystemen [3] (zum Beispiel *MINIX* [82], *EXT* und *EXT2* [16]) werden daher die Blöcke einer Datei in einer Baumstruktur verwaltet, deren Wurzel der Block ist in dem die Inode-Informationen der Datei gespeichert werden. Innerhalb des Blocks gibt es vier Arten von Verweisen:

direkte, indirekte, doppelt-indirekte und dreifach-indirekte Verweise. Die Blöcke kurzer Dateien können über die direkten Verweise gefunden werden (bei EXT2 werden 12 direkte Verweise gespeichert). Bei längeren Dateien wird, sobald keine direkten Verweise mehr verfügbar sind, der indirekten Verweis verwendet. Er verweist auf einen Block, der wiederum Verweise auf die eigentlichen Dateiblöcke enthält. Sind alle indirekten Verweise ausgeschöpft, werden die Blöcke zunächst über doppelte und schließlich über dreifach Indirektion lokalisiert. Der Nachteil dieses Vorgehens ist, dass ein Zugriff auf einen bestimmten Block einer langen Datei mehrere Zugriffe auf das Medium zur Folge hat.

Neuere Dateisysteme, wie beispielsweise *JFS*, *ReiserFS* und *XFS* verwenden effizientere Methoden. Dazu gehört die Verwendung von *Extents*, die eine zusammenhängende Folge von Blöcken beschreiben und statt der Referenzierung einzelner Blöcke verwendet werden. Weiterhin werden die *Extents* und deren Position innerhalb der Datei in *B-Bäumen* verwaltet. Auf der Suche nach einem Dateiblock wird der *B-Baum* durchsucht, bis der entsprechende Block auf dem Medium gefunden wurde. Aus Effizienzgründen werden die *B-Bäume* erst ab einer gewissen Dateigröße verwendet, damit bei kleinen Dateien der Aufwand der Baumverwaltung vermieden werden kann. Weitere Optimierungen zielen darauf ab, bei kleinen Dateien die Zahl der Zugriffe auf das Medium zu verringern: So können die Metadaten und Daten einer solchen Datei innerhalb des gleichen Blocks gespeichert werden, oder sogar mehrere kurze Dateien innerhalb eines Blocks abgelegt werden.

2.1.4 Speicherverwaltung

Die Daten eines Dateisystems werden auf Blockgeräten gespeichert, daher ist die Verwaltung der Information, ob ein bestimmter Block bereits benutzt wird und die Suche nach freien Blöcken eine wichtige Aufgabe eines Dateisystems. Die offensichtliche Lösung mit einem Bit-Feld, dessen Bits die Verwendung der einzelnen Blöcke repräsentieren, wird von den traditionellen UNIX-Dateisystemen verwendet. Die Datenstruktur muss ebenfalls auf dem Medium gespeichert werden und der benötigte Platz wächst mit der Anzahl der zu verwaltenden Blöcke. Der für das Bit-Feld verwendete Platz stellt zusammen mit dem linearen Aufwand einer Suche nach freien Blöcken den größten Nachteil dieser Datenstruktur dar.

Moderne Dateisysteme verwenden wiederum eine Kombination aus *Extents* und *B-Bäumen* um diesen Problemen zu begegnen. Der Vorteil von *Extents* liegt einerseits darin, dass sie weniger Platz auf dem Medium verbrauchen, wenn das Dateisystem nicht zu stark fragmentiert ist und andererseits werden bei der Suche nach freiem Speicher gleich mehrere, aufeinander folgende Blöcke gefunden. Die Kombination aus *B-Bäumen* und *Extents* erlauben es eine bestimmte Zahl zusammenhängender Blöcke zu finden, wenn der *B-Baum* nach der Größe der *Extents* indiziert ist. Andererseits erlaubt die Indizierung nach der Blocknummer das Finden von nahe beieinander liegenden Block-Gruppen, falls beispielsweise eine Datei verlängert werden soll. Nimmt man an, dass das Medium eine Festplatte ist, dann kann auf diese Weise die Zahl der Kopfbewegungen beim Lesen einer Datei vermindert werden.

Das Dateisystem XFS [24] verwendet alle im letzten Abschnitt beschriebenen Techniken zur Speicherverwaltung.

2.1.5 Caches

Beim Lesen aus einer Datei werden die Daten, unabhängig von der Länge der Anfrage blockweise angefordert, weil das zugrunde liegende Gerät dies erfordert (siehe auch Abbildung 2.1). Damit bei wiederholten Zugriffen auf den gleichen Block dieser nicht mehrfach vom Medium gelesen werden muss, wird er in einem *Cache*, einem vom Betriebssystem verwalteten speziellen Speicherbereich abgelegt. Der Cache speichert nicht nur den zuletzt gelesenen Block, sondern eine feste Zahl an Blöcken. Bei einer Anfrage wird zunächst geprüft, ob der angeforderte Block bereits im Cache liegt und gegebenenfalls von dort gelesen.

Die Verwendung von Caches verbessert die Geschwindigkeit von Zugriffen auf das Dateisystem, wenn die Anwendung eine *zeitliche Lokalität* in ihren Zugriffen aufweist. *Zeitliche Lokalität* bedeutet, dass ein Zugriff auf einen Block es sehr wahrscheinlich macht, dass auf diesen Block in naher Zukunft wieder zugegriffen wird. Untersuchungen haben gezeigt, dass die meisten Anwendungen diese Eigenschaft aufweisen [65, 77]. Daher kommen Caches in allen lokalen Dateisystemen zum Einsatz und sind zumeist als Teil des Betriebssystems implementiert.

2.1.6 Spekulatives Laden

Das *spekulative Laden* (engl. *prefetching*) dient, wie auch das Cachen, der Vermeidung von synchronen Lesezugriffen auf das Medium und damit dem Verringern der Zugriffszeiten auf das Dateisystem. Dazu werden Datenblöcke, die voraussichtlich in naher Zukunft benötigt werden, bereits vor dem eigentlichen Zugriff auf diese, in den Cache geladen. Die Technik setzt das Vorhandensein eines Caches voraus und wird dadurch motiviert, dass trotz der Existenz eines Caches, das erstmalige Lesen eines Blockes dennoch einen Zugriff auf das Medium erfordert. Diese Zugriff ist um Größenordnungen langsamer als das Lesen aus dem Cache und verzögert daher die Leseoperation erheblich.

Den möglichen Vorteilen stehen aber auch Nachteile gegenüber. Werden spekulativ Blöcke in den Cache geladen, die in der Folge nicht verwendet werden, dann verdrängen diese möglicherweise Blöcke, die kurz darauf wieder benötigt werden. Damit würde das *spekulative Laden* die Leistung des Dateisystems verringern.

In der Literatur werden verschiedene Ansätze für das *spekulative Laden* beschrieben:

Sequentielles Vorauslesen stützt sich auf die Beobachtungen von Studien, die besagen, dass Zugriffe auf Dateien größtenteils sequentiell sind. Diese Art von *spekulativem Laden* wird von den meisten aktuellen Dateisystemen unterstützt, darüber hinaus wird sie auch von einigen Betriebssystemen direkt unterstützt.

Lernende Verfahren verwenden Aufzeichnungen vergangener Zugriffe auf bestimmte Dateien und versuchen daraus zukünftige Zugriffsmuster abzuleiten. [50]

Anwendungsgesteuertes Vorausladen: Die Anwendung kann dem Dateisystem über eine spezielle Schnittstelle Datei-spezifische Hinweise für das *spekulative Laden* geben. [15]

Statische Analyse kann von einem Übersetzer dazu verwendet werden in einem Programm Zugriffsmuster zu erkennen und entsprechende Hinweise in das generierte Programm einzusetzen. Diese Technik automatisiert das *anwendungsgesteuerte Vorausladen*. [57]

Spekulative Ausführung von Programmteilen kann zur Laufzeit des Programms dazu verwendet werden, zukünftige Zugriffe auf das Dateisystem vorauszusagen. Die spekulative Ausführung findet dann statt, wenn das Programm auf das Ergebnis einer Dateisystem-Anfrage warten muss. [18]

Mit Ausnahme des *anwendungsgesteuerten Vorausladens* sind die Verfahren automatisch und erfordern keinen Eingriff der Programmierers. Das *sequentielle Vorauslesen* und die *lernenden Verfahren* können auch nachträglich auf bereits übersetzte Programme angewandt werden, während die übrigen Verfahren einen speziellen Übersetzer benötigen.

Nur das sequentielle Vorauslesen ist in aktuellen Dateisystemen (auch aufgrund der Unterstützung von Seiten der Betriebssysteme) zu finden, die restlichen Verfahren wurden bislang nur für Prototypen implementiert.

2.1.7 Wiederherstellung nach Fehlern

Dateisysteme müssen in der Lage sein nach einem Systemabsturz einen konsistenten Zustand erreichen zu können und normal weiter zu arbeiten. Dies ist deshalb schwierig, weil Veränderungen an einem Dateisystem in der Regel mehrere Blöcke betreffen, die bei einem Absturz zwischen zusammen hängenden Schreiboperationen in einen inkonsistenten Zustand gelangen können. Wird beispielsweise eine Datei beschrieben, so müssen einerseits die Daten selber auf das Medium geschrieben werden und andererseits die Verwaltungsinformationen aktualisiert werden, die mit großer Wahrscheinlichkeit in unterschiedlichen Blöcken gespeichert werden.

Das Problem wird einerseits durch das bereits in den vorhergegangen Abschnitten behandelte *Caching* und andererseits durch die *Umordnung der Schreibebefehle* zur Verminderung von Kopfbewegungen bei Festplatten verschärft. Beide Techniken dienen der Latenzverringerung bei Zugriffen auf das Medium. Sowohl das *Caching*, wie auch die eventuelle *Umordnung der Schreibbefehle* können innerhalb des Dateisystems, in dem Blockgerätetreiber und in der Hardware des Gerätes selbst vorgenommen werden, wobei die oberen Schichten auf den Optimierungsgrad der unteren Schichten Einfluss nehmen können. Doch auch dann wenn das Dateisystem die absolute Reihenfolge der Schreiboperationen festlegt, kann es, wie im vorangegangen Abschnitt beschrieben, bei voneinander abhängigen Schreiboperationen zu Inkonsistenzen kommen.

Traditionelle Dateisysteme wie beispielsweise EXT2 [16] überprüfen bei der Montage des Dateisystems ob dieses zuvor ordnungsgemäß heruntergefahren wurde. Ist dies nicht der Fall wird die Konsistenz des gesamten Dateisystems überprüft indem alle Blöcke durchlaufen, untersucht und gegebenfalls korrigiert werden. Der Zeitaufwand ist abhängig von der Größe des verwendeten Mediums und kann bei aktuellen Festplatten 10 bis 60 Minuten in Anspruch nehmen. Während dieser Zeit ist das Dateisystem und damit in der Regel auch der Rechner nicht verfügbar.

Um diesem Problem zu begegnen wurden *log-basierte Dateisysteme* (engl. *log-structured file systems*, *LFS* [68]) entwickelt. Die Grundidee besteht darin,

dass bestehende Blöcke nicht überschieben, sondern in einer neuen Version an das Ende einer zirkulären Liste geschrieben werden, welche von den Blöcken des Mediums gebildet wird. Diese Liste wird *Log* genannt. In regelmäßigen Abständen schreibt das Dateisystem einen *Kontrollpunkt* (engl. *Checkpoint*) an das Ende des Logs um zu markieren, dass ein konsistenter Zustand erreicht wurde. Stürzt zu einem bestimmten Zeitpunkt der Rechner ab, kann das Dateisystem trotz einer eventuell unterbrochenen Änderung an dem Dateisystem zu dem letzten Kontrollpunkt zurückkehren und das Dateisystem damit von dem letzten bekannten konsistenten Zustand aus weiter führen.

Ein *LFS* erfordert eine sehr komplexe *Freispeichersammlung* (engl. *garbage collection*), weil nicht nur die Verwaltungsinformationen, sondern auch die Daten in dem Log abgelegt werden. Daher verwenden aktuelle Dateisysteme eine Variante dieses Ansatzes, bei dem nur die Metadaten in einem so genannten *Journal* abgelegt werden. Durch den Einsatz von *Journalen* kann es zwar im Falle eines Systemabsturzes zu Inkonsistenzen bei den Dateiinhalten kommen, die Metadaten lassen sich jedoch in kurzer Zeit in einen konsistenten Zustand bringen, wodurch das Dateisystem schnell wieder einsatzbereit ist. Beispiele für solche Dateisysteme sind *JFS* [5], *ReiserFS* [55] und *XFS* [24].

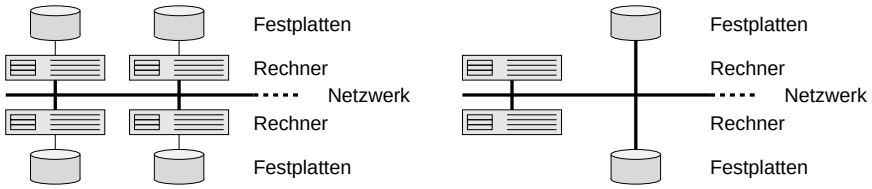
2.2 Verteilte Dateisysteme

Unter *verteilten Dateisystemen* versteht man Dateisysteme, die eine Zusammenarbeit von mehreren eigenständigen Komponenten über ein Netzwerk erfordern. Ziel eines *verteilten Dateisystems* ist es in der Regel, dass von mehreren Rechnern auf das Dateisystem zugegriffen werden kann und es dem Benutzer wie ein lokales Dateisystem erscheint.

Verteilte Dateisysteme weisen große Unterschiede auf, die auf die verwendeten Komponenten und die Entwurfsziele zurück zu führen sind. Die folgenden Kriterien erlauben eine grobe Charakterisierung:

Netzwerkverbindung: Die Netzwerkverbindung spielt eine wichtige Rolle für die Aufgabe, die das Dateisystem erfüllt. Handelt es sich um ein Hochgeschwindigkeitsnetzwerk mit hohem Durchsatz und geringer Latenz, wie sie beispielsweise in Clustern anzutreffen sind, kann das Dateisystem von wissenschaftlichen Anwendungen verwendet werden, die große Datenmengen verarbeiten und üblicherweise parallel von mehreren Rechnern auf das Dateisystem zugreifen. Der hohe Durchsatz der Netzwerkverbindung kann von dem Dateisystem an die Anwendung weitergegeben werden und die geringe Latenz ermöglicht eine enge Zusammenarbeit der einzelnen Komponenten des Dateisystems.

Auf der anderen Seite können mit *Grid-Dateisysteme* mehrere räumlich entfernte Rechenzentren verbunden werden, deren Ressourcen ein gemeinsames Dateisystem bilden. In diesem Fall gibt es große Variationen in der Verbindungsqualität: Während lokale Rechner die im letzten Abschnitt beschriebenen Hochgeschwindigkeitsnetzwerke nutzen können, weisen die Verbindung zu entfernten Komponenten des Dateisystems einen geringeren Durchsatz und eine höhere Latenz auf. Auch ist die Sicherheit der Daten bei einer verteilten Organisation schwerer zu garantieren als bei einem



(a) CAS (b) NAS

Abbildung 2.2: CAS vs. NAS

in sich geschlossenen System. Diesen heterogenen Bedingungen müssen Grid-Dateisysteme Rechnung tragen.

Mobile Rechner, *PDA*s (*Personal Digital Assistant*) und *Mobiltelefone* bringen noch einen weiteren Typ der Netzwerkverbindung ins Spiel. Diese Geräte besitzen keinen permanenten Netzwerkzugriff und können daher nur zeitweise auf die entfernten Komponenten eines *verteilten Dateisystems* zugreifen. Dennoch besteht die Notwendigkeit auch bei unterbrochener Verbindung das Dateisystem weiter benutzen zu können.

Anschluss der Festplatten: Die Festplatten sind entweder an einen Rechner angeschlossen (engl. CAS, *computer-attached storage*, Abbildung 2.2(a)) oder direkt an das Netzwerk (engl. NAS, *network-attached storage*, Abbildung 2.2(b)). Im Fall von CAS hat nur ein einzelner Rechner direkten Zugriff auf die Festplatte und fungiert für die übrigen Rechner als Dienstgeber, wenn diese die Festplatte nutzen wollen. Die ist üblicherweise bei *verteilten Dateisystem* der Fall. Wenn ein *verteiltes Dateisystem* NAS verwendet, wird es auch *gemeinsames Dateisystem* (engl. *shared file system* [60]) genannt.

Paralleler Zugriff: Eine weitere Sonderform der *verteilten Dateisysteme* bilden die *parallelen Dateisysteme*. Diese Dateisysteme sind auf parallele wissenschaftliche Anwendungen spezialisiert und bieten mehreren Rechnern gleichzeitig Zugriff auf die Dateien oder sogar auf verschiedene Teile der selben Datei. Dies wird dadurch erreicht, dass die Dateien über mehrere Festplatten verteilt werden, die von unterschiedlichen Dienstgebern verwaltet werden (im Fall von CAS).

Die folgenden Abschnitte betrachten verschiedene Aspekte der unterschiedlichen Typen von *verteilten Dateisystemen* und geben einen Überblick über den Stand der Technik.

2.2.1 Namensraum

Üblicherweise verwalten *verteilte Dateisysteme*, ähnlich wie lokale Dateisysteme, die Dateien in einem Baum. Auf eine Datei kann über einen Pfad zugegriffen werden, wobei die Datei im Unterschied zu lokalen Dateisystemen über mehrere entfernte Rechner verteilt sein kann.

Ein *verteiltes Dateisystem* speichert eine Datei *ortstransparent* (engl. *location transparent*), wenn der Benutzer anhand des Pfades nicht erkennen kann, ob

eine Datei lokal oder entfernt gespeichert ist. Das Dateisystem *NFS* [70] bietet beispielsweise *Ortstransparenz*.

Der Pfad einer Datei ist *ortsunabhängig* (engl. *location independence*), wenn die Datei auf allen Rechnern, die das *verteilte Dateisystem* verwenden, unter dem gleichen Pfad zugreifbar ist. Aus *Ortsunabhängigkeit* folgt daher *Ortstransparenz*, aber nicht umgekehrt. *AFS* [14, 41] ist ein Beispiel für ein Dateisystem das *Ortsunabhängigkeit* bietet.

2.2.2 Architekturen und Skalierbarkeit

NFS [70] gilt als eines der ersten *verteilten Dateisysteme*. Mit seiner *Dienstnehmer-Dienstgeber-Architektur* (engl. *client-server architecture*) war es beispielhaft für die erste Generation der *verteilten Dateisysteme*. Intern verwendete NFS die Technik der entfernten Methodenaufrufe (*RPC*, engl. *Remote Procedure Call* [58]), wobei der Dienstgeber für die Verwaltung des Dateisystems verantwortlich ist. Im einzelnen sind dies folgende Aufgaben:

- Beantworten von Anfragen der Klienten
- Verwalten der Metadaten
- Verwalten und Speichern der Daten auf der lokalen Festplatte
- Eventuell: Cachen von Daten und Metadaten, um die Bearbeitung der Anfragen von Klienten zu beschleunigen.

Ein NFS-Dienstgeber exportiert einen Teil(-baum) eines lokalen Dateisystems an entfernte Klienten, die diesen in ihren lokalen Verzeichnisbaum einfügen. Für den Benutzer auf einem Klienten ist nicht unmittelbar erkennbar, welche Teile des Verzeichnisbaumes lokal und welche entfernt abgelegt werden. Genau genommen stellt NFS kein eigenständiges *verteiltes Dateisystem* dar, weil auf Seite des Dienstgebers die Verwaltung der Daten und Metadaten von einem lokalen Dateisystem übernommen wird; NFS ermöglicht somit den entfernten Zugriff auf ein lokales Dateisystem.

Demgegenüber besteht *AFS* [41] aus einer Menge von eigenständigen Dienstgebern (*Vice*), die jeweils einen unterschiedlichen Teil des Dateisystems verwalten. Auf Seite der Klienten werden die verschiedenen Dienstgeber in einem gemeinsamen Namensraum zusammengeführt auf den von allen Klienten einheitlich zugegriffen werden kann.

Obwohl ein *verteiltes Dateisystem* aus mehreren Komponenten besteht, die über ein Netzwerk verwaltet werden, bedeutet dies nicht, dass es dabei die potentielle Parallelität dieser Komponenten ausnutzt. Bei der zuvor betrachteten Dienstnehmer-Dienstgeber-Architektur, die bei *NFS* und *AFS* anzutreffen ist, werden die Daten und Metadaten des gesamten Dateisystems (oder im Fall von *AFS*: eines Teils des Dateisystems) von einem einzelnen Dienstgeber verwaltet. Auf dem Dienstgeber werden die eingehenden Anfragen von Klienten serialisiert und nacheinander bearbeitet. Dadurch kann der Dienstgeber bei wachsender Zahl an Klienten zu einem Flaschenhals werden, der die Leistungsfähigkeit des Dateisystems beschränkt.

Eine höhere Skalierbarkeit des Datendurchsatzes kann durch *Verteilung* oder *Replikation* der Daten auf mehrere Komponenten erreicht werden. Replikation

(bzw. *Cachen*) ist vor allem dann vorteilhaft, wenn die Daten vor allem gelesen und selten geändert werden, wie beispielsweise in dem Internet-Dateisystem *PAST* [69]. Der Nachteil von Replikation sind die erhöhten Kommunikations- und Verwaltungskosten, die bei der Konsistenzerhaltung zwischen den Replikaten anfallen. Im folgenden Abschnitt 2.2.3 wird das Cachen in *verteilten Dateisystemen* ausführlich behandelt.

Höherer Datendurchsatz kann auch durch Verteilung der Daten auf mehrere Komponenten erreicht werden. Parallele Dateisystem, wie beispielsweise *PVFS* [17] und *Clusterfile* [47] trennen dazu zunächst die Datenverarbeitung von der Metadatenverarbeitung. Weiterhin werden die Daten, ähnlich wie bei der *RAID*-Technik [19] blockweise auf verschiedene Dienstgeber verteilt. Auf diese Weise können die Klienten durch parallele Anfragen an mehrere Dienstgeber einen höheren Durchsatz erreichen, als es mit einem einzelnen Dienstgeber möglich wäre. Die parallelen Dateisystem werden in Abschnitt 2.2.5 genauer betrachtet.

Ein letzter Engpass bleibt die Metadatenverwaltung, die zumeist bei einem einzigen zentralen Dienstgeber verbleibt. Es gibt einige konzeptionelle Ansätze, auch wenn die praktische Umsetzung in reale Dateisysteme bislang noch nicht vorgenommen wurde. In Abschnitt 2.3 werden die verwandten Arbeiten in diesem Bereich untersucht. Am weitesten geht die Verteilung bei *xFS* [2], das vollständig ohne zentralen Dienstgeber entworfen wurde. Dadurch wird verhindert, dass eine einzelne Komponente die Leistungsfähigkeit des Systems beschränkt. Sowohl Daten, wie auch Metadaten sind über alle Komponenten verteilt und teilweise repliziert. Aufgrund der Komplexität des Entwurfs ist eine vollständige Implementierung von *xFS* jedoch bis heute nicht gelungen.

2.2.3 Cachen

Ziel des Cachens in *verteilten Dateisystemen* ist es einerseits, wie bereits bei den lokalen Dateisystemen (siehe Abschnitt 2.1.5), Festplattezugriffe durch Zwischenspeichern der Daten im Hauptspeicher zu verhindern. Diese Technik kommt zumeist auf den Dienstgebern zum Einsatz, insbesondere wenn diese (wie *NFS*) auf einem lokalen Dateisystem aufsetzen. Andererseits bietet sich bei *verteilten Dateisystemen* als weitere Cache-Ebene das Cachen auf den Dienstnehmern an, um auf diese Weise Netzwerkkommunikation und die damit verbundene Verzögerung möglichst zu vermeiden.

Schon *NFS*, als eines der ersten *verteilten Dateisysteme*, nutzt das Zwischenspeichern sowohl auf dem Dienstgeber, als auch auf den Dienstnehmern. Durch die zwei Ebenen steht jedem Klienten ein größerer Cache zur Verfügung, der sich einerseits aus dem gemeinsamen Cache auf dem Dienstgeber und andererseits aus dem lokal genutzten Cache auf dem Klienten selber zusammen setzt.

Um die Cache-Größe und damit die Trefferquote im Cache weiter zu erhöhen, wurde erstmals 1994 von den Entwicklern von *xFS* (Dahlin et al. [26]) vorgeschlagen, die Klienten zusammen arbeiten zu lassen und so die einzelnen Caches der Klienten zu einem gemeinsamen Cache zusammen zu fügen. Diese Technik wird *kooperatives Cachen* (engl. *cooperative caching*) genannt und ist dadurch motiviert, dass bei Hochgeschwindigkeits-Netzwerken der Zugriff auf den Speicher eines anderen Klienten um ca. zwei Größenordnungen schneller ist, als der Zugriff auf die Festplatte des Dienstgebers.

Nach Dahlin et al. setzt sich ein Algorithmus zum *kooperativen Cachen* aus drei Teilen zusammen:

Eine lokale und globale Suchstrategie : Beim *kooperativen Cachen* kann jeder Klient einen gesuchten Block nicht nur im eigenen Cache, sondern auch in den Caches der anderen Klienten suchen. Ist ein Block in keiner Klienten-Cache zu finden, wird eine Anfrage an den Dienstgeber gesendet. Die Suchstrategie beschreibt auf welche Weise ein Klient in den Caches nach einem angefragten Block sucht.

Eine lokale und globale Ersetzungsstrategie : Wenn ein Block in einen Cache geladen wird, der bereits voll ist, muss der Algorithmus mittels der Ersetzungsstrategie entscheiden, welcher Block aus dem Cache verdrängt wird. Darüber hinaus muss entschieden werden, ob der Block verworfen, zurückgeschrieben oder in den Cache eines anderen Klienten verschoben wird.

Ein Konsistenzprotokoll : Wenn ein Block in den Caches mehrerer Klienten zu finden ist, muss ein Konsistenzprotokoll im Fall von Änderungen die Konsistenz der Kopien sicherstellen. (Die von Dahlin et al. beschriebenen Algorithmen betrachten nur die Leseleistung der Algorithmen und definieren kein Konsistenzprotokoll.)

In dem Papier von Dahlin et al. werden eine Reihe von Algorithmen beschrieben und anhand von Simulationen vermessen. Am besten schneidet der *n-chance forwarding*-Algorithmus ab, der aus diesem Grund in dem Dateisystem *xFS* [2] implementiert wurde:

Suchstrategie : Zunächst sucht der Klient in seinem lokalen Cache. Bei Misserfolg stellt er eine Anfrage an den Dienstgeber, der den Block sendet, falls er diesen im Cache hält. Andernfalls sendet er die Anfrage an einen Klienten weiter, der laut seiner internen Liste den Block im Cache speichert; auf die Festplatte wird nur dann zugegriffen, falls kein solcher Klient gefunden wurde.

Ersetzungsstrategie : Der Anteil des Caches, den ein Klient dem gemeinsamen Cache zur Verfügung stellt, ändert sich dynamisch in Abhängigkeit von der Auslastung des Klienten: Je weniger aktiv ein Klient ist, desto mehr Speicher stellt dieser zur Verfügung.

Blöcke können auf mehreren Klienten repliziert im Cache gespeichert werden; der Dienstgeber führt Buch über alle Blöcke. Blöcke werden aus den Cache verdrängt, sobald sie mittels einer lokalen *LRU*-Strategie (engl. *least recently used*, dient dem Auffinden des am längsten nicht mehr referenzierten Blocks) ausgewählt wurden. Eine Sonderrolle spielen die nur einmal im globalen Cache gespeicherten Blöcke, die so genannten *Singlets*: Sie werden im Falle ihrer Verdrängung bis zu n Mal an einen weiteren, zufällig ausgewählten Klienten weitergeleitet und anschließend verworfen.

Ein weiterer von Dahlin et al. geschriebener Algorithmus ist das *Hash-distributed caching*. Bei diesem Verfahren wird jedem Block anhand des Hashwertes seiner Adresse ein Klienten-Cache fest zugeordnet, der den Block speichert. Falls also ein Klient den Block nicht in seinem lokalen Cache findet, kann

er sich direkt mit dem zugeordneten Klienten in Verbindung setzen. Speichert auch dieser den Block nicht, wird die Anfrage an den Dienstgeber gestellt. Dieser Algorithmus erfordert keine globale Ersetzungsstrategie, weil jeder Block genau einem Klienten-Cache zugeordnet ist und nicht auf andere Klienten verdrängt werden kann. Die globalen Caches von *PAFS* [25] und *Clusterfile* [45] verwenden diesen Algorithmus.

Neuere Arbeiten verfeinern die Algorithmen. Der *Home-based Serverless Cooperative Caching*-Algorithmus (*HSCC* [62]) stellt beispielsweise eine Kombination aus den *hash-distributed caching* und dem *n-chance forwarding* dar: Im Unterschied zum *n-chance forwarding*, bei dem der globale Cache von einem zentralen Dienstgeber verwaltet wurde, verteilt *HSCC* die Verwaltung. Dazu weist es jedem Block einen *Heimat-Klienten* zu, der für die Verwaltung der ihm zugewiesenen Blöcke verantwortlich ist. *HSCC* wird in *CARDs* (*Cluster-Aware Remote Disks* [61]), einem virtuellen Blockgerätetreiber eingesetzt, der die transparente Verwendung von entfernten Festplatten ermöglicht.

2.2.4 Schreib-Semantik

Durch den Einsatz von Caches, die eine Replikation der Daten erlauben, kann es in *verteilten Dateisystemen* zu einem Konsistenzproblem kommen, sobald mehrere Prozesse auf eine Datei oder ein Verzeichnis zugreifen und mindestens einer der Prozesse die Daten oder den Verzeichnisinhalt verändert. Für diesen Fall muss festgelegt werden, wie mit den eventuell entstehenden Inkonsistenzen umgegangen wird. Die *Schreib-Semantik* eines Dateisystems definiert, zu welchem Zeitpunkt lesende Prozesse eine Veränderung der Daten (oder Metadaten) sehen, wenn die Daten (oder Metadaten) gleichzeitig geschrieben werden.

Unix-Semantik

Die bekannteste Semantik ist die *UNIX-Schreib-Semantik*. Sie stammt ursprünglich von den Einzelrechnern und sieht vor, dass nach dem Schreiben einer Datei durch einen Prozess alle nachfolgenden Leseoperationen durch beliebige Prozesse die geänderten Daten liefern. Die Semantik ist offenbar auf Einzelrechnern einfach implementierbar, weil sie über einen einzelnen lokalen Cache verfügen, über den alle Anfragen abgewickelt werden. Bei *verteilten Dateisystemen* können jedoch mehrere Caches auf verschiedenen Rechnern den selben Block speichern. Wenn nun ein Klient eine Änderung an dem Block vornimmt, müssen laut UNIX-Semantik alle nachfolgenden Zugriffe auf den Block von allen Klienten die neue Version lesen. Daher müssen replizierte Blöcke entweder aktualisiert oder invalidiert werden, bevor die Schreiboperation beendet ist.

Bei dem Dateisystem *xFS* [2] kommt beispielsweise ein Invalidierungsprotokoll zum Einsatz, das vor dem Schreiben eines Blockes die Freigabe durch einen Block-Manager abwartet. Der Block-Manager stellt sicher, dass alle Replikate des Blockes in anderen Caches als ungültig markiert werden, bevor er das Schreiben des Blockes erlaubt. Das Protokoll kann aufgrund der zusätzlichen Kommunikation und des Verwaltungsaufwands bei den anderen Caches zu einer großen Verzögerung der Schreiboperation führen.

Auf der anderen Seite lässt sich das Konsistenzproblem auch vermeiden, indem keine Replikation von Blöcken erlaubt wird, wie es beispielsweise beim *Hash-distributed caching* (siehe Abschnitt refrel:cache) vorgesehen ist. Nachteil

dieser Einschränkung ist die geringere Wahrscheinlichkeit von lokalen Treffern im Cache und somit eine schlechtere Leistung bei überwiegend lesenden Zugriffen auf Dateien.

Um den großen Aufwand, der bei Einhaltung der *UNIX-Semantik* anfällt, zu reduzieren wurden schwächere Schreib-Semantiken vorgeschlagen:

Sitzungs-Semantik

Als *Sitzung* wird der Zeitraum zwischen Öffnen und Schließen der Dateien bezeichnet. Die *Sitzungs-Semantik* sieht vor, dass alle Änderungen die ein Prozess an einer Datei vornimmt erst dann für andere Prozesse sichtbar sind, wenn der schreibende Prozess die Datei schließt.

Die Semantik verringert die für die Konsistenzerhaltung notwendige Zeit, weil das Konsistenzprotokoll nicht nach jeder einzelnen Schreiboperation, sondern erst beim Schließen der Datei durchgeführt werden muss. Das Dateisystem *AFS* [41] verwendet die *Sitzungs-Semantik*.

Transaktions-Semantik

Die *Transaktions-Semantik* stammt ursprünglich aus dem Bereich der Datenbanken. Eine Transaktion beinhaltet mehrere Operationen auf dem Dateisystem, die atomar bearbeitet werden. Das bedeutet, dass die Veränderungen an der Datei erst mit dem Beenden der Transaktion für andere Prozesse sichtbar werden. Werden mehrere Transaktion auf unterschiedlichen Klienten zur gleichen Zeit durchgeführt, ist undefiniert, in welcher Reihenfolge sie zur Ausführung kommen. Dateioperation, die nicht Teil einer Transaktion sind, werden wie ein Transaktion mit nur einer Operation behandelt.

Diese Semantik hat, wie die *Sitzungs-Semantik*, den Vorteil, dass die Konsistenzprotokolle nicht nach jeder einzelnen Dateioperation ausgeführt werden müssen, sondern nur nach Beendigung einer Transaktion.

NFS-Semantik

Das Dateisystem *NFS* [70] verwendet ein festes Zeitschema für die Konsistenzherstellung. Die Daten in den Caches verfallen nach kurzer Zeit, so dass sie neu vom Dienstgeber geladen werden müssen. Beim Schreiben einer Klienten werden die Daten durch den lokalen Cache hindurch direkt auf den Dienstgeber geschrieben.

Die Aktualisierungsabstände betragen üblicherweise 3 Sekunden für Daten und 30 Sekunden für Metadaten. *NFS* garantiert, dass alle Klienten nach Ablauf dieser Zeit die Änderungen empfangen haben.

2.2.5 Parallele Dateisysteme

Untersuchungen von Dateizugriffsmustern haben gezeigt, dass Ein-Prozessor-Anwendungen selten gleichzeitig auf die selbe Datei zugreifen [4]. Die Architektur der klassischen *verteilten Dateisysteme* orientiert sich an dieser Beobachtung und serialisiert alle Anfragen auf einem einzelnen Dienstgeber.

Wie bereits in Abschnitt 2.2.2 beschrieben, unterscheiden sich *parallele Dateisysteme* von den übrigen *verteilten Dateisystemen* dadurch, dass sie nicht nur die Verwaltung der Daten und Metadaten trennen, sondern auch die Daten

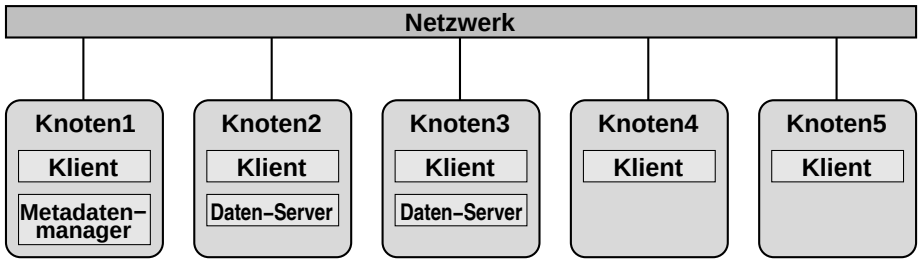


Abbildung 2.3: Verschiedene Knotenrollen in parallelen Dateisystemen

auf mehrere Knoten verteilen, um auf diese Weise einen höheren Durchsatz zu ermöglichen. Dies ist motiviert durch Untersuchungen, die festgestellt haben, dass bei parallelen Anwendungen häufig von mehreren Knoten gleichzeitig auf die selbe Datei zugegriffen wird [59, 76]. Die Teile der Datei auf welche die unterschiedlichen Knoten zugreifen, überscheiden sich normalerweise nicht.

Verteilung der Daten

Damit ein echter paralleler Zugriff möglich ist, müssen die Teile der Dateien, auf die parallel zugegriffen wird, auf verschiedenen Festplatten gespeichert sein und diese Festplatten müssen sich in unterschiedlichen Rechnern befinden. In *parallelen Dateisystemen* gibt es verschiedene Rollen, welche die Knoten eines Supercomputers einnehmen können: *Metadatenmanager*, *Daten-Server* und *Klient*, die auch *Rechenknoten* genannt werden. Ein Knoten kann auch mehrere Rollen übernehmen, wie Abbildung 2.3 illustriert. Die *parallelen Dateisysteme* für Cluster, wie beispielsweise *PVFS* [17], *Clusterfile* [44] und *GPFS* [72], verwenden alle diesen Entwurf.

Um die mögliche Parallelität ausnutzen zu können, müssen die Zugriffsmuster der parallelen Anwendungen möglichst gut zu der Aufteilung der Dateien auf die Festplatten passen. Bei einer ungünstigen Verteilung der Daten kann es zu einer ungewollten Serialisierung der Anfragen kommen, wie beispielsweise von Isailă et al. [47] beschrieben wird. Es ist daher wünschenswert, dass ein *paralleles Dateisystem* eine möglichst flexible Austeilung der Dateien auf die Daten-Server erlaubt.

Bei *GPFS* und *PVFS* werden die Dateien in Blöcke gleicher Größe aufgeteilt, die reihum auf alle verfügbaren Daten-Server verteilt werden. Während diese Verteilung dem Dateisystem eine besonders einfache Verwaltung der Datenblöcke erlaubt, ist sie jedoch für den Benutzer des Dateisystems transparent und kann auch nicht auf die Anwendung abgestimmt werden.

Andere Dateisysteme offenbaren die physikalische Verteilung einer Datei, beispielsweise durch die Darstellung als *Teildateien* (engl. *subfiles*), die weiterhin linear adressiert werden können (zum Beispiel: *DPFS* [75]). Der Benutzer kann Einfluss darauf nehmen, auf welchem Daten-Server eine Teildatei gespeichert wird, um diese auf das Zugriffsmuster einer Anwendung abzustimmen.

Das Dateisystem *Vesta* [22] orientiert sich bei der Aufteilung der Dateien in *Teildateien* an Matrizen. Matrizen werden in parallelen Anwendungen sehr häufig verwendet und in der Regel bearbeitet ein Knoten einen Teil der Gesamtmatrix, so dass es wünschenswert ist, die verschiedenen Teile der Matrix

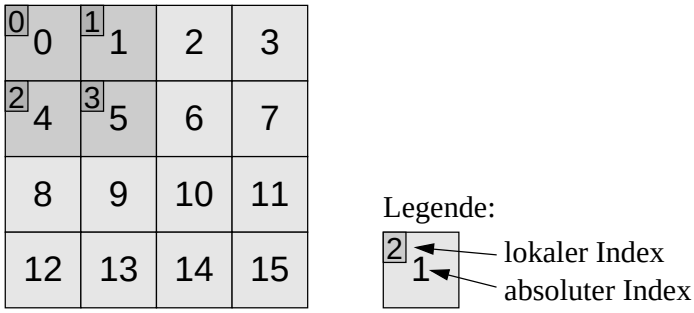


Abbildung 2.4: Beispiel: Definition einer Sicht auf eine Datei, die eine Matrix speichert

auf unterschiedlichen Daten-Servern zu speichern [59]. *Vesta* erlaubt allerdings nur eine Aufteilung in zwei-dimensionale rechteckige Felder; höhere Dimensionen werden nicht unterstützt.

Auch *Clusterfile* orientiert sich an Matrizen, ermöglicht jedoch durch eine rekursive Verteilungsfunktion auch mehrdimensionale Verteilungsmuster. *Clusterfile* verwendet zur Beschreibung der Verteilung einen Formalismus namens *PITFALLS*, der ursprünglich aus dem Übersetzerbau stammt und für die Umverteilung von Feldern in Rechnern mit verteiltem Speicher eingesetzt wurde [67]. Durch die rekursive Erweiterung der *PITFALLS* in *Clusterfile* sind beliebige Verteilungen der Dateien auf die Daten-Server möglich.

Sichten

Einige *parallele Dateisysteme* ermöglichen den Klienten nach dem Öffnen einer Datei das Definieren von *Sichten* (engl. *views*). Eine Sicht ist eine transparente Abbildung des lokalen (auch: logischen) Dateiindexes auf den absoluten Index der Datei. Nach der Definition einer Sicht kann der Klient mehrere unzusammenhängende Teile der Datei mit einer Anweisung lesen oder beschreiben, weil sie durch die Abbildung auf den lokalen Dateiindex linear zusammenhängend erscheinen.

Eine Sicht kann beispielsweise beim Lesen aus einer Datei, die eine große Matrix speichert, dazu verwendet werden, dass der Klient nur auf den für ihn relevanten Teil als zusammenhängenden Datenblock zugreifen kann. Abbildung 2.4 zeigt dies beispielhaft an einer 4x4-Matrix, die zeilenweise gespeichert wird: Der *absolute Index* bezieht sich auf die gesamte Datei, der *lokale Index* bezieht sich auf einen Klienten. Nach Definition der *Sicht* (dunkelgrau unterlegt), sind für den Klienten nur noch die Matrixeinträge 0, 1, 4 und 5 unter den lokalen Indizes 0, 1, 2 und 3 zugreifbar; der Rest der Datei ist für diesen Klienten nicht mehr verfügbar. Mit Hilfe der Sicht ist es dem Klienten möglich, alle für ihn relevanten Informationen mit einer Operation einzulesen.

Sichten werden von einigen *parallelen Dateisystemen* und auch von Bibliotheken, wie MPI-IO [34] angeboten. Die einzelnen Systeme unterscheiden sich in der Ausdruckskraft der Formalismen, die sie für die Definition der Sichten verwenden: Das Dateisystem Vesta [22] erlaubt die Definition von zwei-dimensionalen Sichten und hat damit die gleichen Einschränkungen, die auch bei der Festlegung

der physikalischen Verteilung der Daten auf die Daten-Server zutreffen. Benutzer des Dateisystems *PVFS* [17] können bei Verwendung der Benutzerebenen-Bibliothek mehrdimensionale Sichten definieren.

Sowohl in MPI-IO, wie auch in *Clusterfile* [47] werden Sichten rekursiv definiert, wodurch die größte Flexibilität erreicht wird. Bei MPI-IO werden geschachtelte MPI-Datentypen verwendet, um beliebige Sichten auf Dateien zu definieren, die auch auf herkömmlichen Dateisystemen gespeichert sein können. Um die erweiterten Fähigkeiten eines *parallelen Dateisystems* innerhalb von MPI-IO zu verwenden, ist eine spezielle Implementierung notwendig, welche die Definition der Sichten mittels der MPI-Datentypen auf den Formalismus des Dateisystems abbildet. Eine solche Implementierung existiert für *Clusterfile* (das intern *PITFALLS* zur Definition von Sichten verwendet). Es konnte nachgewiesen werden, dass die Mächtigkeit der Sichten-Definition in *Clusterfile* mindestens so groß wie die von MPI-IO ist [81].

Neben den Vorteilen von Sichten für den Benutzer gibt es auch Vorteile für Dateisysteme. Sichten werden nach dem Öffnen einer Datei und vor dem ersten Zugriff auf diese gesetzt. Das ermöglichst schon zu diesem frühen Zeitpunkt die Berechnung der Abbildung zwischen der Sicht und der physikalischen Verteilung der Daten auf den Daten-Servern. Weiterhin können Sichten von dem Dateisystem als Hinweis auf das Zugriffsmuster einer Anwendung interpretiert werden. Diese Hinweise können zur Verbesserung des Cachens und Vorausladens verwendet werden.

2.2.6 Gemeinsame Dateisysteme

Ursprünglich waren die Festplatten bei *verteilten Dateisystemen* direkt an die Rechner angeschlossen. Damit ein entfernter Rechner auf eine Festplatte zugreifen kann, muss er zunächst den Rechner kontaktieren an den diese angeschlossen ist. Falls dieser Dienstgeber nicht verfügbar ist, aufgrund eines Absturzes, Überlast oder einem anderen Grund, ist auch die Festplatte nicht verfügbar. Der direkte Anschluss der Festplatten an das Netzwerk (engl. *network-attached storage*, *NAS*) trennt die Rechner von den Festplatten und macht sie allen Klienten direkt über das Netzwerk zugänglich (siehe auch Abbildung 2.2).

Dieser Ansatz hat mehrere Vorteile: [37]

- *NAS* ermöglicht eine einfache Trennung von Daten und Metadaten: ein Metadatenmanager verwaltet die Metadaten, während die Daten auf den Festplatten gespeichert werden. Beim Öffnen einer Datei kontaktiert ein Klient zunächst den Manager und erhält eine *Marke* (engl. *Token*), die den Zugriff auf den Dateiinhalt autorisiert. Nachfolgende Zugriffe des Klienten auf die Datei können den Manager umgehen und direkt zwischen Festplatte und Klient abgewickelt werden.

- Im Allgemeinen kann auf dedizierte Dienstgeber-Rechner verzichtet werden und sie können aufgrund der geringeren Last für andere Zwecke verwendet werden.

- Der Durchsatz kann gesteigert werden, weil die Anfragen der Klienten und die Daten der Antworten nicht im Hauptspeicher des Daten-Servers zwischengespeichert und von dem Betriebssystem verarbeitet werden müssen. Stattdessen richtet der Klient seine Anfrage direkt an die Festplatte.

- Im Fall von *CAS* (engl. *computer.attached storage*) werden die Daten zunächst über den internen Bus des Daten-Servers und anschließend über das Netzwerk zwischen Dienstgeber und Klient versendet. Bei *NAS* entfällt der erste Schritt.

- Wenn ein Rechner ausfällt ist der Zugriff auf die Festplatten nicht in Mitleidenschaft gezogen; sie sind weiterhin für die übrigen Rechner verfügbar.

Die größte Herausforderung bei *NAS* ist der gleichzeitige Zugriff mehrerer Klienten auf den selben Datenblock. Besitzen zwei Klienten Schreibrechte auf einem Block, kann dies zu Inkonsistenzen führen, insbesondere dann, wenn es sich bei den im Block gespeicherten Informationen um Verwaltungsinformationen handelt. Daher muss eine externe Verwaltungsinstanz den schreibenden Zugriff auf Blöcke koordinieren, beispielsweise durch die Ausgabe von *Marken*, die den schreibenden Zugriff auf einen bestimmten Block oder einen Bereich von Blöcken erlauben.

2.2.7 Grid-Dateisysteme

Ein *Grid* ist ein System, das entfernte Ressourcen koordiniert, die nicht einer zentralen Instanz untergeordnet sind und das offene, standardisierte Protokolle und Schnittstellen verwendet um nicht triviale Dienstgüten bereitszustellen (nach Ian Foster [35]). Praktisch bedeutet dies, dass eine Reihe von geographisch entfernten Rechnenzentren, ihre Ressourcen (Speicher und Rechenkapazität) zusammen schließen, um auf diese Weise Aufgaben zu lösen, die sie alleine nicht bewältigen könnten. Der Begriff *Grid* hat seinen Ursprung in dem Vergleich dieser Technologie zum Stromnetz (engl. *power grid*). Demnach soll das *Grid* ebenso einfach Ressourcen zur Verfügung stellen, wie es möglich ist Strom aus einer Steckdose zu beziehen.

Grid-Dateisysteme [63] fassen die Speicher-Ressourcen verschiedener verteilter Einrichtungen zusammen. Speicher-Ressourcen sind in diesem Fall nicht Festplatten, sondern auf einer höheren Ebene zu verstehen, wie beispielsweise Dateien, Dateisysteme oder Datenbanken. Ein Grid-Dateisystem fasst die Ressourcen zu einem gemeinsamen Namensraum zusammen und ermöglicht die gemeinsame Nutzung.

Die besonderen Herausforderungen bei der Implementierung von Grid-Dateisystemen sind größtenteils administrativ: Dezentrale Verwaltung der Benutzer und Benutzerrechte, Sicherheit der einzelnen Rechenzentren, sowie Integration und Entfernen von Ressourcen, ohne die Gesamtintegrität des Systems zu gefährden sind nur einige Beispiele für Aufgaben, die ein Grid-Dateisystem von den zuvor beschriebenen verteilten Dateisystemen unterscheidet.

2.2.8 Verteilte Dateisysteme für mobile Rechner

In *verteilten Dateisystemen* kommunizieren die einzelnen Komponenten über ein gemeinsames Netzwerk miteinander. In der Regel spielt sich die Kommunikation zwischen einem zentralen Dienstgeber und den räumlich verteilten Klienten ab. In den vorangegangen Abschnitten wurde zumeist angenommen, dass das Netzwerk zu jedem Zeitpunkt verfügbar ist; dies trifft bei mobilen Rechnern nicht zu.

Damit der Benutzer eines *verteiltes Dateisystems* auch bei einer Unterbrechung des Netzwerkdienstes seine Arbeit fortsetzen kann, müssen spezialisierte Dateisysteme eingesetzt werden. Bei diesen *mobilen Dateisystemen* wird der gesamte Inhalt des Dateisystems auf die Festplatte des lokalen Rechners vorgeladen. Im Fall einer Netzwerkunterbrechung kann der Benutzer weiterhin auf das Dateisystem zugreifen, das intern die lokale Kopie der Daten verwendet. Während der Unterbrechung kann der Benutzer ohne Einschränkungen das Dateisystem modifizieren. Die Veränderungen werden an den Dienstgeber gesendet, sobald dieser wieder per Netzwerk verfügbar ist.

Falls zwei Benutzer unabhängig voneinander Veränderungen vornehmen, die sich widersprechen, muss dieser Konflikt behoben werden. In einigen Fällen, wie beispielsweise bei Quelltexten kann dies, analog zu Konflikten bei Versionshaltungssystemen, automatisch behoben werden. In den übrigen Fällen ist ein manueller Eingriff notwendig. Das Dateisystem *Coda* [71] implementiert diese Techniken.

2.3 Metadatenverwaltung in verteilten Dateisystemen

Das Thema dieser Arbeit ist die Verteilung der Metadatenverwaltung eines Cluster-Dateisystems auf mehrere Dienstgeber. Wie bereits im Abschnitt 2.2.2 beschrieben wurde, kann ein zentraler Metadatenmanager bei einer hohen Zahl von Anfragen zu einem Engpass werden, weil 50% bis 80% aller Zugriffe auf ein Dateisystem die Metadaten betreffen [65]. Trotz der, im Vergleich zu den Daten, geringen Größe der Metadaten, kann die Verwaltung dieser Informationen die Leistungsfähigkeit des gesamten Dateisystems beschränken.

Bezüglich der Verwaltung von Metadaten gibt es bereits einige Arbeiten. Im Mittelpunkt dieser Arbeiten steht zumeist die Frage nach dem besten Verteilungsmuster für Metadaten, obwohl die grundsätzliche Machbarkeit einer solchen Verteilung bis heute nur anhand von Simulationen gezeigt wurde. In den folgenden Abschnitten werden, ausgehend von einer Auswahl verteilter Dateisysteme, verschiedene Verteilungstechniken vorgestellt.

2.3.1 Statische Verteilung: NFS, AFS und andere

Bei den traditionellen Netzwerkdateisystemen wird meist nicht zwischen Metadaten- und Datenverwaltung unterschieden. Trotzdem ist auch bei diesen Dateisystemen eine Lastverteilung möglich, indem Teile der Dateisystemhierarchie statisch verschiedenen Dienstgebern zugeordnet werden. Diese *statische Verzeichnisverteilung* muss normalerweise manuell durch einen Systemadministrator vorgenommen werden. Für die Klienten ist diese Verteilung besonders einfach zu handhaben, da das Finden des verantwortlichen Dienstgebers einfach ist und Anfragen direkt an den für einen Verzeichnisbaum zuständigen Knoten geschickt werden können. Zu den zahlreichen Vertretern dieser Gruppe von Dateisystem gehören beispielsweise *NFS* [70], *AFS* [41], *Coda* [71] und *Zebra* [40].

Eine solche Verteilung kann durchaus den gewünschten Effekt der Lastverteilung haben. Offensichtlich ist aber der Aufwand relativ groß, denn die Balancierung ist ein manueller Prozess, der nicht automatisiert abläuft und daher nur begrenzt skaliert. Durch die statische Verteilung können sich diese Dateisysteme

auch nicht an schwankende Anforderungsmuster anpassen. Sie werden von einem Administrator auf bestimmte Annahmen hin optimiert, verhalten sich aber in anderen, nicht vorhergesehenen Situationen, suboptiomal.

Weil bei den meisten dieser Dateisystemen gleichzeitig mit den Metadaten auch die Daten verteilt werden, sind diese besonders anfällig für Engpässe, so genannte *Hot Spots*. *Hot Spots* entstehen dadurch, dass bestimmte Dateien besonders „populär" sind, also von vielen Klienten gleichzeitig angefordert werden. In dieser Architektur ist für jede Datei nur ein einziger Dienstgeber verantwortlich, der bei vielen Anfragen auf mehrere seiner Dateien überlastet werden kann, während die Dienstgeber, die für andere Teile der Dateihierarchie zuständig sind, ungenutzt bleiben.

2.3.2 Streuverteilung: Frangipani und Petal

Frangipani [84] ist ein verteiltes Dateisystem, das vor allem auf leichte Erweiterbarkeit und gute Lastverteilung spezialisiert ist. *Frangipani* besteht aus zwei Schichten: Die obere Schicht ist das Dateisystem selbst, während die untere Schicht von *Petal* [51], einem virtuellen Blockgerät, gebildet wird.

Petal besteht aus einer Reihe von verteilten Speicherdienstgebern, die ein gemeinsames, Block-basiertes Speichersystem implementieren. Klienten greifen auf die Blöcke mittels einer RPC-Schnittstelle zu und besitzen nur wenige Informationen über die Verteilung der Daten, die über eine *virtuelle Festplattennummer* und einen *Blockindex* indiziert werden. Insgesamt hat *Petal* einen Adressbereich von 2^{64} Bytes. Die Dienstgeber sorgen für eine gleichmäßige und redundante Verteilung der Blöcke auf die verfügbaren Festplatten und erlauben auch eine nachträgliche Erweiterung des Systems um weitere Dienstgeber und Festplatten. Das Verfahren zur Verteilung der Blöcke auf die Festplatten ist *Chained-Declustering*, das ursprünglich aus dem Bereich der Mehrprozessor-Datenbankverwaltung stammt. Es verteilt die Blöcke reihum auf die Dienstgeber, so dass bei n Dienstgebern, nummeriert von 0 bis $n-1$, ein Block b auf dem primären Dienstgeber $d = b \; modulo \; n$ gespeichert wird. Um Redundanz zu erreichen, wird zusätzlich eine Kopie des Blockes auf eine sekundären Dienstgeber $(b+1) \; modulo \; n$ gespeichert. Beim Zugriff auf einen Block hat immer der primäre Dienstgeber Priorität, der sekundäre Dienstgeber kann in Überlastsituationen jedoch Anfragen übernehmen. Für Schreiboperationen wird die Konsistenz durch eine Sperre auf dem primären Dienstgeber gewährleistet. Fällt dieser aus, so kann der sekundäre Dienstgeber den Auftrag übernehmen.

Das Dateisystem *Frangipani* ist oberhalb von *Petal* implementiert und ähnelt der Implementierung eines lokalen Dateisystems, das Blockgeräte benutzt. Dadurch, dass es sich bei *Petal* um ein virtuelles Blockgerät handelt, welches in Wirklichkeit ein verteilter Dienst ist, erbt *Frangipani* die Redundanz und die balancierte Verteilung der Daten. Obwohl *Petal* Konsistenz auf Blockebene garantiert, so ist jedoch, wie auch bei anderen verteilten Dateisystemen, ein Sperrmechanismus für Strukturen höherer Ebene, wie beispielsweise Dateien notwendig, um die Konsistenz des gesamten Dateisystems zu sichern. Der Sperrmechanismus wird von verteilten Dienstgebern implementiert, die miteinander kooperieren um Skalierbarkeit und Fehlertoleranz zu erreichen.

Die Metadaten werden in *Frangipani* zusammen mit den Dateiinhalten auf einem *Petal*-Gerät gespeichert. Daher werden sie durch die implizite Verteilung der Blöcke automatisch auf mehrere Dienstgeber verteilt. Entscheidend für die

Zuordnung ist allein der Index des Blockes in dem die Metainformationen gespeichert werden. Es handelt es sich also im Kern um eine Verteilung anhand von Streuadressierung. Vorteil ist die einfache Auffindbarkeit der Daten und die geringe Wahrscheinlichkeit von Ballungen (engl. *hot spots*) bei intensiver Metadatenaktivität in einem Verzeichnis. Ein Nachteil ist die geringe Kopplung zwischen Dateisystem und Verteilungsfunktion, die aus der Gleichbehandlung von Daten und Metadaten resultiert. Mit den zusätzlichen Informationen eines Dateisystem wäre eine bessere Verteilung der Metadaten möglich. So könnten beispielsweise der Inhalt eines Verzeichnisses und die Metadateninformationen des Verzeichnisses auf dem gleichen Knoten gespeichert werden. Um das zu erreichen, müsste zwischen Metadaten und Daten unterschieden werden. Weiterhin dürften die Metadaten nicht auf den *Petal* Blockgeräten gespeichert werden, aber dies widerspräche der Architekturkonzeption dieses Dateisystems.

2.3.3 Streuverteilung: Lustre

Lustre [9] ist eines der bekanntesten Cluster-Dateisysteme und gleichzeitig auch eines des jüngsten. Das Projekt begann 1999, die Entwicklung startete ein Jahr später. *Lustre* wurde so entworfen, dass es den Anforderungen der größten Cluster genügen kann und zeigt in großen Installationen, wie beispielsweise einem 4000 Knoten Cluster der *Lawrence Livermore National Laboratories* „angeblich“ gute Skalierbarkeit. Es gibt allerdings keine Veröffentlichung, die dies anhand von Messungen nachvollziehbar beschreibt. Auch sind viele der in verschiedenen Dokumenten (wie beispielsweise dem *White Paper* [8]) beschriebenen Eigenschaften noch weit von der Fertigstellung entfernt. Dies ist allerdings nicht offensichtlich und erschließt sich erst nach intensiver Nachforschung. Laut Informationen auf der Lustre-Web-Seite wird *Lustre* auf sechs der schnellsten elf Linux-Clustern verwendet.

Lustre verwendet die übliche Architektur mit den Rollen des Metadatenmanagers (*MetaData* Server, *MDS*), Daten-Servers (*Object Storage Target*, *OST*) und Klienten, sieht aber die Möglichkeit von mehreren *MDS*en vor. In der aktuellen Version von *Lustre* ist eine echte Verteilung von Metadaten noch nicht vorgesehen, es ist jedoch der gleichzeitige Betrieb von zwei *MDS*e möglich. Es findet dabei zwischen diesen Dienstgebern kein Lastausgleich statt; der zweite *MDS* spiegelt den ersten und ersetzt ihn, falls er ausfällt. Es wird allerdings versichert, dass der Metadatenmanager auch bei Systemen mit mehr als tausend Knoten bisher keinen Engpass darstellt.

Dennoch ist die Verteilung der Metadaten geplant. Es ist aber unklar, wann dieses Problem in Angriff genommen wird. Die Metadaten sollen anhand der Hashsummen ihrer Namen auf die verfügbaren *MDS* verteilt werden. Dieser Ansatz wird dem hierarchischen Verteilen vorgezogen, da er auch bei großen Verzeichnissen mit tausenden von Dateien, auf die gleichzeitig zugegriffen wird, skaliert.

2.3.4 Streuverteilung: xFS

Dem Dateisystem *xFS* [2] liegt die Idee zugrunde, dass alle Dienste über die beteiligten Rechner verteilt werden (*Anything, Anywhere*). Damit soll einerseits hohe Leistung und Skalierbarkeit und andererseits auch Fehlertoleranz erreicht werden. *xFS* baut dabei auf der Forschung aus verschiedenen anderen

Bereichen auf: Zur Speicherung der Daten wird eine Kombination aus RAID (*Redundant Arrays of Inexpensive Disks* [19]) und Log-basierten Dateisystemen [68] verwendet. Mittels Redundanz wird, wie bei RAID, Fehlertoleranz erreicht, während gleichzeitig die große Schwäche von RAID, nämlich der durch die Redundanz verringerte Durchsatz, durch eine Log-basierte Verwaltung der Daten auf den Daten-Servern abgemildert wird. Um Daten- und Metadatenkonsistenz zwischen den gleichberechtigten Teilnehmern zu erreichen, kommen Cache-Konsistenztechniken aus dem Bereich der Multiprozessoren [52] zum Einsatz.

In dem vollständig verteilten System, das sich darüber hinaus auch dynamisch ändern kann, stellt sich das Wiederauffinden von Daten und Metadaten als ein großes Problem dar. Um die verschiedenen Informationen aufzufinden, werden 4 Datenstrukturen beziehungsweise Abbildungen verwendet: *Manager Map*, *imap*, *file directory* und *stripe group map*.

Die *stripe group map* enthält mögliche Verteilungsschemata der Dateiinhalte auf die Daten-Server. Die Art der Verteilung, das heißt, welche *stripe groups* für die Verteilung verwendet wird, kann für jede Datei separat festgelegt werden. Die Datenstruktur wird global auf allen Rechnern repliziert. Intern werden die Dateien in *xFS*, wie auch in anderen Dateisystemen, über eine Indexnummer (*Inode-Nummer*) angesprochen. Die Abbildung von Dateinamen auf Dateiindex wird durch die *file directory* Datenstruktur ermöglicht, die auf den Daten-Servern gespeichert wird. Weil *xFS* Verzeichnisse wie normale Dateien verwaltet, entspricht die Suche nach dem Dateiindex einer Suche in einer bestimmten Verzeichnisdatei. Die *Imap* Datenstruktur enthält Informationen über die Position des Logs für diese Datei und ist auf dem für die Datei zuständigen Metadatenmanager zu finden. Dieser wird wiederum mit Hilfe der *Manager Map* bestimmt.

Die *Manager Map* beschreibt die Abbildung von einem Dateiindex auf einen Metadatenmanager. Dazu wird ein Hashwert des Dateiindexes berechnet und als Index in der *Manager Map* verwendet. Diese Datenstruktur ist im Grunde nur eine Tabelle, die alle Metadatenmanager enthält. Durch die Indirektion ist es möglich, dass Metadatenmanager im laufenden Betrieb hinzukommen können oder auch das System verlassen können. Da es sich um eine kleine Datenstruktur handelt, ist sie global auf allen Knoten repliziert, so dass es den Klienten möglich ist, direkt den verantwortlichen Manager zu finden und anzusprechen.

Es ist wichtig festzustellen, dass es sich bei *xFS* in weiten Teilen um einen konzeptionellen Entwurf handelt, dessen Funktionstüchtigkeit nicht vollständig gezeigt wurde. Im einzelnen lässt sich nicht feststellen, welche Funktionalität tatsächlich lauffähig war und welche nur per Simulation untersucht wurde. Klar ersichtlich ist allerdings, dass alle dynamischen Funktionen, wie beispielsweise das Ändern der Verteilung von Metadaten auf Metadatenmanager und auch dynamische Veränderungen an der *Stripe Group Map*, nicht innerhalb des Prototyps implementiert wurden.

Obwohl *xFS* eine Verteilung von Metadaten auf verschiedene Metadatenmanager anhand ihres Dateiindexes erlaubt, existieren keine Messungen, welche die angebliche Lastbalancierung zeigen. Auch wird nicht beschrieben, wie die Datenstruktur *Manager Map* dynamisch manipuliert werden müsste, um dies zu erreichen.

2.3.5 Lazy Hybrid

Lazy Hybrid (LH) [10] ist eine Metadatenverteilungstechnik, die versucht, die Verteilung nach Verzeichnissen und nach Hashwerten miteinander zu kombinieren. Dabei sollen die Vorteile beider Verfahren, nicht jedoch deren Nachteile übernommen werden.

Ähnlich wie einer reinen Streuverteilung, werden bei LH die Metadaten anhand eines Hashwertes einem Metadatenmanager zugewiesen. Der Hashwert wird aus dem Pfad der Datei oder des Verzeichnisses berechnet, wodurch die Kosten einer Traversierung des Pfades vermieden werden sollen.[1] In Verzeichnissen werden aber trotzdem noch hierarchische Informationen gespeichert, um die Auflistung von Verzeichnissen und andere Operationen auf Verzeichnissen zu ermöglichen. Wenn ein Klient auf eine Datei zugreifen möchte, berechnet er zunächst den Hashwert und benutzt diesen als Index in eine global replizierte Tabelle (*Metadata Lookup-Table*, MLT), um den verantwortlichen Dienstgeber zu finden. Durch die Indirektion über die MLT ist es relativ einfach möglich, Metadatenmanager hinzuzufügen oder zu löschen.

Ein Problem, das durch die fehlende Traversierung des Pfades entsteht, ist die Zugriffskontrolle in POSIX, bei der ein Zugriff auf eine Datei durch Restriktionen der Oberverzeichnisse eingeschränkt werden kann. Dies soll bei LH durch Zugriffskontrolllisten (*Access Control Lists (ACL)*) behoben werden, allerdings bleibt unklar, wie genau dies funktionieren soll.

Bei LH gibt es aufgrund der Architektur vier problematische Operationen, die besonders aufwändig sein können: 1. Durch die *Änderung der Zugriffsrechte eines Verzeichnisses* müssen die ACLs aller untergeordneten Verzeichnisse und Dateien modifiziert werden. 2. Das *Umbenennen eines Verzeichnisses* ändert nicht nur den Hashwert des Verzeichnisses selber, sondern auch den aller untergeordneten Verzeichnisse und Dateien, die in Folge dessen unter Umständen auf einen anderen Metadatenmanager migrieren müssen. 3. Beim *Löschen eines Verzeichnisses* entsteht ein ähnlicher Aufwand, nur müssen hier die Metadaten der untergeordneten Verzeichnisse und Dateien gelöscht werden. 4. Wird die *MLT verändert*, weil beispielsweise ein weiterer Metadatenmanager hinzukommt, muss ein Teil der Metadaten auf diesen neuen Dienstgeber „umziehen“, eine globale Operation, die mit hohem Synchronisierungs- und Kommunikationsaufwand verbunden ist.

Um diesen potentiellen Problemen zu begegnen, werden die notwendigen Aktualisierungen verzögert bis auf die betroffenen Metadaten zugegriffen wird (*Lazy Update*). Dazu besitzen alle Metadatenmanager Invalidierungslisten, die den Zugriff auf veraltete Daten verhindern. Diese Listen entstehen auf Grund von Invalidierungs- oder Aktualisierungsnachrichten, welche die Metadatenmanager austauschen.

Das *Lazy Hybrid* Metadatenmanagement existiert nur als konzeptioneller Vorschlag, an einem Prototyp wurde zwar im Jahre 2004 gearbeitet [11], aber dieser wurde bis heute nicht fertig gestellt. Obwohl Simulationen gute Ergebnisse zeigten, ist schwer vorhersehbar, ob das Verfahren wirklich den anderen Verfahren überlegen ist, so lange die Implementierung aussteht.

[1] Aufgrund der Betrachtungen über den Aufbau der Dateisystemschnittstelle des Linux-Kerns in Abschnitt 3, ist es zweifelhaft, ob sich diese Technik in aktuellen UNIX-Dateisystemen implementieren lässt.

2.3.6 Zusammenfassung

Nach umfangreicher Literaturrecherche konnte kein Cluster-Dateisystem gefunden werden, das erfolgreich eine Metadatenverteilung implementiert und sich über die Dateisystemschnittstelle eines Betriebssystems montieren lässt. Einige Dateisysteme existieren nur als Simulationen (*Lazy Hybrid*) oder besitzen diese Fähigkeit nur auf dem Papier (*Lustre*, *xFS*). Andere Dateisysteme sind nicht verfügbar oder es existieren keine Informationen über die Effektivität der Metadatenverteilung (*Frangipani*).

Kapitel 3

Grundlagen: Linux Dateisysteme

Bei *CLF* handelt es sich um einen Dateisystem-Prototyp, der nicht nur für eine Simulationsumgebung, sondern für den Einsatz unter realistischen Bedingungen in einem Cluster entworfen wurde. Der Prototyp muss zeigen, dass die von ihm neu eingeführten Techniken auch unter Wahrung der vom Betriebssystem vorgeschriebenen Schnittstelle anwendbar sind. Aus diesem Grund wird in diesem Kapitel zunächst ein Blick auf die Dateisystemschnittstelle *VFS* des Linux Kerns geworfen und insbesondere auch auf die Einschränkungen, die durch ihre Nähe zur Implementierung von EXT2, des Standarddateisystems von Linux, entstanden sind.

3.1 Geschichte des *VFS*

Die Möglichkeit unterschiedliche Dateisysteme gleichzeitig auf die gleiche Art und Weise, das heißt über eine einheitliche Schnittstelle, zu verwenden ist eine Selbstverständlichkeit für moderne Betriebssysteme. Als erstes Betriebssystem besaß 1985 SunOS 2.0 von Sun Microsystems diese Fähigkeit, motiviert durch die gleichzeitige Einführung von NFS (Network File System [70]). Die Technik wurde unter dem Namen *VNODES* [49] bekannt, ist aber heute nur noch als *VFS* oder Virtual Filesystem Switch bekannt. Linux besitzt den *VFS*-Mechanismus seit der Einführung des „Extended file system“ im Jahre 1992 (Linux 0.96c).

Auch wenn die Einführung des *VFS* ursprünglich durch die Einführung eines Netzwerkdateisystems motiviert war, so ist dieser Mechanismus dennoch hauptsächlich auf die effiziente Implementierung von lokalen Dateisystemen ausgerichtet. Auch heute noch orientiert sich die Implementierung an dem Standarddateisystem *Ext2*, das durch die geringen „Reibungsverluste“ sehr effizient implementiert ist. Das hat aber zur Folge, dass theoretisch mögliche Optimierungen, wie zum Beispiel im Bereich der Pfadverfolgung, nur bedingt oder auch gar nicht realisierbar sind.

Dieses Kapitel behandelt die Schnittstellen und Datenstrukturen des Linux Kerns (Version 2.6), die bei der Implementierung eines Dateisystems benutzt werden müssen. Dabei werden hauptsächlich zwei der fünf verschiedenen Dateitypen behandelt: *Dateien* und *Verzeichnisse*. *Symbolische Verknüpfungen* (engl.

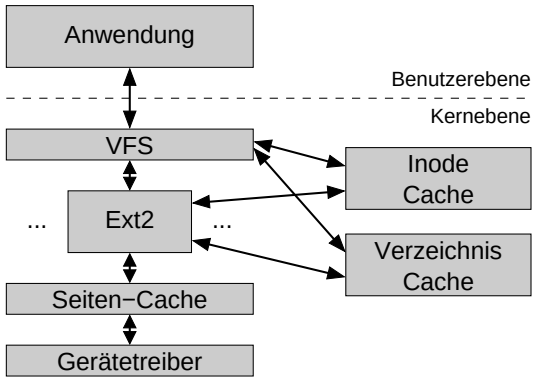


Abbildung 3.1: Der Aufbau des VFS im Linux-Kern

symbolic links), *Gerätedateien* (engl. device files) und *Kanäle* (engl. pipes) werden nicht betrachtet, da sie für die Implementierung von *CLF* nur geringe Bedeutung haben. (Siehe auch [7])

Ziel dieses Abschnittes ist es, einen Einblick in die Implementierung von Dateisystemen unter Linux zu geben und dabei Möglichkeiten und Grenzen des vorgegebenen Rahmenwerks zu betrachten. Das Hauptaugenmerk liegt auf der Frage, welche Aufgaben dem konkreten Dateisystem zufallen und welche direkt von der VFS-Schicht übernommen werden. Kapitel 5 wird bei der Beschreibung des Entwurfs von *CLF* auf dieses Wissen aufbauen.

3.2 Der *Virtual Filesystem Switch*

Der *Virtual Filesystem Switch* (auch bekannt als *Virtual Filesystem*) ist eine Softwareschicht innerhalb des Linux Kerns, die alle Datei-bezogenen Operationen verarbeitet [38]. Abbildung 3.1 zeigt schematisch die Rolle dieser Schicht: Sie ist für die Verarbeitung aller Dateizugriffe auf der Benutzerebene zuständig. Funktionalität, die allen Dateisystemen gemeinsam ist, wie beispielsweise die Pfadauflösung und die Kontrolle der Zugriffsrechte werden von ihr übernommen, um unnötige Duplikationen zwischen den Dateisystem-Implementierungen zu vermeiden. Diese Funktionalität kann durch Einschubmethoden des spezifischen Dateisystems (in der Abbildung steht das Dateisystem *Ext2* stellvertretend für eine beliebiges spezifisches Dateisystems) erweitert oder teilweise auch ersetzt werden. Daher müssen spezifische Dateisysteme die gemeinsamen Datenstrukturen, wie beispielsweise die *Inode* Struktur (repräsentiert eine Element des Dateisystems, wie beispielsweise eine Datei oder ein Verzeichnis) benutzen und auch Gebrauch von den vorgegebenen Caches machen. In der Abbildung sind der *Inode Cache* und der *Verzeichnis Cache* als separate Module dargestellt, man kann sie aber durchaus auch als Teil des VFS ansehen.

Unterhalb eines spezifischen Dateisystems findet sich der *Seiten-Cache* (engl. Page Cache), den lokale Dateisysteme benutzen können, um auf das darunter liegende Gerät zuzugreifen.[1] Durch diese Architektur ist das Dateisystem un-

[1] Seit Linux 2.4 ist der Seiten-Cache mit dem so genannten *Buffer-Cache* zusammengelegt

abhängig von dem zugrundeliegenden Medium, da es über eine standardisierte Schnittstelle mit dem Gerätetreiber kommuniziert. Aber auch Dateisysteme, die ohne Block-Geräte (engl. block device) arbeiten, wie beispielsweise Netzwerke-Dateisysteme können den Seiten-Cache benutzen. Allerdings müssen sie die Seiten dann selber beschreiben, oder einen speziellen Gerätetreiber anbieten, der die Lese- und Schreibanforderungen über das Netzwerk an den zuständigen Rechner weiterleitet und die Antworten im Seiten-Cache ablegt.

Die Grundidee hinter dem VFS ist die Einführung eines gemeinsamen Dateimodells, an das sich die unterstützten Dateisysteme halten müssen. Dieses Modell entspricht dem der traditionellen Unix-Dateisysteme, da Linux seine nativen Dateisysteme (Ext, Ext2, Ext3) mit möglichst minimalem Overhead durch die zusätzliche Schicht benutzen möchte. Daher muss jede spezifische Dateisystem-Implementierung ihre physikalische Organisation auf dieses Modell abbilden.

Ein Beispiel dafür ist die Verwaltung von Verzeichnissen. Nach dem Unix-Dateimodell wird ein Verzeichnis wie eine Datei behandelt, die eine Liste aller in diesem Verzeichnis enthaltenen Dateien und Verzeichnisse speichert. Im Gegensatz dazu speichert beispielsweise das MS/DOS-Dateisystem die Position aller Dateien mit Hilfe einer Tabelle (FAT = *File Allocation Table*). Bei diesem Dateisystem sind Verzeichnisse keine Dateien und daher muss die Linux-Implementierung dieses Dateisystems, die Verzeichnis-"Dateien" bei Bedarf erstellen, um die Schnittstelle zu erfüllen, obwohl diese Dateien kein physikalisches Äquivalent auf dem Datenträger besitzen.

Die Benutzung des Unix-Dateimodells ist die größte Einschränkung des VFS. Sie ist aber unumgänglich, da Linux alle Dateisysteme innerhalb eines großen, einheitlichen Baumes verwaltet.

3.3 Objekte des VFS

Bei der Implementierung eines Dateisystems für Linux ist der Programmierer auf die Benutzung von vier Objekten festgelegt: `Superblock-`, `Inode-`, `File-` und `Dentry`-Objekte. Der Linux-Kern ist zwar in der nicht-objektorientierten Programmiersprache *C* programmiert, aber die Schnittstelle des VFS ist trotzdem im gewissen Maße objektorientiert, wenn man ein Objekt als ein Softwarekonstrukt sieht, das sowohl die Datenstruktur, wie auch die Methoden, die diese manipulieren, definiert.

Die Objekte sind als normalen C-Strukturen implementiert, die neben den Datenfeldern auch Funktionszeiger enthalten, die Funktionen mit festgelegten Signaturen referenzieren. Diese Funktionen entsprechen den Methoden des Objekts und ihre Signaturen enthalten in der Regel als ersten Parameter einen Zeiger auf das Objekt (die Struktur) auf der die Operation ausgeführt werden soll. In manchen Fällen können die Methoden auch ohne Implementierung bleiben (der Funktionszeiger enthält in diesem Fall den Wert „0") um anzuzeigen, dass die Standardimplementierung der VFS-Schicht verwendet werden soll.

3.3.0.1 Das Superblock-Objekt

speichert Informationen über ein Dateisystem, wie beispielsweise die Gerätenummer, die Blockgröße, Daten zur Festplatten-Quota und Listen zur Verwal-

worden, der in Linux 2.2 zwischen dem Seiten-Cache und dem Gerätetreiber angeordnet war.

tung von Inode- und File-Objekten. Die Methoden von Superblock-Objekten dienen der Handhabung von Inodes, der Überprüfung von Quota-Beschränkungen und dem Abmontieren (engl. unmount) des Dateisystems. Zu jedem montierten Dateisystem existiert genau ein Superblock-Objekt.

3.3.0.2 Das Inode-Objekt

speichert allgemeine Informationen über eine Datei oder ein Verzeichnis, wie beispielsweise Zugriffsrechte, Besitzer, Größe oder Änderungsdatum. Es speichert jedoch nicht den Dateinamen oder andere Informationen bezüglich des Pfades der Datei, stattdessen ist jedes Objekt mit einer innerhalb des Dateisystems eindeutigen Nummer verknüpft, über die jede Datei und jedes Verzeichnis identifiziert werden kann. Die Methoden dieser Objekte dienen zum Erstellen, Löschen und Verändern von Dateien und Verzeichnissen, sowie zu deren Auffinden. Auch *Symbolische Verknüpfungen*, *Gerätedateien* und *Kanäle* werden durch Inode-Objekte repräsentiert. Diese Arten von Inodes werden jedoch in den folgenden Betrachtungen nicht behandelt.

3.3.0.3 Das File-Objekt

speichert Informationen zur Interaktion zwischen Prozessen und einer offenen Datei (im wesentlichen die aktuelle Position in der Datei und die Optionen mit denen die Datei geöffnet wurde). Die Methoden von File-Objekten dienen dem Öffnen, Lesen, Schreiben und Schließen von Dateien, sowie dem Lesen von Verzeichnissen. Weiterhin sind auch Operationen zur Speicherabbildung (engl. memory mapping) und zum Sperren (engl. locking) von Dateien innerhalb dieses Objektes definiert.

3.3.0.4 Das Dentry-Objekt

speichert die Verknüpfung zwischen dem Dateinamen mit dem zugehörigen Inode-Objekt (directory entry). Weiterhin speichert es auch einen Verweis auf das Dentry-Objekt des Oberverzeichnisses. Im Falle von Verzeichnissen werden auch Zeiger auf die in diesem Verzeichnis enthaltenen Dentry-Objekte verwaltet. Dentry-Objekte werden bei der Suche nach Dateien (Funktion: `lookup()`) erstellt und in einem speziellen Cache (Dentry-Cache bzw. *Verzeichnis Cache* in Abb. 3.1) gespeichert, der bei weiteren Suchen nach dieser Datei den Zugriff auf die unteren Ebenen (Page-Cache, Festplatte) vermeidet. Dentry-Objekte besitzen Methoden zu ihrer eigenen Verwaltung, das heißt zum Vergleichen, Löschen und Freigeben der Einträge, sowie die für Netzwerkdateisysteme wichtige Funktion `d_revalidate()`, mit der das VFS abfragt, ob das Objekt noch gültig ist.

Jedes dieser Objekte besitzt einen Zeiger, mit dem auf so genannte „private“ Informationen des Objekts verwiesen werden kann. Dieser Zeiger wird von einigen Dateisystemen dazu verwendet weitere Informationen zu referenzieren, die über die in den Standardfeldern gespeicherten hinausgehen. Bei der bereits erwähnten Implementierung des MS/DOS-Dateisystems kann so beispielsweise einem Verzeichnis ein Verweis auf die FAT-Tabelle angehängt werden.

Üblicherweise werden dazu von den Dateisystemen Strukturen angelegt, die diese Informationen enthalten und durch den Zeiger an die vom VFS erwarte-

ten Objekte „angehängt“ werden. Neben des Speichers für das eigentliche Objekt wird zusätzlicher Speicher angefordert, so dass auf diese Weise erhöhte Speicherfragmentierung verursacht wird. Dies ist gerade bei den vielbenutzten Inode-Objekten schädlich für die Leistungsfähigkeit der Implementierung.

Aus diesem Grund gibt es seit der Kern-Version 2.6 zusätzliche Funktionen im Superblock-Objekt, die vom VFS für die Speicheranforderung der Inode-Objekte verwendet werden, sofern sie von einem Dateisystem implementiert wurden. Mit Hilfe dieser Funktion kann der Speicher für die Standard-Inode-Objekte und die zusätzlichen Informationen eines Dateisystems „am Stück“ angefordert und auch gemeinsam freigegeben werden. So lässt sich zumindest bei den am häufigsten benutzten Datenstrukturen eines Dateisystems, den Inodes, die zusätzliche Felder zur Speicherung von Informationen benötigen, eine Reduktion im Speicherverbrauch und eine zeiteffizientere Allozierung und Freigabe erreichen.

3.3.1 Beispiel einer Dateioperation

Zur Veranschaulichung des Zusammenwirkens der Objekte betrachten wir den Aufruf der Funktion `fd = open("/path/file", O_RDONLY)` innerhalb eines Benutzerlevel-Programms. Der Aufruf bewirkt, dass die Datei `/path/file` im *nur-lesen*-Modus geöffnet wird und ein Dateideskriptor (ein Prozess-eindeutiger Index) angelegt wird, unter dem folgende Funktionsaufrufe die Datei referenzieren können. Die Funktion „open“ ist eine POSIX-Funktion [21], die von einer C-Bibliothek (zum Beispiel Glibc [30]) implementiert wird und einen Systemaufruf auf die Kern-Funktion `sys_open` durchführt.

`sys_open` ist Teil der VFS-Schicht und muss nun zunächst die Datei finden, oder genauer gesagt: Das Inode-Objekt, das die Datei repräsentiert. Die Funktion `do_path_lookup` traversiert dazu den Dateibaum. Im Fall von „/path/file“ stellt die Funktion zunächst fest, dass es sich um einen absoluten Pfad handelt, daher greift sie auf das im Kern gespeicherte Wurzeldateisystem zu und erhält das Inode-Objekt des Wurzelverzeichnisses.

Bis zu diesem Punkt ist der Ablauf für alle Dateisysteme gleich, aber die Suche nach dem Unterverzeichnis „path“ ist eine Operation, die sich von Dateisystem zu Dateisystem unterscheidet. Daher wird die Suche in einem Verzeichnis durch eine Methode des Inode-Objektes ausgeführt. Jedes neue Inode-Objekt wird so initialisiert, dass die Funktionszeiger auf die Methoden „seines“ Dateisystems verweisen. In diesem Fall wird die Methode `lookup()` benutzt, um festzustellen, ob ein Unterverzeichnis mit dem Namen „path“ existiert. Angenommen, dieses Verzeichnis existiert, dann fordert die `lookup()`-Methode vom Inode-Cache die Inode mit der gesuchten Nummer an (Funktion: `iget()`) und initialisiert sie; dazu gehört auch die Initialisierung der Funktionszeiger.

Nachdem das Inode-Objekt für das Verzeichnis „path“ angelegt wurde, übernimmt wieder die `do_path_lookup` Funktion die Kontrolle und benutzt zur Suche nach der Datei „file“ wiederum die `lookup`-Methode eines Inode-Objektes, in diesem Fall die des gerade erstellten Objektes, welches das Verzeichnis repräsentiert. Diese Operation erzeugt (wiederum unter der Annahme, dass die Datei existiert) ein entsprechendes Inode-Objekt für die Datei.

Zum Öffnen der Datei wird schließlich ein Objekt vom Typ `File` erzeugt und die `open()` Funktion aufgerufen, auf die in der Datenstruktur von `File` verwiesen wird. Die `File`-Struktur existiert nicht auf dem Speicher-Medium,

sondern nur im Hauptspeicher, als ein Konstrukt, das Anwendungen den Zugriff auf Dateien erlaubt. Im letzten Schritt weist die `sys_open`-Funktion schließlich dem `file`-Objekt einen Dateideskriptor (engl. file descriptor) zu, über den die Anwendung auf die Datei zugreifen kann.

Anmerkung: Tatsächlich ist die Suche nach einer Datei aufwändiger, als hier dargestellt wurde. Insbesondere die Verwendung des Inode- und des Verzeichnis-Caches (engl. dentry cache) wurde zur Vereinfachung weggelassen.

Dieses Beispiel zeigt, dass das VFS einem Dateisystem Freiheiten bei der Implementierung der entsprechenden Einschubmethoden (`open`, `lookup`, etc.) erlaubt, aber die Methoden auf die Verwendung der gemeinsamen Dateisystem-Objekte und damit auf das gemeinsame Dateimodell festlegt.

3.4 Systemaufrufe vs VFS-Aufrufe

Tabelle 3.1 zeigt alle Betriebssystemaufrufe für die das VFS zuständig ist, mit Ausnahme von Operationen bezüglich Gerätedateien, Kanälen, symbolischen Verknüpfungen und Netzwerkoperationen. Die Tabelle enthält eine vollständige Liste dieser Aufrufe, allerdings werden die in *CLF* nicht implementierte Operation grau dargestellt. Es handelt sich dabei um asynchrone Lese- und Schreiboperationen (eine Neuerung in Linux 2.6), Kombinationen aus Datei- und Netzwerkoperationen, Synchronisationsoperationen und die Verwaltung von erweiterten Attributen (auch eine Neuerung in Linux 2.6). Auf die Implementierung der Operationen für den Prototyp wurde verzichtet, da es sich um „optionale“ Operationen handelt, deren Implementierung keinen weiteren Erkenntnisgewinn gebracht hätte.

Wie im vorhergehenden Abschnitt (3.3.1) anhand eines Beispiels gezeigt wurde, korrespondieren die Einschubmethoden der VFS-Objekte nicht in allen Fällen 1:1 mit den Betriebssystemaufrufen. Insbesondere die für diese Dissertation wichtige Metadaten-Verwaltung wird bereits teilweise vom VFS selber übernommen. Dies ist einerseits ein Vorteil, da sich so die notwendige Implementationsarbeit reduziert, aber andererseits schränkt dies auch die Optimierungsmöglichkeiten ein. Zur Illustration wird in diesem Abschnitt das Verhältnis zwischen den Systemaufrufen und den möglichen Aufrufen an VFS-Einschubmethoden durch die Einführungen von Kategorien systematisiert.

Dabei ist zu beachten, dass das VFS bei jedem Systemaufruf Vor- und Nacharbeiten leistet und in keinem Fall den Kontrollfluss direkt an die Einschubmethoden übergibt. Zu diesen Arbeiten gehört beispielsweise das Überprüfen von Zugriffsrechten auf Dateien, Verzeichnisse und auf eventuell übergebene Speicherbereiche.

3.4.1 Kategorie 1: Reine VFS-Aufrufe

Zu dieser Kategorie gehören Systemaufrufe, die vollständig von der VFS-Schicht mit Hilfe der VFS-Objekte durchgeführt werden ohne auf eine native Dateisystemoperation in Form einer Einschubmethode zuzugreifen. Sie gliedern sich weiterhin in vier Gruppen:

1. Funktionen, die Dateideskriptoren manipulieren, nicht aber die Datei selber: `dup() dup2()`

Systemaufruf	Beschreibung
mount() umount() umount2()	Montieren von Dateisystemen
sysfs()	Liefert Informationen über verfügbare Dateisysteme
statfs() fstatfs() statfs64() fstatfs64() ustat()	Liefert Statistiken über ein Dateisystem
chroot() pivot_root()	Ändern des Wurzelverzeichnisses
chdir() fchdir() getcwd()	Ändern des aktuellen Verzeichnisses
mkdir() rmdir()	Erstellen und Entfernen von Verzeichnissen
getdents() getdents64() readdir() link() unlink() rename()	Abfragen und Ändern von Verzeichniseinträgen
chown() fchown() lchown() chown16() fchown16() lchown16() chmod() fchmod()	Ändern des Besitzers und der Zugriffsrechte der Datei
stat() fstat() lstat() access() oldstat() oldfstat() oldlstat() stat64() lstat64() fstat64()	Abfragen des Dateistatus
open() close() creat() umask()	Öffnen, Schließen und Erzeugen von Dateien
dup() dup2() fcntl() fcntl64()	Handhabung von Dateideskriptoren
select() poll()	Warten auf Ereignisse einer Menge von Dateideskriptoren
truncate() ftruncate() truncate64() ftruncate64()	Ändern der Dateigröße
lseek() _llseek()	Ändern der Dateiposition
read() write() readv() writev() sendfile() sendfile64() readahead()	Lese- und Schreiboperationen
io_setup() io_submit() io_getevents() io_cancel() io_destroy()	Asynchrone Lese- und Schreiboperationen
pread64() pwrite64()	Setzen der Dateiposition und anschließender Zugriff
mmap() mmap2() munmap() madvise() mincore() remap_file_pages()	Speicherabbildung
fdatasync() fsync() sync() msync()	Synchronisieren von Dateien
flock()	Abschließen von Dateien
setxattr() lsetxattr() fsetxattr() getxattr() lgetxattr() fgetxattr() listxattr() llistxattr() flistxattr() removexattr() lremovexattr() fremovexattr()	Setzen, Lesen und Ändern von erweiterten Attributen

Tabelle 3.1: Systemaufrufe an das VFS (nach [7])

Grau gedruckte Systemaufrufe sind in *CLF* nicht implementiert.

2. Funktionen, welche die Seitenverwaltung betreffen, aber nicht direkt auf Dateien zugreifen: `madvise()`, `mincore()` und `remap_file_pages()`

3. Funktionen, welche den aktuellen Prozess betreffen: `umask()` `getcwd()`. `umask()` ändern die Standard-Sichtbarkeit von neu erzeugten Dateien indem das entsprechende Feld in der Prozess-Datenstruktur geändert wird. Anfragen an `getcwd()` werden vollständig aus dem Dentry-Cache erfüllt und erfordern keinen Zugriff auf das native Dateisystem.

4. Die Funktion `fchdir()` ändern das aktuelle Verzeichnis eines Prozesses auf den Pfad einer geöffneten Datei.

3.4.2 Kategorie 2a: Aufrufe, die das gesamte Dateisystem betreffen

Die Systemaufrufe dieser Kategorie betreffen keine einzelnen Dateien oder Verzeichnisse, auch wenn beispielsweise die Aufrufe zum Montieren des Dateisystems einen Pfad als Parameter erwarten. Aus Sicht des Dateisystems in dem das Verzeichnis existiert auf das ein weiteres Dateisystem montiert wird, gehören diese Befehle in die Kategorie 3.

Der Linux-Kern führt eine Liste mit den unterstützten Dateisystemtypen, welche durch `file_system_type`-Objekte repräsentiert werden. Diese Liste ist dynamisch, weil durch das Laden von Modulen zur Laufzeit weitere Dateisysteme hinzukommen können.

Die Aufrufe in dieser Kategorie gliedern sich in drei Gruppen und benutzen jeweils genau eine Einschubmethode aus dem Superblock-Objekt oder dem `file_system_type`-Objekt:

1. `sysfs()` liefert Informationen über die im Kern registrierten Dateisysteme. Die Informationen stammen aus verschiedenen Feldern des `file_system_type`-Objektes.

2. Funktionen zum Montieren und De-Montieren des Dateisystems: `mount()`, `umount()` und `umount2()` rufen die Methoden `get_sb()` beziehungsweise `kill_sb()` zum Erzeugen und Freigeben des Superblock-Objektes auf.

3. Funktionen zum Abfragen von Statistiken über das Dateisystem: `statfs()`, `statfs64()` und `ustat()`. Die aufgezählten Funktionen führen zum Aufruf der `statfs()` Methode des Superblock-Objektes.

3.4.3 Kategorie 2b: Aufrufe mit Dateideskriptor

Diese Systemaufrufe arbeiten mit Dateideskriptoren und vereinfachen dadurch die Kontrollmechanismen der VFS Schicht. In den Aufrufen muss nur sichergestellt werden, dass der übergebene Deskriptor gültig ist, das heißt auf eine geöffnete Datei verweist und auf eventuell übergebene Speicherbereiche zugegriffen werden kann. Die Existenz der Datei und die erforderlichen Zugriffsrechte können als gegeben vorausgesetzt werden und auch die Suche nach der Datei im Dateibaum entfällt.

Bei Aufrufen von Funktionen dieser Kategorie wird genau eine Einschubmethode aufgerufen (falls definiert), welche die eigentliche Aufgabe zu erfüllen hat. Es lassen sich darüber hinaus vier Gruppen unterscheiden:

1. Funktionen, die auf Methoden des Superblock-Objektes zugreifen: `fstatfs()` und `fstatfs64()` rufen die Funktion `statfs` des Superblock-Objektes der aktuellen Datei auf.

2. Funktionen, die auf Methoden des Inode-Objektes zugreifen: `fchown()`, `fchown16()`, `ftruncate()`, `ftruncate64()` und `fchmod()` bewirken einen Aufruf der Methode `setattr()` (falls definiert) des Inodes-Objektes, welches mit der geöffneten Datei assoziiert ist. Alle betroffenen Felder und die neuen Werte werden in eine speziellen Struktur (`struct iattr`) als Parameter übergeben. Im Fall der Systemaufrufe `ftruncate()` und `ftruncate64()` wird die Methode `truncate()` des Inode-Objektes aufgerufen. Die Systemaufrufe `fstat()`, `oldfstat()` und `fstat64()` dienen zur Abfrage von Feldern des Inode-Objektes. Bevor die Daten aus dem Inode-Objekt kopiert werden, wird zunächst die Einschubmethode `getattr()` aufgerufen um die Aktualität der Felder sicher zu stellen.

3. Funktionen, die auf Methoden des File-Objektes zugreifen. Hinter den Systemaufrufen `getdents()`, `getdents64()` und `readdir()` steht jeweils ein Aufruf der Methode `readdir()` des File-Objektes, das in diesem Fall ein Verzeichnis repräsentiert. Der Aufruf von `lseek()`, `_llseek()`, `read()`, `write()`, `pread64()` und `pwrite64()` führt zu einem Aufruf der gleichnamigen Methoden, wobei die Implementierung der `lseek()` Methode optional ist. Bei den Funktionsaufrufen `readv()` und `writev()` werden entweder die gleichnamigen Methoden zum Lesen bzw. Schreiben in die übergebene Liste der Speicherbereiche aufgerufen, oder es erfolgt ein mehrfacher Aufruf der `read()` bzw. der `write()` Methoden. Bei Aufrufen von `mmap()` und `mmap2()` kommt die `mmap()` Methode des File-Objektes zum Einsatz. Der Aufruf von `fcntl()` und `fcntl64()` führen zu einem Aufruf der VFS-Methode `do_fcntl()`, welche in Abhängigkeit von den übergebenen Parametern die `File`-Objekt-Methoden `check_flags()`, `dir_notify`, `lock()` oder `tasync` aufrufen oder auch nur zu einer Manipulation der Prozess- oder File-Objekte führen. Der Systemaufruf `close()` verringert schließlich den Referenzzähler der File-Objektes. Erreicht dieser den Wert 0, ruft das VFS die `release()` Methode des File-Objektes auf.

4. Die Systemaufrufe `readahead()` und `munmap()` betreffen die Seitenverwaltung des Kerns. Die durch sie aufgerufenen Einschubmethoden gehören zu den Speicherverwaltungsoperationen, die innerhalb eines `address_space`-Objektes definiert werden, das in dem Inode-Objekt referenziert wird und Informationen über die im Seiten-Cache gespeicherten Seiten einer Datei trägt. `readahead` kann, sofern die Seiten nicht bereits geladen wurden, zu einem mehrfachen Aufruf der `readpage()` Methode führen, sofern diese implementiert wurde, andernfalls wird mit einem Fehler abgebrochen. Bei `munmap()` werden gegebenenfalls die Methoden `writepage()` bzw. `writepages()` verwendet, um veränderte Seiten zurückzuschreiben. Genau genommen greift `munmap()` nicht über einen Dateideskriptor auf die Datei zu, sondern über eine im `address_space`-Objekt gespeicherte Referenz auf das Inode-Objekt.

3.4.4 Kategorie 3: Aufrufe mit Dateisuche

In die letzte Kategorie fallen schließlich Aufrufe, die neben ihrer eigentlichen Funktion auch eine oder mehrere Suchen innerhalb des Dateisystems auslösen, welche zu weiteren Aufrufen von Einschubmethoden führen. Beispielhaft wurde dieser Vorgang in Abschnitt 3.3.1 gezeigt. Der folgende Abschnitt 3.5 wird die Auflösung von Dateinamen im Detail betrachten, an dieser Stelle wird zunächst nur die eigentlich Funktion des Systemaufrufs betrachtet, die im Anschluss an die Pfadauflösung ausgeführt wird.

Die Aufrufe lassen sich in folgende drei Gruppen gliedern:

1. Die Systemaufrufe dieser Kategorie führen, abgesehen von den für die Pfadauflösung notwendigen Aufrufen, zu keinen weiteren Aufrufen von Einschubmethoden: `chroot()` und `pivot_root()` ändern das Wurzelverzeichnis des aktuellen Prozesses und `chdir()` ändert das aktuelle Verzeichnis des Prozesses.

2. Der Systemaufruf `open()` führt zum Aufruf der gleichnamigen Methode in dem entsprechenden File-Objekt. Alle übrigen Zugriffe auf Methoden aus File-Objekten finden über Dateideskriptoren statt.

3. Der Großteil der Aufrufe in dieser Kategorie bezieht sich auf Inode-Objekte, das heißt Dateien und Verzeichnisse. Zum Erzeugen von Dateien und Verzeichnissen werden die Aufrufe `creat()` und `mkdir()` verwendet, genau wie die Gegenstücke zum Löschen (`unlink()` und `rmdir()`) zum Aufruf der gleichnamigen Methoden in dem Inode-Objekt des Oberverzeichnisses führen. Der Systemaufruf `rename()` dient nicht nur dem Umbenennen von Dateien und Verzeichnissen sondern auch dem Verschieben. Auch hier ruft das VFS die gleichnamige Methode in der Inode des Oberverzeichnisses auf. Zum Ändern des Besitzers, der Sichtbarkeit oder der Länge einer Datei (abgesehen von der Länge lassen sich diese Operationen auch auf Verzeichnisse anwenden) durch die Funktionen `chown()`, `lchown()`, `chown16()`, `lchown16()`, `chmod()`, `truncate()` und `truncate64()` ruft das VFS die Einschubmethode `setattr()` aus, um das darunter liegende Dateisystem über die Änderungen zu informieren. Zur Abfrage der Felder eines Inode-Objektes können die Systemaufrufe `stat()`, `lstat()`, `access()`, `oldstat()`, `oldlstat()`, `stat64()` und `lstat64()` verwendet werden. Das VFS stellt zunächst durch den Aufruf der Einschubmethode `getattr()` sicher, dass die Daten aktuell sind und kopiert sie anschließend in den gewünschten Speicherbereich oder prüft sie (im Falle von `access()`).

3.4.5 Übersicht über alle Kategorien

Tabelle 3.2 listet noch einmal alle Kategorien und die ihnen zugeordneten Systemaufrufe auf. Wie auch schon in die vorhergehenden Abschnitten, wurde Aufrufe weggelassen, die für *CLF* nicht implementiert wurden.

Kategorie 1: Reine VFS-Aufrufe	
Gruppe	*Aufrufe*
1. Manipulation von Dateideskriptoren	`dup() dup2()`
2. Seitenverwaltung	`madvise() mincore() remap_file_pages()`
3. Prozessoptionen	`getcwd() umask()`
4. Verarbeitung im VFS	`fchdir()`

Kategorie 2a: Aufrufe, die das Dateisystem als Gesamtheit betreffen	
Gruppe	*Aufrufe*
1. Registrierte Dateisysteme	`sysfs()`
2. Montieren von Dateisystemen	`mount() umount() umount2()`
3. Statistiken über ein Dateisystem	`statfs() statfs64() ustat()`

Kategorie 2b: Aufrufe mit Dateideskriptor	
Gruppe	*Aufrufe*
1. Aufruf von `superblock`-Methoden	`fstatfs() fstatfs64()`
2. Aufruf von `inode`-Methoden	`fchown() fchown16() ftruncate() ftruncate64() fchmod() fstat() oldfstat() fstat64()`
3. Aufruf von `file`-Methoden	`getdents() getdents64() readdir() lseek() _llseek() read() write() pread64() pwrite64() readv() writev() mmap() mmap2() fcntl() fcntl64() close()`
4. Seitenverwaltung	`readahead() munmap()`

Kategorie 3: Aufrufe mit Dateisuche	
Gruppe	*Aufrufe*
1. Kein Aufruf von Einschubmethoden	`chroot() pivot_root() chdir()`
2. Aufruf von `file`-Methoden	`open()`
3. Aufruf von `inode`-Methoden	`creat() mkdir() unlink() rmdir() rename() chown() lchown() chown16() lchown16() chmod() stat() lstat() access() oldstat() oldlstat() stat64() lstat64() truncate() truncate64()`

Tabelle 3.2: Systemaufrufe nach Kategorie

3.5 Pfadauflösung

Viele Systemaufrufe erhalten den Pfad der Datei, auf der sie operieren sollen, als Parameter übergeben. Damit die Operation ausgeführt werden kann, muss das VFS zunächst eine *Pfadauflösung* durchführen, also die durch die übergebene Zeichenkette repräsentierte Inode finden. Nachdem dieser Prozess in Abschnitt 3.3.1 beispielhaft vorgeführt wurde, wird dieser Abschnitt die allgemeine Vorgehensweise erläutern. Anschließend folgt eine Beschreibung des Einflusses, den ein konkretes Dateisystems durch Einschubmethoden auf diesen Algorithmus nehmen kann.

3.5.1 Allgemeine Vorgehensweise

Der Algorithmus zerlegt den Pfad in seine Bestandteile, die Pfadelemente genannt werden. Diese Pfadelemente müssen, mit Ausnahme des letzten, Verzeichnisse repräsentieren. Zunächst wird dabei zwischen absoluten Pfaden (beginnen

mit einem „/“) und relativen Dateinamen unterschieden. Im ersten Fall beginnt die Suche im Wurzelverzeichnis des Prozesses, im zweiten Fall im aktuellen Verzeichnis des Prozesses. In der Prozessstruktur sind Zeiger auf die zugehörigen Dentry-Objekte gespeichert, die wiederum mit einem Inode-Objekt verknüpft sind.

Ausgehend von diesem initialen Verzeichnis wird nach dem ersten Element des Pfades gesucht, um das dazu gehörige Inode-Objekt zu finden und zu laden. Wurde die Inode gefunden, wird darin nach der Inode des zweiten Bestandteil des Pfades gesucht. Dies wird für alle Elemente des Pfades wiederholt, bis der gesamte Pfad aufgelöst wurde.

Um unnötige Zugriffe auf das Medium, auf dem das Dateisystem gespeichert ist, zu vermeiden, wird die iterative Suche durch den Verzeichnis-Cache (Dentry-Cache) beschleunigt. Er speichert das Ergebnis von bisherigen Suchen im Hauptspeicher und erlaubt daher bei der wiederholten Suche eines Dateinamens in einem Verzeichnis die Umgehung des Mediums. Die Objekte vom Typ `Dentry` existieren nur zu diesem Zweck: Sie stellen eine Verbindung zwischen den Inode-Objekten und ihren Namen her und sind untereinander im Sinne der Verzeichnishierarchie verknüpft.

In traditionellen Unix-Dateisystemen existiert die Verknüpfung zwischen einer Inode und ihrem Namen nur implizit durch das Oberverzeichnisses. Verzeichnisse werden als Dateien verstanden, welche die im Verzeichnis enthaltenen Objekte als Paare von Inode-Nummern und Namen speichern. Diese Darstellung ermöglicht eine einfache Implementierung der elementaren Such-Operation in der Pfadauflösung, welche in einem Verzeichnis nach einem Namen sucht und als Ergebnis die zugehörige Inode-Nummer liefert.

Das iterative Suchverfahren wird durch die folgenden Eigenschaften von UNIX-Dateisystemen erschwert:

- Die Zugriffsrechte müssen bei jedem Schritt daraufhin überprüft werden, ob der Prozess die für das Lesen des Verzeichnisinhaltes notwendigen Rechte besitzt.

- Bei einem Dateinamen kann es sich um eine symbolische Verknüpfung handeln. In diesem Fall muss die Suche auf den Pfad der Verknüpfung fortgesetzt werden, um anschließend mit dem ursprünglichen Pfad fort zu fahren.

- Durch symbolische Verknüpfungen kann es zu Endlosschleifen bei der Pfadauflösung kommen, wenn eine der Verknüpfungen auf sich selbst verweist. Das VFS muss diese Möglichkeit in Betracht ziehen und die Endlosschleife aufbrechen.

- Auf jedes Verzeichnis kann ein Dateisystem montiert werden. Trifft das VFS bei der Pfadauflösung auf ein solches Verzeichnis, muss die Suche in dem neuen Dateisystem fortgeführt werden.

- Die Pfadauflösung ist Prozess-spezifisch: Der gleiche Pfad kann in unterschiedlichen Prozessen unterschiedliche Dateien beschreiben. Das gilt auch für absolute Pfade, weil das Wurzelverzeichnis eines Prozesses verändert werden kann.

Die Beachtung der genannten Punkte wird von der VFS-Schicht garantiert. Ausgehend von der Funktion `path_lookup()` ist die Auswertung des Pfades im Sinne einer Schablonenmethode implementiert, die zugrunde liegenden Dateisysteme können durch die Implementierung der Einschubmethoden in jeder Iteration Einfluss auf den Ablauf der Pfadauflösung nehmen, jedoch nicht die Auflösung als ganzes ändern oder ersetzen.

3.5.2 Verwendung der Einschubmethoden bei der Pfadauflösung

Im vorangegangenen Abschnitt wurde der grobe Ablauf der Pfadauflösung vorgestellt, wie sie vom VFS im Linux-Kern durchgeführt wird. In diesem Abschnitt soll die Frage geklärt werden, welche Einflussmöglichkeiten ein konkretes Dateisystems auf diesen Prozess hat, oder genauer: An welcher Stelle des Suchalgorithmus werden welche Einschubmethoden aufgerufen?

Die Pfadauflösung ist ein iterativer Algorithmus, der bei der Inode des Wurzelverzeichnisses oder des aktuellen Verzeichnisses des Prozesses beginnt. In jedem Schritt wird ein weiteres Pfadelement aus dem Gesamtpfad entnommen und in dem zuletzt gefundenen Verzeichnis gesucht. Sobald eines der Pfadelemente nicht gefunden werden kann, bricht der Algorithmus mit einem Fehler ab. Diese Schleife findet sich in der Kernel-Routine `__link_path_walk()`, die für die Suche nach der nächsten Inode die Funktion `do_lookup()` aufruft.

Die Funktion `do_lookup()` erhält als Parameter einen Zeiger auf das aktuelle Verzeichnis (als Dentry-Objekt) `dir` und die Zeichenkette des zu suchenden Pfadelements (`name`). Der Algorithmus zur Suche sieht dann wie folgt aus:[2]

1. Suche das Dentry-Objekt zu dem Pfadelement `name` im Dentry-Cache. Bei Erfolg fahre bei 3., andernfalls bei 2. fort.
2. Rufe die `lookup()` Methode des Inode-Objektes von `dir` mit `dir` und `name` als Parameter auf. Kehrt diese erfolgreich zurück, fahre bei 4. fort, ansonsten kehre mit Fehlermeldung zum Aufrufer zurück.
3. Wurde der Verzeichniseintrag im Dentry-Cache gefunden, muss geprüft werden, ob der gefundene Eintrag (`dentry`) noch gültig ist. Dazu wird die `d_revalidate()` Methode des Dentry-Objektes aufgerufen, sie hat 2 mögliche Rückgabewerte:
   (a) `dentry` ist nicht mehr gültig: Fahre bei 2. fort.
   (b) `dentry` ist noch gültig oder wurde aktualisiert.
4. Vor der erfolgreichen Beendigung von `do_lookup()` wird untersucht, ob auf der gefundenen Inode ein weiteres Dateisystem montiert wurde. Ist dies der Fall wird die Wurzel-Inode des montierten Dateisystems, ansonsten die gefundene Inode an `__link_path_walk()` zurück gegeben.

Die Suche nach Pfadelementen beruht also hauptsächlich auf den Einschubmethoden `lookup()` (definiert im Inode-Objekt) und `d_revalidate()` (definiert

[2] Die Darstellung ist stark vereinfacht und dient in erster Linie dazu den Einsatz der wichtigsten Einschubmethoden darzustellen. Für mehr Details siehe `fs/namei.c` in den Kernel-Quelltexten.

im Dentry-Objekt). Die Methode `d_revalidate()` wurde für Netzwerkdateisysteme eingeführt, weil sich diese Dateisysteme aufgrund von anderen Klienten, die das auf das gleiche Dateisystem zugreifen, verändern können, ohne dass die Änderung durch die lokale VFS-Schicht läuft. Dadurch sind die Einträge im Dentry-Cache, die normalerweise von den entsprechenden Routinen des VFS aktualisiert werden, nicht immer auf dem neusten Stand. Bei nicht-netzwerkbasierten Dateisystemen ist diese Methode normalerweise nicht implementiert, woraufhin `do_lookup()` die Gültigkeit eines im Dentry-Cache gefundenen Eintrags voraussetzt. Es entfällt in dem oben angegebenen Algorithmus der Schritt 3.

Bei jeder Pfadauflösung, das heißt bei der Benutzung von Systemaufrufen der Kategorie 3, erfolgt für jedes Pfadelement eine Suche im Verzeichnis-Cache. Wurde das Element nicht gefunden, erfolgt ein Aufruf von `lookup()`, ansonsten wird die Gültigkeit durch `d_revalidate()` sichergestellt. Damit ist der Aufwand der Prozedur direkt abhängig von der Länge des Pfades, wobei es vorteilhaft ist, wenn der Eintrag bereits im Cache gespeichert und auch noch gültig ist. Im ungünstigsten Fall werden für ein Pfadelement beide Einschubmethoden aufgerufen.

3.6 Zusammenfassung

Das VFS bietet ein Rahmenwerk, das bei der Implementierung eines Dateisystems benutzt werden muss. Es legt das Dateisystem auf die Verwendung des Linux Dateimodells fest, insbesondere auf die Verwendung der vier zentralen Datentypen *Inode*, *File*, *Superblock* und *Dentry*, die aufgrund der in ihnen abgelegten Funktionszeiger als Objekte angesehen werden können.

Die Implementierung eines konkreten Dateisystems erfolgt durch Implementierung dieser Funktionen, welche vom VFS bei Bedarf aufgerufen werden. Nicht implementierte Methoden (der Funktionszeiger wurde auf „Null“ gesetzt) werden im Regelfall durch Standard-Algorithmen aus der VFS-Schicht ersetzt.

Vom VFS werden alle Datei-bezogenen Systemaufrufe bearbeitet, wobei die Schicht Aufgaben übernimmt, die allen Dateisystemen gemeinsam sind. Dazu gehören neben der Verwaltung der zentralen Datentypen auch die Überprüfung der Parameter von Systemaufrufen und schließlich des Aufrufs der korrekten Einschubmethoden eines konkreten Dateisystems. Anhand des Verhältnisses zwischen Systemaufrufen und Einschubmethoden, lassen sich 4 Kategorien von Systemaufrufen unterscheiden, die in Tabelle 3.2 dargestellt sind.

Aufrufe in Kategorie 1 führen zu keinem Aufruf einer Einschubmethode. Bei Aufrufen der Kategorien 2a und 2b erfolgt genau ein Aufruf einer Einschubmethode, wobei sich diese Kategorien darin unterscheiden, dass die Aufrufe in einem Fall auf eine bestimmten Datei gerichtet sind (Kategorie 2b) und im anderen Fall das gesamte Dateisystem betreffen (Kategorie 2a). Systemaufrufe der Kategorie 3 führen zu den meisten Aufrufen von Einschubmethoden, da sie neben der eigentlich Dateisystemoperation auch eine Pfadsuche beinhalten, die zu mehrfachen Aufrufen von Einschubmethoden führen (siehe Abschnitt 3.5).

Die Betrachtung des Verhältnisses zwischen Systemaufrufen und Aufrufen von Einschubmethoden ist wichtig, um die Entwurfsfreiheiten bei der Implementierung von Linux-Dateisystemen zu erkennen und um im Speziellen die Entwurfsentscheidungen von *CLF* nachvollziehen zu können. Im Allgemeinen lässt sich feststellen, dass Systemaufrufe der Kategorie 3, aufgrund der notwen-

digen Validierung in jedem Pfadauflösungsschritt bei Netzwerkdateisystemen besonders aufwändig sind.

Kapitel 4

Vorarbeiten: *Paradis-Net*

Die verschiedenen Komponenten von *CLF* kommunizieren über das Hochgeschwindigkeitsnetzwerk des Clusters miteinander und verwenden dabei komplexe Kommunikationsmuster, die mehr als zwei Teilnehmer haben. Um diese Interaktionen möglichst einfach und elegant implementieren zu können, verwendet *CLF Paradis-Net*, eine Netzwerkschnittstelle, die für diesen Zweck entworfen und implementiert wurde. Dieses Kapitel beschreibt die Bibliothek beginnend bei den Konzepten bis hinunter zu der konkreten Schnittstelle.

4.1 Einleitung

Paradis-Net [54] ist eine Netzwerkbibliothek, die sich besonders für den Einsatz in verteilten Anwendungen und Betriebssystemdiensten eignet. Dabei spielt es keine Rolle, ob die Anwendung über Hochgeschwindigkeitsnetzwerke (wie zum Beispiel Myrinet [6] oder InfiniBand [42]) über oder relativ langsame Transportmedien kommuniziert (wie zum Beispiel dem Internet). Sie abstrahiert von der verwendeten Netzwerkhardware und bietet eine einfache Schnittstelle, welche die Implementierung von komplexen Kommunikationsmustern erleichtert.

Paradis-Net wurde auf Grund der Anforderungen des parallelen Dateisystems *Clusterfile* und der neuen Anforderungen von *CLF* entwickelt und behebt dabei Schwächen, die andere Kommunikationsparadigmen in diesen Anwendungsbereichen zeigen. Auf diese Schwächen sind schon die Entwickler des Dateisystems *xFS* [2] im Jahre 1996 gestoßen, wie sie in einem abschließenden Erfahrungsbericht über die Arbeit an *xFS* schreiben [87]. Sie stellen fest, dass die Hauptursache für ihre Probleme bei der Implementierung die Netzwerkschnittstellen waren, die nicht zu den zu verwirklichenden Diensten passten. Gleichzeitig formulieren sie eine Liste von Eigenschaften, die eine geeignete Netzwerkschnittstelle für *xFS* erfüllen sollte. Der folgende Abschnitt fasst diese Forderungen zusammen und ergänzt sie um die eigenen Erkenntnisse bei der Entwicklung von *CLF*.

4.2 Anforderungen

Die Anforderungen an die Netzwerkschnittstelle, die in diesem Abschnitt abstrakt beschrieben werden, entstammen einerseits der eigenen Erfahrung bei

der Implementierung von *CLF* und andererseits dem Erfahrungsbericht des Entwicklerteams von *xFS* [87].

Obwohl es unzählige Netzwerkschnittstellen für die verschiedensten Netzwerke gibt, steht der Entwickler eines verteilten Betriebssystemdienstes vor einer schwierigen Wahl: Soll für die Anwendung möglichst hohe Performanz erreicht werden, dann muss er eine systemnahe und auf das verwendete Netzwerk spezialisierte Bibliothek verwenden, wodurch die Portabilität der Anwendung auf andere Hardware zumindest gefährdet ist. Soll hohe Portabilität erreicht werden, so muss eine möglichst überall verfügbare Schnittstelle verwendet werden, wodurch die Leistungsfähigkeit leidet: *Die Schnittstelle soll als möglichst dünne Schicht zwischen der Anwendung und einer systemnahen, spezialisierten Bibliothek agieren, um einerseits die gewünschte* Abstraktion *und andererseits auch hohe* Performanz *zu bieten.*

Vor allem für die Implementierung von Dienstgebern ist es von Vorteil, wenn die verwendete Netzwerkschnittstelle Paket-orientiert ist und nicht nur aus einem unstrukturierten Datenstrom besteht. Schnittstellen, die einen Datenstrom zwischen zwei Knoten anbieten, zwingen den Benutzer in den meisten Fällen dazu, ein Protokoll zu implementieren, welches das Versenden von Nachrichten einer bestimmten Länge erlaubt. Diese Aufgabe sollte direkt von der Netzwerkschnittstelle erfüllt werden. Darüber hinaus sollen den einzelnen Nachrichten vom Absender Typen zugeordnet werden können, damit der Empfänger eingehende Nachrichten ohne Betrachtung des Inhalts sortieren kann: *Die Netzwerkschnittstelle soll* paketorientiert *sein und verschiedene* Nachrichtentypen *unterstützen.*

Auf der Empfängerseite sollen die ankommenden Nachrichten asynchron von Behandlungsfunktionen abgearbeitet werden. Die Auswahl der Behandlungsfunktionen soll anhand des Nachrichtentyps erfolgen, so dass diese Vorgehensweise dem Verhalten eines RPC-Aufrufs [58] auf der Empfängerseite ähnelt. Die Verwendung von Behandlungsfunktionen erleichtert insbesondere die Implementierung von Dienstgebern.

Weiterhin ist es bei Dateisystemen unerlässlich, dass die Reihenfolge der Nachrichten zwischen zwei Knoten erhalten bleibt. Erzeugt ein Klient beispielsweise eine Datei, um sie anschließend zu öffnen, dann kann eine Vertauschung dieser beiden Anforderungen zu unvorhergesehenem Verhalten führen. Daher wird auf die Verarbeitung der Nachrichten in parallelen Fäden verzichtet: *Die Netzwerkschnittstelle soll ankommende Nachrichten innerhalb einer* Ereignisschleife *mittels* Behandlungsfunktionen *abarbeiten und dabei die* Reihenfolge *der Nachrichten zwischen Sender und Empfänger aufrecht erhalten.*

Ein Teil des Dateisystems *CLF* sind im Linux-Kern und damit im Betriebssystem verankert. Es handelt sich dabei um das Klienten-Modul, welches das Montieren des Dateisystems auf einem Rechenknoten erlaubt. Dieses Kern-Modul muss sowohl mit dem Metadatenmanager, als auch mit den Daten-Servern kommunizieren. Das setzt voraus, dass die Netzwerkschnittstelle innerhalb des Betriebssystems implementierbar ist und eine Kommunikation mit einer entsprechenden Gegenstelle auf der Benutzerebene erlaubt: *Die Netzwerkschnittstelle soll sich für eine Implementierung innerhalb des* Linux-Kernels *eignen.*

Das größte Problem sehen Wang und Anderson [87] jedoch in fehlender Unterstützung bei der Implementierung komplexer Protokolle, die über das klassische Dienstnehmer/Dienstgeber-Muster hinausgehen. Dabei handelt es sich im

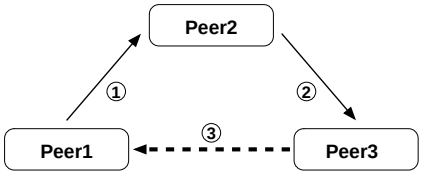


Abbildung 4.1: Weiterleitung

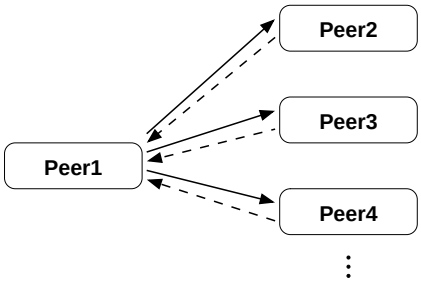


Abbildung 4.2: Verteilen/Sammeln

Durchgezogener Pfeil: Anfrage – Gestrichelter Pfeil: Antwort

Wesentlichen um zwei generische Muster: Die *Weiterleitung* (engl. *Continuation Passing*) und das *Verteilen/Sammeln* (engl. *Scatter/Gather*).

Abbildung 4.1 zeigt das *Weiterleitungsmuster*: Der Knoten *Peer1* stellt eine Anfrage an *Peer2* (①). Da dieser die Anfrage nicht direkt beantworten kann, leitet er sie an einen dritten Knoten weiter (②). Dieser antwortet auf die Anfrage, und schickt die Antwort direkt zu *Peer1* zurück (③). Dieses Muster tritt beispielsweise in xFS [2] beim kooperativen Cachen (siehe Abschnitt 2.2.3) auf: Auf einem Klienten (*Peer1*) wird aus einer Datei gelesen. Da sich die entsprechenden Daten nicht im lokalen Cache befinden, schickt der Klient eine Anfrage an den Manager. Dieser verwaltet nicht nur die Daten auf seiner Festplatte, sondern auch Listen über den aktuellen Cache-Bestand der Klienten und findet heraus, dass ein zweiter Klient (*Peer3*) die gewünschten Daten speichert. Moderne Netzwerke weisen eine niedrige Latenz auf und aus diesem Grund ist es besser, die Daten aus dem Speicher des anderen Klienten zu kopieren, als sie von der Festplatte des Managers zu lesen. Daher wird die Anfrage an *Peer3* weitergeschickt und von diesem beantwortet, indem die gewünschten Daten direkt an den lesenden Klienten geschickt werden.

Das zweite typische Muster für Dateisysteme wie *xFS* und *Clusterfile*, welche die Daten einer Datei über mehrere Daten-Server verteilen, ist das *Verteilen/-Sammeln*-Muster. Es tritt dort auf, wo ein Klient Anfragen an verschiedene andere Knoten schickt und anschließend auf Antworten wartet, die in beliebiger Reihenfolge eintreffen können. Dabei kann es sich beispielsweise um eine Lese-Anfrage an verschiedene Daten-Server handeln, die jeweils nur einen Teil der betreffenden Datei speichern. Abbildung 4.2 illustriert das Protokoll: Die Anfragen werden gleichzeitig losgeschickt (wenn möglich) und die Interaktion ist erst dann beendet, wenn jeder angesprochene Knoten eine passende Antwort zurückgeschickt hat.

Bisherige Netzwerkschnittstellen bieten dem Programmierer nur wenig Unterstützung bei der Implementierung solcher Kommunikationsmuster. *Das Ziel beim Entwurf von* Paradis-Net *war es daher, dass die Netzwerkschnittstelle dem Benutzer die effiziente Implementierung von Protokollen, wie dem* Dienstnehmer/ Dienstgeber- *(Client/Server), dem* Weiterleitungs- *(Continuation Passing) und dem* Verteilen/Sammeln-Muster *(Scatter/Gather), erleichtert.*

4.3 Verwandte Arbeiten

In diesem Abschnitt werden verschiedene Netzwerkschnittstellen vorgestellt und an den Forderungen des letzten Abschnitts gemessen. Die Auswahl orientiert sich an dem, was in anderen Dateisystemen zum Einsatz kommt.

4.3.1 TCP-Sockets

Das *Transmission Control Protocol* (TCP) ist ein zuverlässiges, verbindungsorientiertes Transportprotokoll für Computernetze, das 1981 als RFC 793 [28] standardisiert wurde. TCP ist Teil der Internetprotokollfamilie und stellt einen virtuellen Kanal zwischen zwei Endpunkten (Sockets) her. Auf diesem Kanal können in beide Richtungen Daten übertragen werden. TCP-Sockets sind unter Linux/UNIX Teil des Betriebssystems und werden über standardisierte Schnittstellen (GLIBC [30]) angesprochen. Trotz des schwergewichtigen Protokolls, werden sie aufgrund ihrer breiten Verfügbarkeit von vielen Dateisystemen, wie zum Beispiel NFS (ab Version 3 [13]), AFS [41], PVFS [17] und von *Clusterfile* [44] benutzt. Neuere Entwicklungen erlauben es, TCP-Sockets auch über Hochgeschwindigkeitsnetzwerke zu verwenden, die mit Hilfe von verschiedenen Optimierungen die Latenz auf die Größenordnung der proprietären Schnittstellen senken können. So erlaubt beispielsweise *Sockets-GM* [32] die Kommunikation über das Myrinet-Netzwerk mittels Sockets, bei InfiniBand heißt die entsprechende Schnittstelle *Sockets Direct Protocol (SDP)* [66].

TCP-Sockets arbeiten aus Anwendungssicht auf der Basis von Byteströmen. Anwendungen müssen daher in vielen Fällen den unstrukturierten Datenstrom mit eigenen Protokollen zu einem Paket-orientierten Dienst erweitern. Dies ist bei der Implementierung von Netzwerkdateisystemen unerlässlich, da diese anfrageorientiert und mit variierenden Nachrichtengrößen arbeiten. Der Empfänger einer Nachricht muss in der Lage sein, einzelne Nachrichten (Anfragen oder auch Antworten) aus dem Bytestrom zu extrahieren und weiter zu verarbeiten. Bei TCP-Sockets muss diese Arbeit von der Anwendung übernommen werden. Weiterhin muss vor der eigentlichen Kommunikation zunächst einmal die virtuelle Verbindung zu dem Kommunikationspartner aufgebaut werden. Bei dem Verbindungsaufbau müssen beide Seiten aktiv werden: Eine Seite muss die Verbindung initiieren und die andere Seite muss sie akzeptieren. Über jede solche Verbindung kann der Knoten mit genau einem Partner kommunizieren.

Alternativ zur Strukturierung des Datenstromes kann eine Anwendung auch für jede Anfrage eine neue Verbindung aufbauen und nach Empfang der Antwort wieder schließen. Nachteil dieser Vorgehensweise ist vor allem die große Verzögerung, die dadurch verursacht wird. Beim Aufbau einer TCP-Verbindung treten beide Seiten miteinander in Kontakt, bevor überhaupt Daten ausgetauscht werden können. Darüber hinaus würde der Empfang einer Antwort von einem anderen Dienstgeber im Rahmen des Weiterleitungs-Musters den Aufbau einer weiteren Verbindungen erfordern, über welche die Nachricht beim Klienten eintreffen würde. Der Klient müsste die Antwort aus der neuen Verbindung empfangen und mit der ursprünglichen Anfrage in Verbindung setzen. Bei diesen Aufgaben bieten TCP-Sockets keinerlei Unterstützung.

Aber selbst wenn man davon ausgeht, dass zwischen allen beteiligten Knoten offene Verbindungen bestehen, helfen TCP-Sockets nicht bei der Implementierung des *Weiterleitungs-* oder des *Verteilen/Sammeln-Musters*, sie erschweren

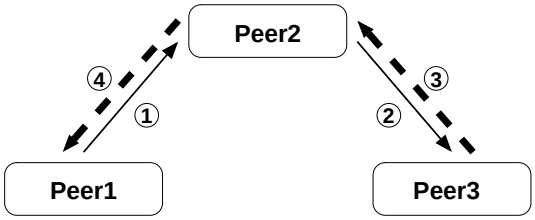


Abbildung 4.3: Weiterleitung per RPC – Variante 1

Durchgezogener Pfeil: Anfrage – Gestrichelter Pfeil: Antwort

es aufgrund der Endpunkt-Semantik sogar. Beim *Weiterleitungsmuster* ist im Vorherein nicht bekannt, von welchem Knoten die Antwort gesendet wird. Daher muss die Anwendung alle offenen Endpunkte auf eventuelle Antworten hin überwachen und die Antwort aus diesen in Empfang zu nehmen. TCP-Sockets arbeiten für die Entwicklung von verteilter Software nicht auf der richtigen Abstraktionsebene.

4.3.2 RPC

Entfernte Prozeduraufrufe (*Remote Procedure Call*, RPC [58]) werden in verteilten Dienstnehmer/Dienstgeber Anwendungen eingesetzt und kommen auch in dem Netzwerkdateisystem *NFS* (bis Version 3 [70]) zum Einsatz. Ihr großer Vorteil liegt in der einfachen Semantik, die einem lokalem Funktionsaufruf ähnelt. Die Grundidee beim Entwurf von RPCs war es, den Entwickler von den unnötigen Komplikationen des Netzwerks fern zu halten, damit er sich auf die Probleme höherer Ebene konzentrieren kann. Dies funktioniert auch recht gut, solange es sich um ein reines Dienstnehmer/Dienstgeber System handelt. Wie bereits im vorangegangenen Abschnitt erläutert, sind für bestimmte Anwendungen nicht nur solche einfachen Kommunikationsmuster notwendig, sondern auch Muster, die eine Interaktion von mehreren Knoten in verschiedenen Rollen erfordern.

Versucht man solche Muster mittels entfernter Prozeduraufrufe zu synthetisieren, so führt diese zu unnatürlichen Konstruktionen, die darauf zurückzuführen sind, dass ein entfernter Aufruf immer aus einem Anfrage-Antwort Paar besteht. Abbildung 4.3 zeigt den Versuch einer Implementierung des *Weiterleitungsmusters* (Abbildung 4.1) mittels RPCs. Die durchgezogenen, dünnen Pfeile stellen dabei die (entfernten) Prozeduraufrufe und die gestrichelten Pfeile die Antworten dar. Die Dicke der Pfeile soll die Größe der Nachricht wiederspiegeln. *Peer1* stellt zunächst die Anfrage an *Peer2*, der wiederum einen entfernten Aufruf an *Peer3* richtet. Aufgrund der Semantik von entfernten Aufrufen muss die Antwort dem Weg der Anfragen zurück zum ursprünglichen Aufrufer folgen. Dabei kommt es zu dem im Grunde unnötigen Datentransfer über Knoten 2, der zusätzlich Speicher zum Zwischenspeichern der Antwort von Knoten 3 zur Verfügung stellen muss.

Eine Möglichkeit den zusätzlichen Transfer zu vermeiden zeigt Abbildung 4.4: Hier findet der eigentliche Datentransfer während eines zusätzlichen entfernten Aufrufs statt und nicht während der Rückkehr. Dabei müssen alle beteiligten

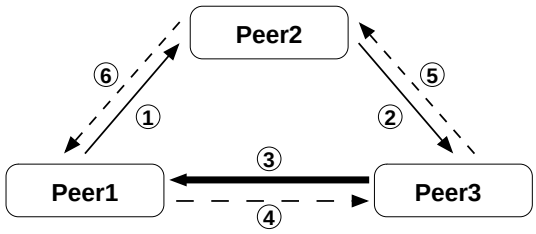


Abbildung 4.4: Weiterleitung per RPC – Variante 2

Durchgezogener Pfeil: Anfrage – Gestrichelter Pfeil: Antwort

Knoten sowohl als Dienstgeber, wie auch als Dienstnehmer auftreten. Für Knoten 1 besteht eine weitere Schwierigkeit darin, den entfernten Aufruf von Knoten 3 (③), mit der eigenen Anfrage (①) an Knoten 2 in Verbindung zu setzen. Vor dem Absetzen des entfernten Aufrufs muss der Knoten 1 dafür zunächst lokal Informationen ablegen, die es ihm erlauben, die spätere Anfrage eines anderen Knoten als Antwort auf die vorausgegangene, eigenen Anfrage zu erkennen. Neben diesen Komplikationen wird durch die Auflösung der entfernten Aufrufe (Schritte ④, ⑤ und ⑥) noch weitere Verzögerung verursacht, da Knoten 1 durch die RPC Semantik bis zum Abschluss seines entfernten Aufrufs blockiert ist.

Alleine an der Anzahl der notwendigen Netzwerknachrichten lässt sich schon die Ineffizienz von RPCs bei der Implementierung dieses Musters ablesen. Unter der Annahme, dass die Nachricht nicht nur einmal, sondern n Mal weitergeleitet wird, sind im optimalen Fall (Abbildung 4.1) $n + 1$ Nachrichten notwendig, während mit entfernten Prozeduraufrufen entweder $2n$ (Abbildung 4.3) oder $2n + 2$ (Abbildung 4.4) Nachrichten verschickt werden müssen.

Für die Implementierung des *Verteilen/Sammeln-Musters* (Abbildung 4.2) scheinen RPCs zunächst besser geeignet: *Peer1* stellt nacheinander die Anfragen an *Peer2*, *Peer3*,... und erhält jeweils eine Antwort zurück. Bei genauerer Betrachtung wird jedoch klar, dass aufgrund der blockierenden Semantik von *entfernten Prozeduraufrufen* nur eine sehr ineffiziente Implementierung möglich ist. Für dieses Muster sollten alle Anfragen, unabhängig vom Eintreffen der Antworten, innerhalb kürzester Zeit abgeschickt werden, damit die Dienstgeber diese parallel beantworten können und um die Verzögerungen durch den Netzwerktransfer zu verstecken. Bei der Verwendung von RPCs kann die nächste Anfrage jedoch erst dann losgeschickt werden, wenn die Antwort auf die letzte Anfrage eingetroffen ist. Durch die Serialisierung der Anfragen geht theoretisch mögliche Parallelität verloren.

Insgesamt besitzen *Entfernte Prozeduraufrufe* zwar eine relativ hohe Abstraktionsebene, sind aber für die Implementierung der komplexen Kommunikationmuster nur schlecht geeignet.

4.3.3 Active Messages

Aktive Nachrichten (*Active Messages*, AM [53, 86]) ist eine systemnahe Kommunikationsschnittstelle, die als eine Art Maschinensprache für Entwickler von

Kommunikationsbibliotheken verstanden werden kann. Die Schnittstelle kam, hauptsächlich aus Performanz-Gründen, bei der Implementierung des Dateisystems *xFS* zum Einsatz.

Die Grundidee der Aktiven Nachrichten ist, dass eine Nachricht die Kennung einer Prozedur mit sich führt, die für ihre Weiterverarbeitung auf der Empfängerseite zuständig ist. Dies ähnelt den entfernten Funktionsaufrufen, mit dem Unterschied, dass die Behandlungsfunktionen auf Empfängerseite nicht für die Beantwortung der Anfrage, sondern nur für das Empfangen der Nachricht zuständig sind.

Die Nachrichten werden zuverlässig ausgeliefert, aber möglicherweise nicht in der richtigen Reihenfolge. Wenn eine Nachricht eintrifft, wird die Behandlungsfunktion aufgerufen und läuft bis zu ihrer Beendigung. Dieser Aufruf ist einer der Gründe für die Effizienz von AM, führt aber gleichzeitig zu Einschränkungen im Programmiermodell. Da während der Laufzeit der Routine keine weiteren Nachrichten empfangen werden, muss sie schnell und blockierungsfrei ablaufen. Daher dürfen Behandlungsfunktionen nicht die Bearbeitung der Anfrage übernehmen, sondern nur die Nachricht in Empfang nehmen und die Anwendung informieren. Die Anwendung übernimmt anschließend in einem anderen Ausführungsfaden die Beantwortung und schickt die Antwort an den Klienten. Bei einigen Implementierungen darf die Behandlungsfunktion einer Anfrage genau eine Nachricht an den Absender der Anfrage senden. Um Verklemmungen zu vermeiden, darf die Behandlungsfunktion der daraufhin abgesetzten Antwort das Netzwerk überhaupt nicht benutzen. So können mit Hilfe der Behandlungsfunktionen einfache Anfrage/Antwort-Muster implementiert werden. Die in Abschnitt 4.2 beschriebenem Kommunikationsmuster sind mit AM zwar implementierbar, erfordern aber relativ aufwändige Programmierung. Die Schnittstelle bietet auch nur schwache Unterstützung für Dienstnehmer/Dienstgeber Muster.

Obwohl AM bei der Implementierung von *xFS* zum Einsatz kam, sprechen sich die Entwickler rückblickend jedoch ausdrücklich gegen den Einsatz der Schnittstelle in verteilten Systemen aus [87]. Die Hauptgründe, die gegen den Einsatz von AM sprechen sind die niedrige Schnittstellenebene, die viele systemnahe Details entblößt und die Beschränkung auf ein Dienstnehmer/Dienstgeber Kommunikationsmodell.

4.3.4 MPI

Das *Message Passing Interface* (MPI [33, 83]) ist ein Schnittstellenstandard, der durch das *MPI Forum*, einem Konsortium von Computerherstellern, Naturwissenschaftlern und Entwicklern von Bibliotheken, im Jahr 1993 festgelegt wurde. MPI hat sich in den letzten Jahren zu einem de-facto Standard bei der Programmierung von parallelen Anwendungen für Cluster nach dem SPMD (Single Program, Multiple Data) Prinzip entwickelt. MPI unterstützt sowohl Punkt-zu-Punkt Nachrichten, wie auch gemeinsame Operationen von mehreren Teilnehmern. MPI wird von festen Gruppen von Prozessen benutzt, die gemeinsam in Kommunikatoren organisiert sind, welche den Kontext der MPI Operationen festlegen. Innerhalb eines Kommunikators werden die teilnehmenden Prozesse anhand einer eindeutigen Nummer, dem so genannten *Rang* unterschieden und adressiert.

Zu den globalen Operationen gehören auch Verteil-Operationen, die Daten,

wie beispielsweise Matrizen, nach bestimmten Mustern an eine Reihe von Prozessen versenden können. Dazu wird auf jedem der beteiligten Knoten der gleiche Befehl ausgeführt, der anhand des Ranges die Rolle des aufrufenden Prozesses innerhalb der globalen Operation erkennt. Weiterhin gibt es auch Sammel-Operationen, die analog zu den Verteil-Operationen funktionieren.

Aus verschiedenen Gründen ist MPI jedoch nicht für den Einsatz in verteilten Dateisystemen geeignet. Zunächst einmal bestehen Netzwerkdateisysteme aus verschiedenen unabhängigen Diensten, die unabhängig voneinander gestartet und beendet werden und nicht aus einem einzelnen Programm, das auf allen teilnehmenden Knoten gemeinsam gestartet wird. In einer MPI-Laufzeitumgebung ist das Hinzukommen von Kommunikationsteilnehmern nach dem Start, sowie das Wegfallen von Kommunikationspartnern zur Laufzeit nicht vorgesehen. Nach dem Start aller MPI-Prozesse registrieren sich diese bei der Laufzeitumgebung, welche die Kommunikatoren bildet; ein nachträgliches Hinzukommen von Prozessen ist nicht vorgesehen. Es gilt als Fehler, wenn sich ein Kommunikationspartner vorzeitig beendet, daher werden auch die übrigen Prozesse automatisch von der Laufzeitumgebung beendet.

Insgesamt gesehen ist das Kommunikationsmodell von MPI nicht für die Implementierung von verteilten Dateisystemen geeignet.

4.4 Das Kommunikationsmodell von *Paradis-Net*

Aufgrund der speziellen Anforderungen (siehe Abschnitt 4.2) wurde für das Clusterdateisystems *CLF* eine neuartige Kommunikationsschnittstelle entwickelt. In diesem Abschnitt soll zunächst das Kommunikationsmodell erläutert werden, das der im folgenden Abschnitt beschriebenen, konkreten Programmierschnittstelle zugrunde liegt.

Die Bibliothek *Paradis-Net* arbeitet mit einem verbindungslosen Peer-to-Peer Kommunikationsmodell, bei dem jeder Kommunikationsendpunkt gleichzeitig Dienstgeber und Dienstnehmer sein kann. Für jeden Kommunikationsendpunkt wird eine Adresse vergeben, die innerhalb des verwendeten Netzwerks eindeutig ist. Eine Instanz von *Paradis-Net* kann auch mehrere Endpunkte definieren, falls der Knoten über unterschiedlich Netzwerke erreichbar ist.

Ein Beispiel dafür ist ein Cluster, der über ein normales und ein Hochgeschwindigkeits-Netzwerk verfügt, wie Ethernet und Infiniband. Mit *Paradis-Net* kann dann ein Programm (= eine *Paradis-Net*-Instanz) jeweils einen Endpunkt für Ethernet und einen für Infiniband registrieren. Anschließend wird das Programm unter einem Endpunktnamen wie „tcp:...“ für Ethernet und „verbs:...“ für Infiniband erreichbar sein.

4.4.1 Nachrichten

Zwischen den Endpunkten können Nachrichten beliebiger Größe ausgetauscht werden. Eine Nachricht besteht aus folgenden Elementen:

Typ: Eine Nachricht besitzt einen Typ. Nachrichtentypen müssen zwischen den verschiedenen Kommunikationsteilnehmern abgesprochen werden. Der Empfänger einer Nachricht entscheidet anhand des Typs der Nachricht, wie diese zu behandeln ist.

Daten: Der eigentliche Inhalt der Nachricht besteht aus unstrukturierten Daten einer bestimmten Länge, die der Empfänger anhand des Nachrichtentyps interpretieren kann.

(optional) Kooperationsmarke: Die Kooperationsmarke spielt eine besondere Rolle bei dem Entwurf von Netzwerkprotokollen. Mehr dazu in Abschnitt 4.4.3.

Nachrichten werden durch Aufruf der Funktion `send` an einen bestimmten Endpunkt gesendet. Kehrt die Funktion ohne Fehler zurück, garantiert *Paradis-Net*, dass sie beim Empfänger eingetroffen ist oder eintreffen wird; der lokale Speicherbereich, der die Nachricht enthält, kann weiterverwendet werden. Werden nacheinander mehrere Nachrichten mit *Paradis-Net* an den gleichen Kommunikationspartner verschickt, garantiert *Paradis-Net*, dass diese in der gleiche Reihenfolge dort eintreffen, in der sie abgeschickt wurden.

4.4.2 Behandlungsfunktionen (Dienstgeber)

In *Paradis-Net* gibt es nur eine Sendeoperation und keine Empfangsoperation. Nachrichten werden automatisch empfangen und an Behandlungsfunktionen weitergeleitet. Die Behandlungsfunktionen werden durch die Anwendung bei *Paradis-Net* registriert. Bei der Registrierung wird angegeben, für welchen Nachrichtentyp eine Behandlungfunktion zuständig ist. Eine Behandlungsfunktion kann auch für mehrere Nachrichtentypen registriert werden; weiterhin gibt es auch die Möglichkeit, eine Standard-Behandlungsfunktion fest zu legen. Diese Funktion wird immer dann aufgerufen, wenn für den Typ einer eingetroffenen Nachricht keine spezielle Behandlungsfunktion registriert wurde.

Trifft eine Nachricht auf einem Endpunkt ein, wird Speicher für die Nachricht angefordert, der nach Beendigung der Behandlungsfunktion wieder frei gegeben wird.[1] Alle eintreffenden Nachrichten werden von *Paradis-Net* serialisiert und die Behandlungsfunktionen werden nacheinander aufgerufen. Auf eine parallele Verarbeitung den Anfragen in mehreren Ausführungsfäden wird verzichtet, um zu vermeiden, dass voneinander abhängige Anfragen gleichzeitig verarbeitet werden. Die registrierte Behandlungsfunktion mit folgenden Parameterm aufgerufen: *Nachrichtentyp*, *Absender*, *Nachricht* und *Kooperationsmarke*.

Behandlungsfunktionen erleichtern die Implementierung von Dienstgebern: Der Typ einer Anfrage entscheidet darüber, welche Behandlungsfunktion aufgerufen wird, um die Anfrage zu bearbeiten und per `send` an den Dienstnehmer zurück zu senden.

Eine vergleichbare Technik liegt auch der Implementierung von entfernten Funktionsaufrufen auf der Dienstgeberseite zugrunde. Der Nachrichtentyp übernimmt bei *Paradis-Net* die Aufgabe des Funktionsnamens bei RPCs, indem er bestimmt, welche Funktion aufgerufen wird. Der Inhalt einer *Paradis-Net*-Anfrage entspricht den Parametern einer entfernten Funktion, während der Inhalt einer von der Behandlungsfunktion gesendeten Antwort dem Rückgabewert eines entfernten Funktionsaufrufs entspricht.

[1] Optional verzichtet *Paradis-Net* auch auf das Freigeben des Speichers, er muss dann von der Anwendung frei gegeben werden.

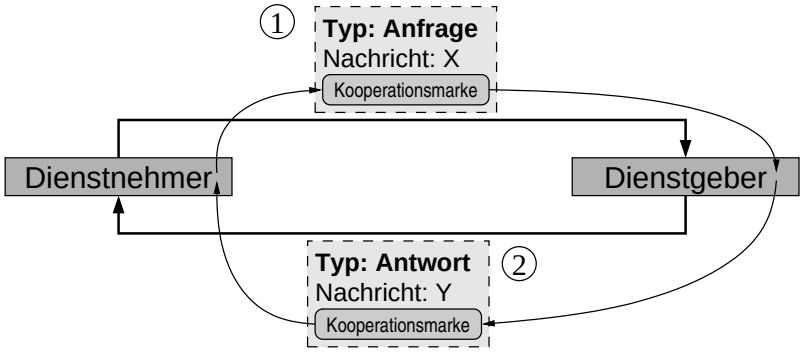


Abbildung 4.5: Der Weg einer Kooperationsmarke bei einer Anfrage

4.4.3 Kooperationen (Dienstnehmer)

Während Behandlungsfunktionen die Implementierung der Dienstgeberfunktionalität erleichtern, erschweren sie die Implementierung des Dienstnehmers. Der Grund dafür liegt in der asynchronen Behandlung der ankommenden Nachrichten in einem eigenen Ablauffaden, der von *Paradis-Net* gestartet wird.

In der Regel können Dienstnehmer nach dem Absetzen einer Anfrage ohne die Antwort des Dienstgebers nicht unbegrenzt weiterarbeiten. Zu einem bestimmten Zeitpunkt muss die Ausführung gestoppt und auf das Eintreffen der Antwort gewartet werden. In *Paradis-Net* existiert jedoch kein Befehl zum Empfangen einer Nachricht, da einkommende Nachrichten automatisch empfangen und an Behandlungsfunktionen weitergeleitet werden. Um dieses Problem mit den bisher vorgestellten Mitteln zu lösen, ist ein erheblicher Aufwand notwendig: Ein Ausführungsfaden, der eine Anfrage stellt, müsste in einem gemeinsamen Speicherbereich einen Eintrag vornehmen, der von einer Behandlungsfunktion ausgelesen werden muss, um auf diese Weise eine Verbindung zwischen beiden Ausführungskontexten herzustellen. Dies müsste von dem Benutzer unter Zuhilfenahme von Techniken wie Monitoren und Signalen implementiert werden.

4.4.3.1 Einfache Kooperationen

Um die Implementierung von Dienstnehmerfunktionalität zu erleichtern wurde in *Paradis-Net* das Konzept der *Kooperation* eingeführt. Mit Hilfe einer *Kooperation* kann eine Anwendung definieren, dass sie auf eine oder mehrere Antworten wartet, die in Zukunft eintreffen werden.

Eine *Kooperation* wird durch einen *Paradis-Net*-Aufruf erstellt, bevor die Anfrage an den Dienstgeber geschickt wird; als Ergebnis des Aufrufs erhält die Anwendung eine so genannte *Kooperationsmarke*. Die *Kooperationsmarke* repräsentiert eine einzelne Kooperation. Sie wird an die Anfrage angehängt (die `send`-Funktion besitzt einen entsprechenden Parameter) und der Dienstgeber hängt sie wiederum an seine Antwort. Damit ermöglicht er es dem Dienstnehmer den Zusammenhang zwischen der zuvor gestellten Anfrage und der eintreffenden Antwort herzustellen.

Um auf das Eintreffen der Antwort zu warten, ruft die Anwendung auf dem Dienstnehmer eine *Paradis-Net*-Funktion zum „Beenden“ der *Kooperation* auf

und übergibt ihr die *Kooperationsmarke*. Diese Funktion blockiert so lange, bis die Antwort des Dienstgebers (genauer: ... bis eine Antwort, der die gleiche *Kooperationsmarke* angehängt wurde ...) eingetroffen ist, oder wahlweise, bis eine Zeitschranke überschritten wurde. Weiterhin bietet *Paradis-Net* auch eine Funktion an, die es ermöglicht den Zustand einer *Kooperation* nicht-blockierend abzufragen.

Abbildung 4.5 stellt den Weg einer *Kooperationsmarke* bei einer einfachen Anfrage an einen Dienstgeber dar. Anfragen und Antworten müssen bestimmte Typen haben. Der Typ der Anfrage muss beim Dienstgeber mit der entsprechenden Behandlungsfunktion assoziiert sein und der Typ der Antwort muss beim Dienstnehmer als *Kooperationstyp* bei *Paradis-Net* registriert werden. Ein Nachrichtentyp, der als *Kooperationstyp* registriert ist, führt nicht zum Aufruf einer Behandlungsfunktion, sondern veranlasst *Paradis-Net* dazu die eintreffende Nachricht anhand ihrer *Kooperationsmarke* einer offenen *Kooperation* zuzuordnen und gemäß der Definition der *Kooperation* zu empfangen. Auf diese Weise ist es beispielsweise möglich, eine Antwort in einem von der Anwendung zur Verfügung gestellten Speicherbereich zu empfangen (siehe Abschnitt 4.4.3.4).

Das in der Abbildung dargestellt Kommunikationsmuster ähnelt dem der *entfernten Prozeduraufrufe*, im Gegensatz zu diesen ist der „Funktionsaufruf" (d.h. das Absenden der Anfrage auf dem Dienstnehmer) jedoch nicht blockierend. Nachdem die Anfrage ① gesendet wurde und bevor die Antwort ② eintrifft, kann der Dienstnehmer noch beliebige weitere Operationen ausführen. Insbesondere ist es auch möglich gleichzeitig mehrere Kooperationen, auch aus mehreren Ausführungsfäden heraus, zu starten und diese in beliebiger Reihenfolge zu beenden.

4.4.3.2 Kooperationen: Weiterleiten

Ein wichtiger Unterschied zwischen *Kooperationen* und *entfernten Prozeduraufrufen* besteht darin, dass die Antwort des Dienstgebers nicht anhand ihrer Herkunft, sondern anhand der mitgeführten *Kooperationsmarke* erkannt wird. Daher ist es möglich, dass auch andere Kommunikationsendpunkte die Antwort an den Dienstnehmer schicken und nicht nur der Endpunkt, an den ursprünglich die Anfrage gestellt wurde.

Mit *Kooperationen* lässt sich daher das *Weiterleitungsmuster* (siehe Abschnitt 4.2) leicht implementieren. Abbildung 4.6 zeigt den Weg des *Kooperationstokens* bei der Weiterleitung der Anfrage von Dienstgeber1 auf Dienstgeber2. Dieses Szenario unterscheidet sich aus der Sicht des Anwendungsprogramms des Dienstnehmers nicht von der einfachen Anfrage aus Abbildung 4.5.

Um die Weiterleitung der Anfrage von Dienstgeber1 auf Dienstgeber2 zu erleichtern, gibt es ein spezielles *Paradis-Net*-Kommando, das nur im Kontext einer Behandlungsfunktion aufgerufen werden darf: `forward`. Das Kommando sendet, wie `send`, eine Nachricht an einen Kommunikationsendpunkt, unterscheidet sich aber von `send` dadurch, dass auf dem Zielrechner der tatsächliche Absender durch den ursprünglichen Auftraggeber ersetzt wird. Auf das Beispiel in Abbildung 4.6 bezogen, bedeutet das: Dienstgeber1 verwendet in der Behandlungsfunktion, in der die Nachricht ① verarbeitet wird, das `forward`-Kommando, um die vom Dienstnehmer stammende Nachricht an Dienstgeber2 weiterzuleiten.

Durch Verwendung des `forward`-Kommandos, statt des `send`-Kommandos,

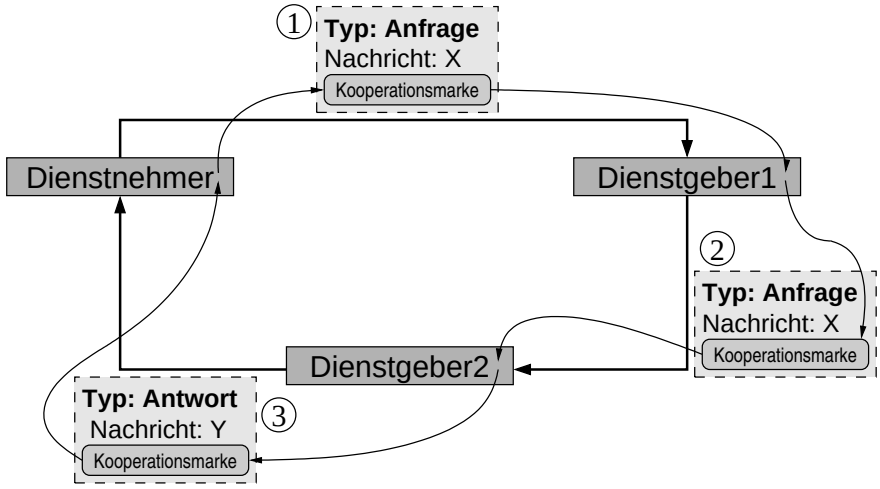


Abbildung 4.6: Der Weg einer Kooperationsmarke bei einer Anfrage mit Weiterleitung

wird beim Aufruf der Behandlungsfunktion durch *Paradis-Net* nicht der wirkliche Absender (Dienstgeber1), sondern der ursprüngliche Absender der Nachricht (Dienstnehmer) übergeben. Für die Behandlungsfunktion auf Dienstgeber2 erscheint es so, als ob die Nachricht direkt von dem Dienstnehmer geschickt wurde. Der Vorteil daran ist, dass bei der Implementierung der Behandlungsfunktion keine Fallunterscheidung zwischen den direkt von einem Dienstnehmer gesendeten und den weitergeleiteten Nachrichten vorgenommen werden muss. Weiterhin muss die Adresse des Dienstnehmers nicht in die Nachricht geschrieben werden.

Auf dem Dienstnehmer wird schließlich die Nachricht ③ empfangen und anhand der angehängten *Kooperationsmarke* als Antwort auf die Anfrage ① erkannt. Wie bereits erwähnt, unterscheidet sich daher die Anwendung auf den Dienstnehmer in diesem Fall nicht von dem in Abbildung 4.5 dargestellten Kommuniktionsfall.

4.4.3.3 Kooperationen: Verteilen/Sammeln

Neben der *Weiterleitung* lässt sich auch das in Abschnitt 4.2 beschriebene *Verteilen/Sammeln*)-Muster in *Paradis-Net* mit Hilfe von Kooperationen implementieren: Eine *Kooperation* beschreibt die Antwort, die ein Dienstnehmer auf eine Anfrage von einem Kooperationspartner erwartet. Im Allgemeinen kann diese Antwort auch aus mehreren Nachrichten bestehen und das Resultat mehrerer Anfragen sein.

Ein Dienstnehmer kann eine *Kooperation* starten, die mehrere Antwort-Nachrichten erwartet und anschließend die gleiche Anzahl Anfragen an unterschiedliche Dienstgeber stellen. Sobald die *Kooperation* gestartet wurde, ist *Paradis-Net* in der Lage die Antworten der Dienstgeber zu empfangen, da das Empfangen der Nachrichten in einem separaten Ausführungsfaden vorgenommen wird. Nach dem Starten der Kooperation beginnt die Anwendung die Anfragen an die Dienstgeber nacheinander zu verschicken, anschließend wartet sie

auf die Beendigung der *Kooperation*, d.h. auf das Eintreffen aller Antworten, weil eine *Kooperation* erst dann als beendet gilt, wenn alle erwarteten Nachrichten eingetroffen sind.

Es ist sogar denkbar das *Verteilen/Sammeln*- und das *Weiterleitungs*-Muster miteinander zu verknüpfen. Das bedeutet, dass die einzelnen Dienstgeber nicht direkt auf die Anfragen antworten, sondern diese an andere Dienstgeber weiterleiten. Aufgrund der mitgeführten *Kooperationsmarken* kann *Paradis-Net* auf dem Dienstnehmer die Antworten der richtigen *Kooperation* zuordnen.

4.4.3.4 Kooperationen: Speicher angeben

Beim Eintreffen einer Nachricht an Kommunikationsendpunkt reserviert *Paradis-Net* automatisch ausreichend Speicher zum Ablegen der eintreffenden Nachricht. Bei Nachrichten mit *Kooperationstyp*, die nicht zum Aufruf einer Behandlungsfunktion führen, kann eine Anwendung dies aber umgehen, indem sie selbst den Speicher zur Verfügung stellt und beim Erstellen der *Kooperation* angibt. Trifft die Antwort-Nachricht mit der passenden *Kooperationsmarke* auf dem Dienstgeber ein, wird sie direkt in den von der Anwendung dafür vorgesehen Speicherbereich geschrieben.

Beim Lesen von Daten von einem Dienstgeber kann diese Möglichkeit dazu verwendet werden, das Kopieren der angeforderten Daten aus dem von *Paradis-Net* automatisch angelegten Speicher in der Anwendung zu vermeiden.

4.4.4 Zusammenfassung

Die Bibliothek *Paradis-Net* bietet einer Anwendung typisierten Nachrichtenaustausch und kombiniert Unterstützung für Dienstgeber- und Dienstnehmer-Rollen. Für die Implementierung von Dienstgeberfunktionalität können Behandlungsfunktionen verwendet werden, die von *Paradis-Net* in Abhängigkeit vom Typ einer eintreffenden Nachricht aufgerufen werden. Bei Implementierung einer Anfrage an einen Dienstgeber kommen *Kooperationen* zum Einsatz, die darüber hinaus auch die Realisierung der zuvor vorgestellten Kommunikationsmuster *Weiterleitung* und *Verteilen/Sammeln* unterstützen.

4.5 Der Aufbau von *Paradis-Net*

In diesem Abschnitt wird die konkrete Schnittstelle und Architektur der *Paradis-Net*-Bibliothek beschrieben, welche die in Abschnitt 4.4 vorgestellten Konzepte umsetzt.

Die *Paradis-Net* Bibliothek bildet eine dünne Schicht zwischen der Anwendung und den jeweiligen nativen Schnittstellen (Abbildung 4.7). Aus Sicht der Anwendung handelt es sich um eine Schnittstelle, die unabhängig von der verwendeten Netzwerktechnologie ist; die Netzwerke werden anhand von eindeutigen Zeichenketten-Namen unterschieden.

4.5.1 Implementierung

Die Bibliothek ist in der Programmiersprache C und unter Verwendung der POSIX Threads Bibliothek (*PThreads*, [12]) implementiert. Alle *Paradis-Net*-

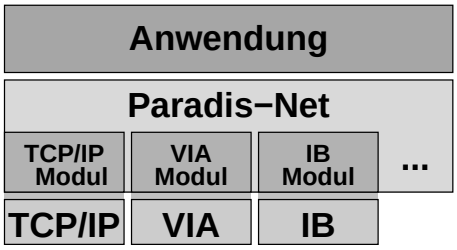


Abbildung 4.7: Die Architektur von *Paradis-Net*

Funktionen wurden Thread-sicher entworfen und können daher aus unterschiedlichen Kontexten gleichzeitig aufgerufen wurden.

Bei zeitkritischen Funktionen, wie beispielsweise dem Senden von Daten werden die kritischen Datenstrukturen fein-granular geschützt und damit das gleichzeitige Senden an unterschiedliche Kommunikationpartner ermöglicht. Wird aus unterschiedlichen Ausführungskontexten die Sende-Operation mit dem gleichen Kommunikationspartner aufgerufen, serialisiert *Paradis-Net* die Nachrichten. Der Schutz der kritischen Bereiche ist mit den Mutex-Strukturen der POSIX Threads-Bibliothek realisiert[2]. Die Bibliothek startet bei ihrer Initialisierung einen Ausführungsfaden, den *Empfands-Faden*, der während der gesamten Laufzeit der Bibliothek aktiv ist. Dieser Faden ist einerseits für den Empfang von eingehenden Nachrichten verantwortlich und andererseits für die Verwaltung der Verbindungen zwischen den Kommunikationsteilnehmern. Im Falle von TCP/IP akzeptiert er beispielsweise neu aufgebauten Verbindungen, die von der Gegenseite durch die Sende-Operation aufgebaut werden.

Der Empfangs-Faden empfängt die eingehenden Nachrichten nicht nur, sondern verarbeitet sie auch entsprechend des Nachrichtentyps: Handelt es sich bei der eintreffenden Nachricht um eine Antwort, die Teil einer Kooperation ist, wird sie mittels der Kooperationsmarke identifiziert und entsprechend der Kooperationsbeschreibung empfangen. Anschließend werden eventuell andere Ausführungsfäden benachrichtigt, die auf die Beendigung der Kooperation warten. Die Kooperationslogik gehört zu dem komplexesten Teilen von *Paradis-Net* und wird vollständig von dem Empfangs-Faden durchgeführt. Dazu gehört auch das eventuelle Anfordern von zusätzlichem Speicher und die Verteilung einer eingetroffenen Nachricht über mehrere zuvor vom Benutzer angelegten Speicherbereiche.

Ist eine eingehende Nachricht nicht Teil einer Kooperation, wird sie in speziell angefordertem Speicher empfangen und die vom Benutzer festgelegte Behandlungsfunktion aufgerufen. Diese Behandlungsfunktion wird jedoch nicht im Ausführungskontext des Empfangs-Fadens aufgerufen, sondern von einem weiteren Faden, der nur für den Aufruf der Behandlungsfunktionen vorgesehen ist, der *Auslieferungs-Faden*. Er kommuniziert mit dem Empfangs-Faden über eine Warteschlange nach dem Erzeuger-Verbraucher-Muster und ermöglichst es den Behandlungsfunktionen wiederum *Paradis-Net*-Funktionen aufzurufen. Würde die Sende-Operation direkt im Kontext des Empfangs-Fadens aufgerufen, be-

[2] In der Kern-Implementierung von *Paradis-Net* kommen hier die Kern-Semaphoren zum Einsatz.

Allgemeine Funktionen
`int initialize(end_point ep[])`
`void finalize()`
`peer_id get_peer_id(char name[])`
Kommunikation
`int send(peer_id to, msg_type type, coop_nr nr, void *msg, int msg_size)`
`int forward(peer_id to, msg_type type, void *msg, int msg_size)`
Behandlungsfunktionen
`void set_handler(msg_type type, int opt, handler_fun *handler)`
`void <handler_fun>(peer_id from, msg_type type, coop_nr nr, void *msg, int msg_size)`
Kooperationen
`coop_nr start_cooperation(rcv_desc *rcvec, int vec_size)`
`int end_cooperation(coop_nr nr, int timeout)`
`int cooperation_finished(coop_nr nr)`

Tabelle 4.1: Die wichtigsten Funktionen von *Paradis-Net*

stände die Gefahr einer Verklemmung, wenn zwei Kommunikationspartner sich gleichzeitig Nachrichten zusenden und dabei blockieren weil die Gegenseite nicht empfängt.

Durch die mehrfädige Programmierung der Bibliothek, kann ein Programm, das *Paradis-Net* verwendet, auf Multi-Prozessor Architekturen implizit parallel ablaufen. Daher ist es wichtig, dass auch in dem Anwendungsprogramm kritische Bereiche geschützt werden. Kritische Bereiche sind dort zu finden, wo Behandlungsfunktionen auf Datenstrukturen zugreifen, die auch in anderen Ausführungskontexten verwendet werden.

4.5.2 Schnittstelle

Die folgenden Abschnitte stellen die wichtigsten Funktionen von *Paradis-Net* vor, welche in Tabelle 4.1 zu finden sind. Abschnitt 4.6 zeigt, wie die *Weiterleitungs-* und *Verteilen/Sammeln-Muster* mit *Paradis-Net* konkret implementiert werden können.

4.5.3 Initialisierung, Peer IDs und das Verschicken von Daten

Die Funktion `initialize` (siehe Tabelle 4.1) muss vor allen anderen Funktionen aufgerufen werden. Sie initialisiert die internen Datenstrukturen, öffnet die lokalen Endpunkte und startet den Empfangs-Faden. Das Argument `ep` enthält eine Liste mit der Konfiguration aller Endpunkte, die auf dem lokalen Knoten geöffnet werden sollen. So muss beispielsweise bei einem TCP/IP-Endpunkt eine Port-Nummer angegeben werden, unter der dieser Endpunkt erreichbar ist. Die inverse Operation `finalize` schließt alle geöffneten Endpunkte, stoppt die Ausführungsfäden und gibt die intern verwendeten Datenstrukturen frei.

In *Paradis-Net* besitzt jeder Endpunkt einen eindeutigen Namen. Dieser Name besteht aus Protokoll- und Adress-Informationen. Der Name eines TCP-

Endpunktes ist beispielsweise: „tcp:*IP-Adresse*:*Portnummer*“. Aus Gründen der Bequemlichkeit und Effizienz werden in den meisten Operationen statt diesen Zeichenketten so genannte *Peer-IDs* verwendet. Dabei handelt es sich um lokal eindeutige Nummern, die von *Paradis-Net* durch die Funktion `get_peer_id` (siehe Tabelle 4.1) zugewiesen und in den übrigen *Paradis-Net*-Funktionen an Stelle der Endpunktnamen verwendet werden.

Die *Paradis-Net* Bibliothek bietet nur eine explizite Sendeoperation: `send` (siehe Tabelle 4.1). Diese Funktion sendet eine typisierte Nachricht an den Endpunkt mit der Peer-ID `to`, im Falle eines Fehlers gibt `send` eine Fehlernummer zurück. Sofort nach Rückkehr der Funktion kann der Speicherbereich (`msg`), der die Nachricht enthält, weiter verwendet werden.

4.5.4 Behandlungsfunktionen

Paradis-Net bietet keine Funktion zum Empfangen von Nachrichten an. Statt dessen werden Behandlungsfunktionen benutzt, die nach Eingang einer Nachricht in Abhängigkeit vom Typ der Nachricht aufgerufen werden. Die Behandlungsfunktion (`handler`) für einen Nachrichtentyp (`type`) werden durch Aufruf der `set_handler` Funktion (siehe Tabelle 4.1) festgelegt. Weil es sich bei den Nachrichtentypen um Ganzzahlen handelt, erfordert dieser Ereignis-gesteuerte Mechanismus, dass beide Seiten eine Übereinkunft über die Bedeutung der Nachrichtentypen getroffen haben. Im Gegensatz zu den im Abschnitt 4.3.3 beschriebenen *Aktiven Nachrichten* sind die Behandlungsfunktionen nicht in ihrer Ausführungszeit begrenzt und dürfen auch weitere *Paradis-Net*-Funktionen aufrufen. Das schließt auch beliebige Sendeoperationen ein.

Die Verwendung von Behandlungsfunktionen erleichtert die Implementierung von Dienstgebern: Sowohl der Empfang einer Anfrage, wie auch der Aufruf der passenden Behandlungsfunktion in einem speziellen Ausführungsfaden werden von der Bibliothek übernommen. Die Behandlungsfunktion dient üblicherweise der Bearbeitung der Anfrage und sendet auch die Antwort an den Dienstnehmer.

Dieses Modell wird in *Paradis-Net* durch die `forward` Funktion (siehe Tabelle 4.1) erweitert. Wird diese Funktion innerhalb einer Behandlungsfunktion aufgerufen, schickt sie eine Nachricht (üblicherweise die unveränderte oder leicht abgewandelte Anfrage) an einen weiteren Knoten. Damit wird auch die Verpflichtung zu antworten an diesen Knoten delegiert. Die Behandlungsfunktion auf dem folgenden Knoten wird anschließend mit einer `peer_id` aufgerufen, die dem Endpunkt des Knoten entspricht, der ursprünglich die Anfrage gestellt hat. Für die Behandlungsfunktion ist also der tatsächliche Absender der Nachricht (der Aufrufer von `forward`) nicht sichtbar, statt dessen scheint es als käme die Anfrage direkt vom Auftraggeber. In Abschnitt 4.6 wird gezeigt, wie diese Funktion zur Implementierung des *Weiterleitungsmusters* verwendet werden kann.

4.5.5 Kooperationen

Während Behandlungsfunktionen die Implementierung von Dienstgebern erleichtern, so sind sie bei der Implementierung von Dienstnehmern hinderlich. Der Grund dafür liegt in der notwendigen Synchronisation von Ausführungsfäden: Nachdem eine Anfrage an einen anderen Knoten geschickt wurde, ist es in

den meisten Fällen erforderlich, vor der Fortsetzung des Programms eine Antwort des Dienstgebers abzuwarten. Der Faden, der die Anfrage abgeschickt hat, muss so lange blockieren, bis innerhalb eines anderen Fadens die entsprechende Behandlungsfunktion mit der erwarteten Antwort aufgerufen wird.

Um die explizite Synchronisation zwischen der Behandlungsfunktion und dem Faden, der die Anfrage gestellt hat, zu vermeiden, führt *Paradis-Net Kooperationen* ein. Eine Kooperation ist ein Konstrukt, das einen Zusammenhang zwischen den verschickten Anfragen und den eingehenden Antworten definiert. Dies geschieht durch die Erzeugung einer speziellen Marke, die allen involvierten Nachrichten angehängt wird. Die Funktion `start_cooperation` (siehe Tabelle 4.1) erzeugt eine solche Marke auf dem Klienten. Der Parameter `rcvec` beschreibt dabei die auf die folgenden Anfrage erwartete Antwort.

Zur Illustration des Konzeptes wird nun als Beispiel der Lebenszyklus einer Kooperation in einem typischen Dienstnehmer/Dienstgeben-Szenario beschrieben:

1. Der Klient ruft die Funktion `start_cooperation` auf, registriert damit die Kooperation und erzeugt eine Marke, die von der Funktion zurückgegeben wird. Der Parameter `rcvec` (*Receive Descriptor*) dient zur Beschreibung der erwarteten Antwort und enthält eventuell auch Zeiger auf Speicherbereiche in denen die Antwort(en) gespeichert werden soll(en). Darüber hinaus enthalten diese Empfangsbeschreibungen auch Kriterien zur Unterscheidung von verschiedenen Nachrichtentypen und Absendern.

2. Der Klient sendet mittels `send` eine Anfrage an den Dienstgeber.

3. Anschließend ruft der Klient `end_cooperation` (siehe Tabelle 4.1) auf. Diese Funktion blockiert so lange, bis die erwartete Antwort eingetroffen ist. (Diese Antwort kann auch aus mehreren Nachrichten von unterschiedlichen Kommunikationspartnern bestehen.)

   Mit der Funktion `cooperation_finished` kann nicht-blockierend überprüft werden, ob die erwartete Antwort bereits eingetroffen ist. Diese Funktion kann genutzt werden, um in der Zeit bis zum Eintreffen der vollständigen Antwort Berechnungen durchzuführen. Es ist auch möglich auf den *Receive Descriptor* zuzugreifen und bereits eingetroffene Teil-Antworten zu verarbeiten bevor alle erwarteten Antworten eingetroffen sind.

   Die vom Klienten erzeugte Marke begleitet die Anfrage auf dem Weg zum Dienstgeber durch Verwendung eines optionalen Parameters der *send* Funktion, dessen Verwendung dafür sorgt, dass die Marke an die eigentliche Nachricht angehängt wird.

4. Auf Dienstgeber-Seite ruft *Paradis-Net* die zuständige Behandlungsfunktion auf und übergibt ihr (unter anderem) die Kooperationsmarke (siehe die Signatur der Behandlungsfunktionen in Tabelle 4.1).

5. Innerhalb der Behandlungsfunktion wird die Anfrage bearbeitet und anschließend wird, wiederum per `send`, eine Antwort an den Dienstnehmer geschickt. Beim Aufruf der `send`-Funktion wird, wie zuvor, die Marke übergeben, wodurch sie auch der von der Antwort-Nachricht angehängt wird und somit an ihren Ausgangspunkt zurückkehrt.

6. Die *Paradis-Net*-Bibliothek auf dem Klienten kann anhand des Nachrichtentyps erkennen, dass es sich bei dieser Nachricht um eine Antwort handelt. Daraufhin wird geprüft, ob es sich bei der angehängten Marke um die Marke einer momentan aktiven Kooperation handelt. Dies ist der Fall und daher speichert die Bibliothek die Antwort direkt in dem Speicherbereich, der beim Aufruf von `start_cooperation` (in 1.) angegeben wurde. Anschließend wird der in `end_cooperation` schlafende Faden aufgeweckt und kann mit der Ausführung fortfahren.

Bei dem beschriebenen Szenario handelt es sich im Prinzip um einen entfernten Methodenaufruf. Kooperationen erlauben darüber hinaus aber noch mehr Möglichkeiten, von denen einige im folgenden Abschnitt dargestellt werden.

4.6 Anwendung

In diesem Abschnitt wird zunächst anhand von zwei Beispielen gezeigt, wie *Paradis-Net* in verteilten Dateisystem eingesetzt werden kann. Anschließend wird beschrieben, wie der von modernen Netzwerkkarten unterstützten entfernte Speicherzugriff (RDMA) transparent für den Anwender eingesetzt wird.

4.6.1 Weiterleitung

In Abschnitt 4.2 wurde das *Weiterleitungsmuster* vorgestellt, welches sowohl in *xFS*, wie auch in *CLF* verwendet wird; Abschnitt 4.3.2 hat gezeigt, dass entfernte Prozeduraufrufe ungeeignet sind, das Muster zu implementieren. In diesem Abschnitt wird nun anhand von C Quelltext vorgeführt, wie die Weiterleitung mit *Paradis-Net* implementiert werden kann. Zur Vereinfachung der Diskussion wird in den Beispielen keine Fehlerbehandlung durchgeführt, darüber hinaus seien die verwendeten Nachrichtentypen bereits bei *Paradis-Net* registriert.

```
void* start(peer_id to, void *msg, int msg_len) {
  coop_nr coop;
  rcv_desc desc =              // Empfangsbeschreibung
    { memory: NULL, size: 0,   // Speicher für Antwort automatisch anlegen
      type: REPLY,             // Akzeptiere nur Typ "REPLY"
      options: RCV_FROM_ANY }; // Akzeptiere Nachr. von bel. Knoten

  coop = start_cooperation(&desc, 1);    // Kooperation anmelden
  send(to, REQUEST, coop, msg, msg_len); // Anfrage abschicken
  ...                                    // (eventuell Berechnungen)
  end_cooperation(coop, 0);              // Antwort abwarten
  return desc.memory;                    // Ergebnis zurückgeben
}
```

Listing 4.1: Verschicken der Anfrage und Empfangen der Antwort

Peer1

Wir betrachten die 3 Knoten aus Abbildung 4.1 nacheinander und beginnen mit *Peer1*, welcher die Anfrage an *Peer2* schickt. Wir nehmen dazu an, dass auf

diesem Knoten die Funktion `start` (Siehe Listing 4.1) aufgerufen wird, die erst dann zurückkehrt, wenn die Antwort empfangen wurde. Dazu muss zunächst eine Empfangsbeschreibung (`recv_desc`) definiert werden, die festlegt, welche Nachrichten als Antwort auf die folgende Anfrage akzeptiert werden. Wie in Abschnitt 4.5.5 beschrieben wurde, kommen grundsätzlich nur solche Nachrichten als Antworten in Frage, die auch die passende Marke tragen. Darüberhinaus ist es noch möglich, zwischen verschiedenen Antworten nach dem Nachrichtentyp und der Herkunft zu unterscheiden. In diesem Fall werden nur Nachrichten vom Typ `REPLY` akzeptiert. Trifft eine solche Nachricht bei *Peer1* ein, wird Speicher für diese Nachricht angelegt bevor sie empfangen wird. Die Adresse des angelegten Speichers wird in der Empfangsbeschreibung `desc` gespeichert und muss vom Benutzer wieder freigegeben werden.

Nach Anmeldung der Kooperation wird die Anfrage mit dem Typ `REQUEST` an *Peer2* geschickt und die Kooperationsmarke angehängt. Anschließend blockiert dieser Faden innerhalb von `end_cooperation`, bis die Antwort eingetroffen ist. Vor dem Aufruf von `end_cooperation` kann *Peer1* noch weitere Operationen ausführen oder auch beliebige *Paradis-Net*-Funktionen aufrufen.

Die `start` Funktion kann auch in einem Dienstnehmer/Dienstgeber-Szenario eingesetzt werden. Würde *Peer2* beispielsweise direkt mit einer Nachricht vom Typ `REPLY` antworten, so würde *Peer1* mit der Funktion aus Listing 4.1 dennoch die Antwort enthalten, da die Empfangsbeschreibung Antworten von allen Knoten akzeptiert.

```
void forward_handler(peer_id from, msg_type type, coop_nr coop,
                     void *msg, int msg_len) {
  peer_id verantwortlicher = finde_verantwortlichen(msg, msg_len);
  forward(verantwortlicher, msg_type, msg, msg_len);
}
```

Listing 4.2: Weiterleitung der Anfrage

Peer2

Nachdem *Paradis-Net* auf *Peer2* die Anfrage empfangen hat, ruft es die Behandlungsfunktion `forward_handler` (Listing 4.2) auf, die zuvor für die Behandlung von Nachrichten des Typs `REQUEST` registriert wurde. Im Sinne des Musters bearbeitet *Peer2* nicht die Anfrage, sondern leitet sie an einen weiteren Knoten weiter, der sie beantworten kann.

Peer2 bestimmt diesen Knoten mit Hilfe der anwendungsspezifischen `finde_verantwortlichen` Funktion, welche die *Peer-ID* des verantwortlichen Knotens zurückgibt. Darauf folgt ein Aufruf der Prozedur `forward`, die für die eigentliche Weiterleitung der Nachricht zuständig ist. Obwohl es sich um keinen Parameter der Funktion `forward` handelt, wird implizit wieder die Kooperationsmarke an die weitergeleitete Nachricht angehängt. *Paradis-Net* stellt sicher, dass `forward` nur aus dem Ausführungskontext einer Behandlungsfunktion heraus aufgerufen wird und bestimmt die zugehörige Kooperationsmarke aus diesem Kontext.

```
void serve_request(peer_id from, msg_type type, coop_nr coop, void *msg,
                   int msg_len) {
  reply_type reply; // Diese Variable wird die Antwort enthalten
  int reply_len;    // Die Länge der Antwort

  fulfill_request(msg, msg_len, &reply, &reply_len);
  send(from, REPLY, coop, &reply, reply_len);
}
```

Listing 4.3: Beantwortung der Anfrage

Peer3

Die weitergeleitete Anfrage von *Peer2* enthält nicht nur die Kooperationsmarke, sondern auch die Adresse von *Peer1*, damit *Peer3* in der Lage ist, diesem direkt zu antworten. Auf *Peer3* extrahiert *Paradis-Net* diese Adresse aus der Nachricht und ruft die Behandlungsfunktion (`serve_request`, Listing 4.3) mit der Peer-ID von *Peer1* auf. Daher ist innerhalb der Behandlungsfunktion `server_request` die Vermittlungsarbeit von *Peer2* nicht erkennbar. Entsprechend einfach ist die Implementierung von `serve_request`: Nach Erfüllung der Anfrage (durch Aufruf von `fullfill_request`), wird die Antwort mittels `send` an den Auftraggeber zurückgeschickt.

Peer1 erhält daraufhin die Antwort auf seine Anfrage von *Peer3*, obwohl er sie an *Peer2* geschickt hat. Aufgrund der mitgeführten Kooperationsmarke kann *Paradis-Net* dennoch den Zusammenhang zwischen der ursprünglichen Anfrage und der eintreffenden Antwort herstellen. Die Antwort wird in einem neu angelegten Speicherbereich abgelegt und eine Referenz auf diesen Speicherbereich in der Empfangsbeschreibung abgelegt, die beim Anlegen der Kooperation der Funktion `start_cooperation` übergeben wurde. *Paradis-Net* stellt weiterhin fest, dass ein Ausführungsfaden in `end_cooperation` auf den Empfang der Antwort wartet, woraufhin dieser aufgeweckt wird.

Die Implementierung von `serve_request` ist für alle Arten von Dienstgebern einsetzbar. Wie bei der Methode `start` aus Listing 4.1 handelt es sich um eine Implementierung, die auch bei der Realisierung eines entfernten Prozeduraufrufs zum Einsatz kommen kann. Würde *Peer1* seine Anfrage direkt an *Peer3* schicken, dann muss `serve_request` nicht verändert werden; der Unterschied zwischen einer weitergeleiteten Anfrage und einer direkten Anfrage wird von *Paradis-Net* verborgen.

4.6.2 Verteilen/Sammeln

Neben dem *Weiterleitungsmuster* wurde in Abschnitt 4.2 das *Verteilen/Sammeln-Muster* als weiteres wichtiges Muster für verteilte Dateisysteme vorgestellt. In diesem Muster schickt ein Knoten (*Peer1*) eine Reihe von Anfragen an andere Knoten und erwartet von jedem beteiligten Knoten (*Peer2*, *Peer3*, ...) eine Antwort (siehe Abbildung 4.2). Eine der Schwierigkeiten liegt hier darin, dass, obwohl die Anfragen in einer bestimmten Reihenfolge verschickt werden, die Antworten in beliebiger Reihenfolge eintreffen können.

Dienstgeber

Die Knoten, welche die Dienstgeberrolle spielen (*Peer2*, *Peer3*, ...) definieren eine Behandlungsfunktion zur Verarbeitung der Anfrage, welche die Antwort direkt an den Auftraggeber zurückschickt. Das Kommunikationsverhalten entspricht somit genau dem von *Peer3* im *Weiterleitungsmuster* und daher ist das Schema der Implementierung auch identisch mit diesem, siehe Listing 4.3.

```
reply_type* start(peer_id* peers, int num_peers, void* msg, int msg_len){
  coop_nr coop;
  int i;
  // Anfordern des Speichers für alle Antworten:
  reply_type *replies = malloc(num_peers * sizeof(reply_type));
  // Initialisieren der Empfangsbeschreibungen:
  rcv_desc *desc = init_desc(peers, num_peers, replies);

  coop = start_cooperation(desc, num_peers); // Kooperation anmelden

  for (i=0; i<num_peers; i++)              // Anfragen verschicken
     send(peers[i], REQUEST, coop, msg, msg_len);
  ...                                      // (eventuell Berechnungen)
  end_cooperation(coop, 0);                // Warten auf die Antworten
  free(desc);                              // Beschreibungen freigeben
  return replies;                          // Antworten zurückgeben
}
```

Listing 4.4: Verschicken der Anfragen

Dienstnehmer

Zur Vereinfachung nehmen wir an, dass an jeden der Dienstgeber die gleiche Anfrage gestellt wird und dass diese Antworten vom Typ `reply_type` sind. Daher wird von der `start` Funktion (Listing 4.4) zunächst mittels `malloc` Speicher für die Antworten angelegt, bevor durch Aufruf von `init_desc` das Feld `desc` mit allen Empfangsbeschreibungen initialisiert wird. Nachdem die Kooperation registriert wurde, werden die Anfragen mittels einer einfachen Schleife versandt. Anschließend wird durch den Aufruf von `end_cooperation` das Eintreffen aller Antworten abgewartet.[3] Zum Schluss erfolgt noch die Freigabe der Empfangsbeschreibungen und die Rückgabe der Antworten.

Die für das Funktionieren des Empfangs wesentliche Arbeit wird von der Funktion `init_desc` übernommen (Listing 4.5). Zunächst wird ein ausreichend großes Feld von Empfangsbeschreibungen angelegt. Anschließend werden innerhalb der Schleife die einzelnen Elemente so initialisiert, dass jedes Element für den Empfang der Antwort eines spezifischen Knotens zuständig ist. Aus dem zuvor (in `start`) angelegten Feld wird jeweils ein Element als Speicherplatz für die von dem entsprechenden Knoten eintreffende Antwort benutzt; der Typ der Nachrichten wird dabei ignoriert. Das fertig initialisiert Feld wird an die `start` Methode zurückgegeben.

[3] Mittels der Funktion `cooperation_finished` könnte dies auch nicht-blockierend erfolgen. Dadurch wäre es möglich, bereits eingetroffene Antworten zu verarbeiten, ohne das alle Antworten vollständig vorliegen.

```
rcv_desc *init_desc(peer_id *to, int n, reply_type *replies) {
  rcv_desc *desc = malloc(n * sizeof(rcv_desc));
  int i;
  for (i=0; i<n; i++) {
      desc[i].memory =  &replies[i];
      desc[i].size =    sizeof(reply_type);
      desc[i].from =    to[i];
      desc[i].options = RCV_ANY_TYPE;
  }
  return desc;
}
```

Listing 4.5: Initialisierung der Empfangsbeschreibung

Die mit dem so initialisierten Feld aufgerufenen `start_cooperation`-Funktion bereitet *Paradis-Net* darauf vor, dass `n` Antworten von denen im `from`-Feld angegebenen Knoten erwartet werden. Der Nachrichtentyp `REPLY` ist auf dem Dienstnehmer als *Kooperationstyp* registriert. Sobald eine Antwort mit diesem Typ auf dem Dienstnehmer ankommt, untersucht *Paradis-Net* die angehängte Kooperationsmarke und kann auf dieser Weise die zugehörige Empfangsbeschreibung finden. Aufgrund des Absenders der Nachricht, kann die Antwort direkt an der vorgesehenen Stelle im Speicher abgelegt werden. Die Kooperation gilt dann als beendet, wenn alle in der Empfangsbeschreibung angegebenen Nachrichten eingetroffen sind, das heißt, wenn alle Dienstgeber eine Antwort geschickt und damit die Anfrage beantwortet haben.

Zusammenfassung

Die letzten beiden Abschnitte haben durch die Implementierung der als schwierig geltenden Kommunikationsmuster *Weiterleitung* und *Verteilen/Sammeln* die Ausdruckskraft des Kooperations-Paradigmas demonstriert. An dieser Stelle sei noch einmal angemerkt, dass es sich bei den vorgestellten Programmstücken nicht um Pseudo-Code handelt. Sie sind vollständig funktionsfähig und wurden, bis auf die fehlende Fehlerbehandlung, nicht vereinfacht.

Der folgende Abschnitt zeigt am Beispiel von entferntem Speicherzugriff, wie *Paradis-Net* hardwarespezifische Optimierungen transparent zur Verfügung stellen kann.

4.6.3 Entfernter Speicherzugriff

Entfernter Speicherzugriff (Remote Direct Memory Access, RDMA) ist eine Technik, bei dem zwei oder mehr Computer über ein Netzwerk mittels direktem Speicherzugriff (Direct Memory Access), auf den Speicher eines anderen zugreifen können. Dadurch, dass weder der Prozessor noch Cache für den Transfer benutzt werden und daher der Transfer parallel zu den übrigen Systemaktivitäten ablaufen kann, ist er besonders in Anwendungen nützlich, die hohen Durchsatz und geringe Latenz benötigen. Die bekannteste Implementierung von RDMA ist die über InfiniBand, es gibt aber auch Vorschläge für RDMA über TCP/IP und Lösungen, wie beispielsweise VI-GM, einer Implementierung der *Virtual Interface Architecture* (VIA [85]), die einen Teil der Funktionalität durch Software

emulieren.

Um RDMA zu ermöglichen, muss zunächst lokaler Speicher registriert werden und eine Zugriffskennung an den Rechner verschickt werden, dem der Zugriff erlaubt werden soll. Mit Hilfe der Zugriffskennung kann der entfernte Rechner anschließend entweder lesend oder schreibend auf den zuvor registrierten Speicher zugreifen. Ting Zheng [88] hat gezeigt, dass sich diese Vorgehensweise gut auf den Kooperationsmechanismus von *Paradis-Net* abbilden lässt, falls die Größe der erwarteten Antwort und deren Herkunft bei Anmeldung der Kooperation bekannt ist.

Bei Anmeldung der Kooperation wird in diesem Fall nicht nur die Kooperationskennung erzeugt, es wird darüber hinaus auch der zum Empfang vorgesehene Speicher für den entfernten Speicherzugriff angemeldet. Beim Verschicken der Anfrage wird anschließend nicht nur die Kooperationskennung an die Nachricht angehängt, sondern auch die Zugriffskennung für den Speicher. Dadurch kann *Paradis-Net* auf dem entfernten Rechner die Antwort auf die Anfrage direkt in den Speicher des Klienten schreiben, ohne den Prozessor oder das Betriebssystem des Klienten zu involvieren.

Die Besonderheit an dieser Vorgehensweise ist, dass die Optimierung für den Benutzer von *Paradis-Net* völlig transparent ist und keine spezielle Programmierung erfordert. Der Grund dafür liegt in der Ähnlichkeit zwischen der Semantik von Kooperationen und von RDMA. Weiterhin in es auch möglich, die übrige Kommunikation über RDMA abzuwickeln, indem registrierte Speicherbereiche vorgehalten werden, und bei Bedarf zur Kommunikation eingesetzt werden. Mehr Details finden sich in der Diplomarbeit von Ting Zheng [88].

4.7 Einsatz von *Paradis-Net*

Ursprünglich benutzte *Clusterfile* direkt die TCP/IP-Sockets-Schnittstelle über GLIBC2. Daraus resultierte ein Vermischung von verschiedenen Programmierebenen: Zum einen die Problemlösungsebene, auf der der Dienst realisiert wurde und zum anderen die komplizierte Handhabung der TCP/IP-Schnittstelle. Der entstandene Quelltext war aufgrund dieser Vermischung schwer zu pflegen und schwer zu verstehen. Mit der Einführung von *Paradis-Net* hat sich die Qualität des Quelltextes subjektiv verbessert und darüber hinaus hat auch der Umfang deutlich abgenommen (um zirka 50%).

Auch in anderen Projekten kam *Paradis-Net*zum Einsatz. Im Medienserver des *CHIL* Projektes [79] löste *Paradis-Net* die Implementierung auf Basis von TCP/IP-Sockets ab. Darüber hinaus wurde im Rahmen dieses Projektes auch ein Modul für Unix Domain Sockets und Unix Pipes implementiert.

4.8 Zusammenfassung

Paradis-Net ist eine Netzwerkschnittstelle, die sich besonders gut zur Implementierung von komplexen Protokollen mit mehreren Teilnehmern eignet. Es wurde als Schnittstelle für das Dateisystem *CLF* entworfen und wird dort für alle Netzwerkoperationen eingesetzt.

Um die Zusammenarbeit zwischen mehreren Teilnehmern zu erleichtern, führt *Paradis-Net* Kooperationen ein, die es dem Benutzer erlauben, das Ergeb-

nis einer Anfrage, an der mehrere Partner teilnehmen, zu definieren. In diesem Abschnitt wurde der Mechanismus beschrieben und seine mögliche Anwendung anhand der Implementierung von zwei, für parallele Dateisysteme typischen, Kommunikationsmustern demonstriert. Darüber hinaus wurde am Beispiel von RDMA gezeigt, wie die Bibliothek –für den Benutzer transparent– Optimierungen vornehmen und dass *Paradis-Net* auch außerhalb von parallelen Dateisystemen sinnvolle Aufgaben übernehmen kann.

Für *Paradis-Net* existiert auf Basis von TCP/IP eine Standard-Implementierung in der Benutzerebene und im Linux-Kern. Für die Benutzerebene wurde weiterhin ein Prototyp auf der Basis von VIA entwickelt.

Kapitel 5

Entwurf und Implementierung

Dieses Kapitel beschreibt den Entwurf des Dateisystems *CLF*. Im folgenden Abschnitt werden zunächst die Rollen beschrieben, die ein Knoten bezüglich des Dateisystem einnehmen kann und wie die einzelnen Komponenten von *CLF* zusammen arbeiten. Der Rest des Kapitels beschreibt detailliert den Aufbau der einzelnen Komponenten und die Kommunikationmuster, die bei der Zusammenarbeit entstehen.

5.1 Knotenrollen in *CLF*

In Abschnitt 2.2.5 wurden die drei verschiedenen Knotenrollen in parallelen Dateisystemen beschrieben, an denen sich auch *CLF* orientiert: *Metadatenmanager*, *Daten-Server* und *Klient*. Ein Knoten kann mehrere dieser Rollen gleichzeitig einnehmen, wobei die Rolle des *Metadatenmanagers* nur von einem Knoten eingenommen werden kann.

5.1.1 *Metadatenmanager*, *Daten-Server* und *Klient*

Die Infrastruktur des Dateisystems wird von einem *Metadatenmanager* und einem oder mehreren *Daten-Servern* gebildet. Der *Metadatenmanager* übernimmt die zentrale Verwaltung des gesamten Dateisystems und speichert sämtliche Metadaten (Verwaltungsdaten). Dazu gehören unter anderem die Dateihierarchie sowie Namen und weitere Informationen über die einzelnen Dateien.

Die *Daten-Server* speichern ausschließlich den Inhalt der Dateien. Sie fungieren als eine Art entfernt zugreifbares Blockgerät; die gespeicherten Blöcke werden über eine Kombination aus Datei- und Block-Index adressiert. Existieren in einer Dateisystemkonfiguration mehrere *Daten-Server* werden die Blöcke der Dateien reihum auf diese verteilt. Der Zusammenhang zwischen einem Dateinamen und dem Datei-Index, sowie die Zuweisung der einzelnen Blöcke der Datei auf die *Daten-Server*, ist Teil der Metadateninformationen, die auf dem *Metadatenmanager* gespeichert werden.

Bei *CLF* wurden *Metadatenmanager* und *Daten-Server* als Benutzerebenen-Programme implementiert. Diese Programme speichern die anfallenden Daten in dem lokalen Dateisystem des Knotens auf dem sie ablaufen. Der modulare Entwurf des *Metadatenmanagers* wird in Abschnitt 5.3 vorgestellt; der Entwurf

des *Daten-Servers* konnte von *Clusterfile* übernommen werden [44] und wird daher, auch wegen seiner geringen Relevanz für die Metadatenverarbeitung, nur im Anhang behandelt.

Ein *CLF* Dateisystem besitzt zur Laufzeit einen festen *Metadatenmanager* und eine feste Anzahl *Daten-Server*; ein Knoten des Clusters nimmt eine der Rollen ein, indem das entsprechende Programm auf ihm abläuft. Beim Start des Dateisystems werden die *Daten-Server* von dem *Metadatenmanager* konfiguriert und können anschließend von *Klienten* benutzt werden.

Die Rolle des *Klienten* wird von jedem Knoten eingenommen, der das Dateisystem benutzt. Um ein *CLF*-Dateisystem zu benutzen, muss es zunächst auf dem Knoten montiert werden. Das Montieren von Dateisystemen ist in UNIX-Betriebssystemen eine notwendige Voraussetzung, um ein Dateisystem zu benutzen. Dabei wird ein Dateisystem als Teilbaum in eine globale Verzeichnishierarchie eingesetzt. Die Verzeichnishierarchie wird von dem Betriebssystem verwaltet, das die Zusammenarbeit unterschiedlicher Dateisysteme durch die Festlegung einer einheitlichen Schnittstelle ermöglicht. Diese VFS-Schnittstelle (VFS = *Virtual Filesystem Switch*) wird in Kapitel 3 ausführlich vorgestellt.

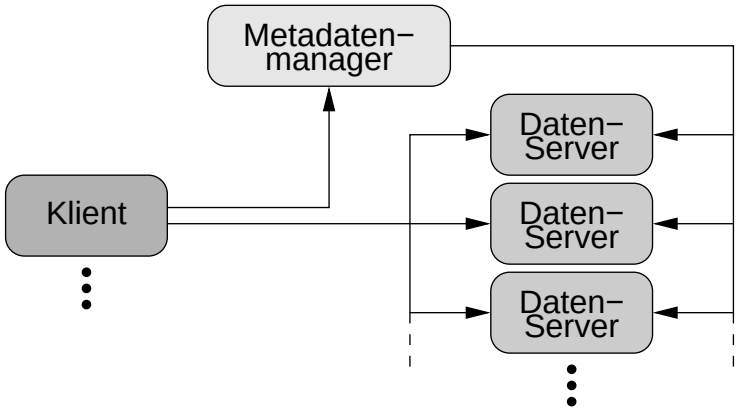


Abbildung 5.1: Kommunikation zwischen den Komponenten in *CLF* (ohne Metadaten-Surrogat)

Das Montieren von *CLF* wird durch ein Betriebssystem-Modul ermöglicht, das die VFS-Schnittstelle erfüllt (Abschnitt 5.2 beschreibt das Modul im Detail) und über *Paradis-Net* mit dem *Metadatenmanager* und den *Daten-Servern* kommuniziert. Beim Montieren nimmt das Modul zunächst mit dem *Metadatenmanager* Kontakt auf und erhält von diesem die Adressen der *Daten-Server*. Anschließend kommuniziert es je nach Bedarf direkt mit den entsprechenden Dienstgebern. Abbildungen 5.1 illustriert die Kommunikationswege zwischen den einzelnen Komponenten. Die Pfeile stellen in der Regel Anfrage-Antwort- oder Verteilen-Sammeln-Kommunikationsmuster (vergleiche Abbildung 4.2) dar, bei denen eine der Komponenten eine Anfrage an einen oder mehrere Partner sendet und anschließen eine Antwort von jedem Partner erhält. In der Abbildung repräsentiert die Richtung eines Pfeils die Richtung der Anfrage.

Die Abbildung zeigt zur Vereinfachung nur einen Klienten, weitere Klienten würden auf gleiche Weise mit den anderen Komponenten kommunizieren.

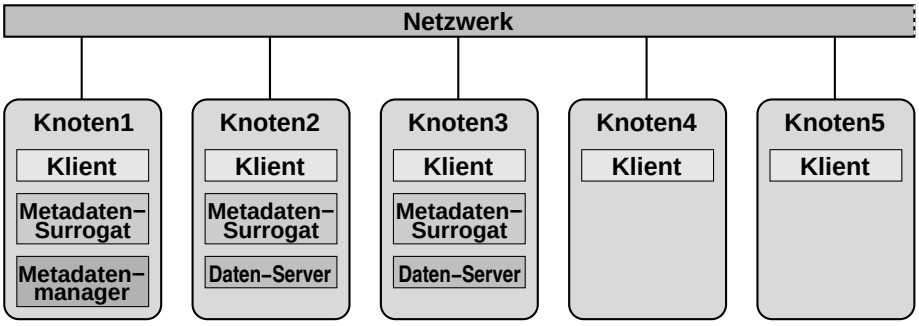


Abbildung 5.2: Verschiedene Knotenrollen in einer Beispielkonfiguration von *CLF*

Dies und auch die mögliche Verwendung von mehr als drei Daten-Servern wird durch „. . .“ angedeutet. Die Anzahl der Daten-Server wird bei Initialisierung des Dateisystems vom *Metadatenmanager* festgelegt und kann während der Laufzeit des Dateisystems nicht verändert werden, während die Anzahl der Klienten zu jeder Zeit flexibel ist.

5.1.2 *Metadaten-Surrogate*

CLF fügt dem bisher beschriebenen Entwurf eine weitere Rolle hinzu: das *Metadaten-Surrogat*. Eine Schwäche des Entwurfs und damit auch der bisherigen Clusterdateisysteme ist der zentrale *Metadatenmanager*, der sich bei einer großen Zahl an Klienten zu einem Flaschenhals entwickeln kann. Das Ziel der Einführung von *Surrogaten* ist es daher, durch die Abgabe der Aufgaben des *Metadatenmanagers* an die *Surrogate*, eine Verteilung der Anfrage-Last auf mehrere Knoten und damit einen höheren Metadatendurchsatz zu erreichen. Die *Surrogate* wurden als Benutzerebenen-Programme implementiert und ihre Zahl ist bei Initialisierung des Dateisystems einstellbar, kann aber im laufenden Betrieb nicht geändert werden.

Abbildung 5.2 zeigt eine Beispielkonfiguration von *CLF* bei der die ersten drei Knoten mehrere Rollen einnehmen. Knoten 1 ist beispielsweise gleichzeitig *Metadatenmanager*, *Metadaten-Surrogat* und *Klient*. Die Konfiguration besitzt drei *Surrogate* und zwei *Daten-Server*, während alle dargestellten Knoten auch *Klienten* sind.

Die *Metadaten-Surrogate* ersetzen den *Metadatenmanager* nicht vollständig. Er ist weiterhin für die Konfiguration aller Komponenten (einschließlich der *Metadaten-Surrogate*) zuständig und erfüllt die wenigen Aufgaben, die eine zentrale Koordination verlangen, wie beispielsweise die Verwaltung der Datei-Indizes. Weiterhin senden die *Surrogate* Aktualisierungsnachrichten über die Veränderungen der Metadaten an den *Manager*, der diese persistent sichert.

Die Aufgabe der *Metadaten-Surrogate* ist es, alle Anfragen der *Klienten* zu beantworten (mit Ausnahme der Montier-Anfragen, die aber nicht zeitkritisch sind). Die *Metadaten-Surrogate* teilen die Metadaten reihum untereinander auf (vergleichbar mit den Blöcken auf den *Daten-Servern*), replizieren die Metadaten aber nicht, weil dies die Konsistenzerhaltung erschweren würde. Aufgrund von Abhängigkeiten zwischen Metadaten, die auf unterschiedlichen *Surrogaten*

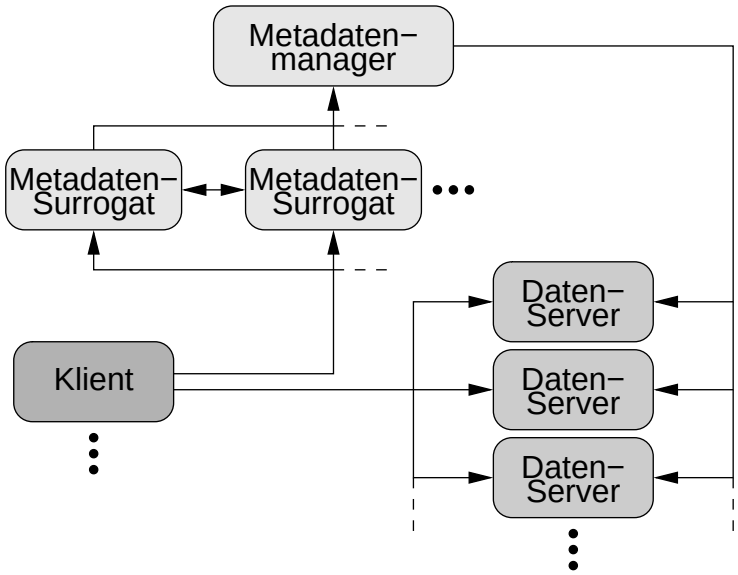


Abbildung 5.3: Kommunikation zwischen den Komponenten in *CLF*

gespeichert werden, ist es bei einigen Klienten-Anfragen notwendig, dass mehrere *Surrogate* zusammen arbeiten. Abbildung 5.3 illustriert die durch das Hinzukommen der *Surrogate* veränderten Kommunikationsrelationen (vergleiche Abbildung 5.1). An der Abbildung ist ablesbar, dass die Metadatenverwaltung, ähnlich wie zuvor schon die Datenverwaltung, durch die Verteilung der Aufgabe auf mehrere Knoten an Skalierbarkeit gegenüber den *Klienten* gewonnen hat.

Der Entwurf der Metadaten-Surrogate wird detailliert in Abschnitt 5.4 behandelt, der folgende Abschnitt 5.2 beschreibt zunächst den Aufbau des Linux-Kern-Moduls, das auf den Klienten zum Einsatz kommt.

5.2 Klient

Die Rolle des Klienten in einem verteilten Dateisystem wird von Rechnern eingenommen, die auf das Dateisystem zugreifen. dies kann auf unterschiedlichen Wegen geschehen:

Zunächst gab es für *CLF* nur eine Bibliothek, welche die Betriebssystems-Schnittstelle nachahmt. Oberhalb dieser Bibliothek wurde eine Unterstützung für die MPI-IO-Schnittstelle [34] implementiert, die es schon bestehenden MPI-Programmen ermöglicht, das Dateisystem zu benutzen. [81] Vor der Einführung des Betriebssystem-Moduls mussten die übrigen, nicht-MPI Anwendungen auf die Verwendung von *CLF* angepasst werden, indem sie, statt der üblichen Systemaufrufe über die glibc-Bibliothek [30], die entsprechenden Funktionen in der *CLF*-Bibliothek verwenden.

Damit alle Programmen mit *CLF* zusammen arbeiten können, wurde das Betriebssystem-Modul für die Dateisystemschnittstelle *VFS* (siehe auch Abschnitt 3) entwickelt, das es dem Linux-Kern erlaubt, ein *CLF*-Dateisystem zu montieren. Dadurch ist es nicht notwendig, ein Programm anzupassen, solan-

ge es über die üblichen Systemaufrufe auf das Linux-Dateisystem zugreift. Das Kern-Modul ist eine wichtige Voraussetzung, damit *CLF* nicht nur spezialisierten Anwendungen zugänglich ist.

Diesem Vorteil stehen die in Abschnitt 3 beschriebenen Einschränkungen entgegen, denen sich ein Linux-Dateisystem unterwerfen muss. Das *VFS* erfordert die Verwendung des Linux-Dateimodells - eine notwendige Bedingung für die Co-Existenz mehrerer Dateisysteme innerhalb eines einzigen Dateibaumes. Das *VFS* nimmt dem nativen Dateisystem einerseits einige Aufgaben ab, während es andererseits auch die Optimierungsmöglichkeiten einschränkt.

Der folgende Abschnitt macht einige grundsätzliche Anmerkungen zu der Implementierung der Betriebssystem-Moduls, anschließend wird die Kommunikation mit Metadatenmanager und Daten-Server betrachtet.

5.2.1 Das Kern-Modul

Die Linux-Unterstützung für *CLF* ist als Kern-Modul implementiert. Unter Verwendung von Modulen ist es möglich die Funktionalität des monolithischen Linux-Kerns zu erweitern, ohne dass der Kern neu kompiliert oder neu gestartet werden muss. Module können zur Laufzeit des Kerns geladen und wieder entfernt werden. Beim Laden registriert das *CLF*-Modul den neuen Dateisystemtyp „clf“ und ermöglicht so das Montieren eines *CLF*-Dateisystems (siehe auch Kap. 3).

Ein montiertes Dateisystem wird im Kern über den *Superblock* identifiziert. Zum Montieren sendet daher die Funktion `clf_get_sb()` (*get superblock*) eine entsprechende Anfrage an den Metadatenmanager. Als Antwort erhält sie das *Layout* des Dateisystems: Die Anzahl und Adressen der Daten-Server (und die der Surrogate), sowie Informationen über die Wurzel-Inode des Dateisystems, die sie in der Superblock-Datenstruktur speichert. Die Adressen der Daten-Server (und der Surrogate) sind die einzigen Daten, die auf dem Klienten nicht veralten, alle übrigen Informationen dürfen nur ein einziges Mal benutzt werden, da sie im nächsten Augenblick schon verändert worden sein können.

Der Grund dafür liegt darin, dass *CLF*, wie jedes Netzwerkdateisystem, von mehreren unabhängigen und nicht koordinierten Klienten verwendet wird. Dadurch, dass ein Klient jederzeit das Dateisystem ändern kann, ist es den anderen Klienten nicht möglich, die in einer Antwort erhaltenen Ergebnisse mehr als einmal zu benutzen. Manche Dateisysteme, wie beispielsweise NFS [70] erlauben es den Klienten, bestimmte Informationen für eine festgelegte Zeit zwischenzuspeichern. Inode-Informationen werden alle 3 Sekunden, Verzeichnisinformationen werden alle 30 Sekunden durch einen Zugriff auf den NFS-Server aktualisiert. Dadurch kann der NFS-Klient wiederholte Zugriffe auf die gleiche Datei sehr effizient verarbeiten, aber gleichzeitig entsteht dadurch ein Konsistenzproblem, weil NFS-Server zustandslos (engl. stateless) sind und daher nicht feststellen können, ob die von ihren Klienten verwendeten Informationen eventuell schon veraltet sind.

Dem gegenüber speichern andere Dateisysteme, wie zum Beispiel PVFS [17], gar keine Informationen auf den Klienten und vermeiden dadurch die Konsistenzprobleme auf Kosten der Effizienz. Der Ansatz von *CLF* ist ähnlich, allerdings gibt es eine Ausnahme: *CLF* hat einen zustandsbehafteten Metadatenmanager, der eine Liste der von Klienten geöffneten Dateien verwaltet, um die *UNIX-Semantik bezüglich offener Dateien* zu erhalten. Nach dieser Seman-

tik kann eine geöffnete Datei zwar gelöscht werden, aber dadurch verschwindet zunächst nur ihr Eintrag im Verzeichnis. Der Inhalt der Datei bleibt bestehen solange noch mindestens ein Prozess die Datei geöffnet hat. Dieser Prozess kann die Datei weiterhin normal verwenden, allerdings wird der Inhalt gelöscht, wenn der Prozess die Datei schließt. Diese Semantik wird weder von NFS, noch von PVFS eingehalten.

Für die Kommunikation zwischen den einzelnen Komponenten verwendet *CLF* *Paradis-Net*, das in Kapitel 4 vorgestellt wurde. *Paradis-Net* wurde ursprünglich als eine Bibliothek in der Benutzerebene entwickelt und musste zunächst in den Kern portiert werden, damit es in dem Kern-Modul zum Einsatz kommen konnte. Dabei wurde aus Zeitgründen allerdings nur die TCP/IP-Version verwirklicht. Während die Originalversion von *Paradis-Net* für TCP/IP die Socket-Schnittstelle von glibc und so indirekt Systemsaufrufe des Kerns verwendet, ist dies für *Paradis-Net* innerhalb des Kerns nicht ohne weiteres möglich. Die Kernversion benutzt auch Sockets, setzt aber auf einer etwas tieferen Ebene an. Es besteht eine große strukturelle Ähnlichkeit zwischen der Benutzerebenen-Version und der Kern-Version, aber viele kleine Änderungen haben es unmöglich gemacht, einen gemeinsamen Codeteil heraus zu faktorisieren.

Die Nachrichtentypen von *Paradis-Net* bezeichnen den Typ der Anfragen an Metadatenmanager und Daten-Server, wie beispielsweise „Datei öffnen“ oder „Dateisystem montieren“. Die Parameter einer Anfrage folgen als Inhalt der Nachricht. Man kann diese Technik mit entfernten Methodenaufrufen vergleichen: Der Nachrichtentyp steht im Prinzip für den Namen der aufzurufenden Funktion auf dem Server, der Nachrichteninhalt entspricht den Parametern der Funktion. Durch die erweiterten Möglichkeiten der Kooperations-Semantik von *Paradis-Net* ist es möglich, eine gemeinsame Anfrage an mehrere Daten-Server zu schicken und so mehrere Blöcke parallel zu lesen oder zu schreiben. Die entspricht dem Verteilen/Sammeln-Muster, das in Abschnitt 4.6 vorgestellt wurde.

5.2.2 Verteilung der Metadaten

CLF ermöglicht es, die Verwaltung der Metadaten auf mehrere Metadaten-Surrogate zu verteilen. Der Einsatz von Metadaten-Surrogaten ist optional und wird beim Start des Dateisystems festgelegt. Werden Surrogate verwendet, stellt der Klient seine Anfragen an diese, andernfalls (wie bisher beschrieben) direkt an den Metadatenmanager.

Das für eine bestimmte Anfrage verantwortliche Metadaten-Surrogat wird von dem Klienten mit Hilfe einer Hash-Funktion aus dem Dateiindex (der Inode-Nummer) berechnet. Weil an der Bearbeitung von bestimmten Anfragen mehrere Surrogate beteiligt sind, erhält der Klient in bestimmten Fällen die Antwort auf eine Anfrage nicht von dem Surrogat an das er ursprünglich die Anfrage gesendet hat. Dabei handelt es sich um eine Anwendung des Weiterleitungs-Musters (siehe Abschnitt 4.4.3), das aufgrund der Kooperations-Semantik von *Paradis-Net* auf Seiten des Klienten keiner besonderen Behandlung bedarf.

5.2.3 Kommunikation mit Metadatenmanager (Surrogat)

Die *Paradis-Net*-Aufrufe des Klienten bei Verwendung von Surrogaten unterscheiden sich von denen ohne Surrogate ausschließlich durch die Adressierung der Anfragen. Daher wird im Rest dieses Abschnitts diese Unterscheidung nicht

mehr gemacht und zugunsten einer einfachereren Beschreibung davon ausgegangen, dass der Klient alle Anfragen an den Metadatenmanager stellt.

Im Vergleich zu der großen Menge an Systemaufrufen, die an das Dateisystem gerichtet werden können (siehe Kapitel 3), ist die Zahl der Anfragen an den Metadatenmanager sehr gering. Der Grund dafür ist, dass es sich bei vielen Systemaufrufen um Bequemlichkeitsmethoden handelt, die im Prinzip redundant sind. Ein Beispiel dafür sind die Systemaufrufe `statfs()`, `fstatfs()`, `statfs64()`, `fstatfs64()` und `ustat()`, die zum Erfragen des Status' eines Dateisystems benutzt werden, sich aber nur in ihren Parametern unterscheiden. Während `statfs()` einen Pfad übergeben bekommt, erwartet `fstatfs()` einen gültigen Dateideskriptor und `ustat()` die Gerätenummer eines montierten Dateisystems. Mit Hilfe der jeweiligen Parameter kann die VFS-Schicht das Dateisystem finden, auf das sich der Aufruf bezieht und das zugehörige Supernode-Objekt lokalisieren. Am Ende bewirken alle diese Funktionen den Aufruf der Methode `statfs()` des Supernode-Objektes.

Insgesamt werden in *CLF* 14 verschiedene Nachrichtentypen für die Kommunikation zwischen Klient und Metadatenmanager verwendet. Diese lassen sich in zwei Kategorien einordnen. Zum einen gibt es die Nachrichten, die einem bestimmten Systemaufruf entsprechen. Anknüpfend an das Beispiel aus dem vorangegangen Abschnitt, wird von der `statfs()` Methode des Supernode-Objektes eine Anfrage des Typs `CLF_RQ_STATFS` an den Metadatenmanager geschickt, um den aktuellen Status des Dateisystems zu erfragen. Wir nennen diese Nachrichten *explizite* Anfragen, im Gegensatz zu den *impliziten* Anfragen (Tabelle 5.2), die in keinem direkten Zusammenhang zu der Aufgabe des Systemsaufrufs stehen. Die impliziten Nachrichten werden unter anderem für die in Abschnitt 3.5 beschriebene Pfadauflösung benötigt.

5.2.3.1 Explizite Anfragen

Die von *CLF* verwendeten, expliziten Anfragen sind in Tabelle 5.1 aufgelistet. Die verschiedenen Nachrichtentypen lassen sich leicht den gleichnamigen Einschubmethoden der Linux-Kerns zuordnen (siehe Auch Kapitelrefbasics:vfs); so wird beispielsweise von der Einschubmethode `create()`, definiert in Inode-Objekten, eine Nachricht des Typs `CLF_RQ_CREATE` an den Metamanager geschickt, die eine reguläre Datei in einem Verzeichnis erstellt. Die Einschubmethode sendet dazu die Parameter der Methode (Verzeichnis, Dateiname und Modus der zu erstellenden Datei) an den Metadatenmanager, der den entsprechenden Eintrag in dem Verzeichnis vornimmt und eine freie Inode-Nummer für die Datei reserviert. Verlief die Operation erfolgreich, werden eine Erfolgsnachricht und die Metadaten der neuen Datei an den Klienten gesendet, ansonsten ein Fehlercode. Der Typ der Antwortnachricht ist analog zu der Anfrage (engl. *Request*, Abkürzung: RQ), `CLF_RS_CREATE` (engl. *Result*, Abkürzung: RS). Der Klient initialisiert im Erfolgsfall das lokale Inode-Objekt mit den empfangenen Metadaten, setzt die Einschubmethoden des neu erstellten Objektes und trägt den neuen Verzeichniseintrag im lokalen Verzeichnis-Cache ein. Obwohl die Informationen sofort verfallen, ist dies dennoch notwendig, da das VFS diesen Eintrag für die interne Konsistenzerhaltung seiner Datenstrukturen benötigt. Wird der Verzeichniseintrag zu einem späteren Zeitpunkt im Verzeichnis-Cache gefunden, wird trotzdem erneut der Metadatenmanager kontaktiert, um die Aktualität der Daten sicher zu stellen (siehe auch den folgenden Abschnitt 5.2.3.2).

Explizite Anfragen	
CLF_RQ_MOUNT	Benachrichtigt den Metadatenmanager darüber, dass ein Klient beginnt das Dateisystem zu nutzen und fragt gleichzeitig die Layoutinformationen des Dateisystems an. Parameter: keine
CLF_RQ_UMOUNT	Benachrichtigt den Metadatenmanager darüber, dass der Klient das Dateisystem nicht mehr verwendet. Parameter: keine
CLF_RQ_CREATE	Erzeugt eine reguläre Datei. Parameter: Oberverzeichnis, Name der Datei, Modus, Benutzer, Gruppe
CLF_RQ_UNLINK	Löscht eine reguläre Datei. Parameter: Verzeichnis, Datei
CLF_RQ_MKDIR	Erzeugt ein Verzeichnis. Parameter: Oberverzeichnis, Name des Verzeichnisses, Modus, Benutzer, Gruppe
CLF_RQ_RMDIR	Löscht ein Verzeichnis. Parameter: Oberverzeichnis, Verzeichnis
CLF_RQ_RENAME	Umbenennung und/oder Verschiebung einer Datei oder eines Verzeichnisses. Parameter: Altes Oberverzeichnis, Neues Oberverzeichnis, Datei/Verzeichnis, alter Name, neuer Name
CLF_RQ_STATFS	Fragt den Status des Dateisystems ab. Parameter: keine
CLF_RQ_OPEN	Öffnet eine Datei. Parameter: Datei, Öffnungsmodus (Flags)
CLF_RQ_CLOSE	Schließt eine Datei. Parameter: Datei
CLF_RQ_READDIR	Liefert den Inhalt eines Verzeichnisses. Parameter: Verzeichnis

Tabelle 5.1: Explizite Anfragen des Klienten an den Metamanager
Die Parameter „(Ober-)Verzeichnis“ und „Datei“ werden als Inode-Nummern kodiert. (siehe 5.2.3.3)

Nach diesem Schema sind alle zu den in Tabelle 5.1 aufgelisteten Anfragetypen gehörigen Einschubmethoden definiert, wobei die Anfragen zum Öffnen und Schließen von Dateien und zum Auflisten von Verzeichnissen eine Sonderstellung einnehmen. Die Nachrichten `CLF_RQ_OPEN` und `CLF_RQ_CLOSE` erlauben es dem Metadatenmanager die korrekte UNIX-Semantik bezüglich offener Dateien einzuhalten. Nach dieser Semantik können geöffnete Dateien zwar von anderen Prozessen gelöscht werden, ihr Inhalt verschwindet allerdings erst dann, wenn der letzte Prozess die Datei schließt. Dazu ist es notwendig, dass der Metadatenmanager eine Liste aller offenen Dateien führt, damit eine Lösch-Anfrage so lange verzögert wird, bis die Datei auf keinem Knoten mehr geöffnet ist (siehe auch Abschnitte 5.3).

5.2.3.2 Implizite Anfragen

Die *impliziten Anfragen* an den Metadatenmanager sind einerseits die Nachrichten, die für die in Abschnitt 3.5 beschriebene Pfadauflösung benötigt werden und andererseits die Nachrichten, die den Metadatenmanager über eine Änderung der Metadaten informieren. Tabelle 5.2 enthält die verwendeten *Paradis-Net* Nachrichtentypen.

Implizite Anfragen	
CLF_RQ_READ_INODE	Fordert die Metadaten einer Datei oder eines Verzeichnisse (Inode) von dem Metadatenmanager an. Parameter: Datei/Verzeichnis
CLF_RQ_NOTIFY_CHANGE	Informiert den Metadatenmanager über eine Änderung der Metadaten einer Datei oder eines Verzeichnisses. Parameter: Datei/Verzeichnis, geänderte Metadaten
CLF_RQ_LOOKUP	Sucht in einem Verzeichnis nach einer Datei oder einem Verzeichnis eines bestimmten Namens. Parameter: Oberverzeichnis, Name des gesuchten Eintrags

Tabelle 5.2: Implizite Anfragen des Klienten an den Metamanager
Die Parameter „Verzeichnis“ und „Datei“ werden als Inode-Nummern kodiert. (siehe 5.2.3.3)

Machrichtentyp `CLF_RQ_READ_INODE`: Nachrichten dieses Typs werden in der Implementierung der `getattr()` Methode von Inode Objekten zum Einsatz. Die Methode dient der Abfrage der Standard-Metadaten eines Inode-Objektes. Diese Metadaten sind folgende:

ino Die Nummer des Inode-Objektes.

mode Der Modus des Inode-Objektes. In diesem Attribut sind sowohl der Inode-Typ (Datei, Verzeichnis, etc.), wie auch die Zugriffsrechte der Inode gespeichert.

nlink Die Anzahl der so genannten „Hardlinks“ der Inode, welche anzeigt wie oft die Inode referenziert wird, also wie oft sie in Verzeichnissen aufgelistet ist. Bei UNIX-Dateisystemen ist es möglich, dass ein und dieselbe Inode in mehreren Verzeichnissen referenziert wird. Erreicht dieser Zähler den Wert 0, wird die Inode gelöscht.

uid, gid Die Benutzer- und Benutzergruppennummer des Besitzers der Datei. In verteilten Dateisystem besteht die Schwierigkeit, dass die Abbildung von Benutzern auf Nummern auf allen Rechnern einheitlich sein muss. Diese Dateisysteme gehen, wie auch *CLF*, davon aus, dass dies gewährleistet ist (z.B. durch „Yellow Pages“ bzw. „NIS“ (Network Information Service) [78]).

atime speichert den Zeitpunkt des letzten Zugriffs auf eine Datei oder eine Verzeichnis. (Von: „access“)

mtime speichert den Zeitpunkt der letzten Änderung des Inhalts einer Datei oder eines Verzeichnisses. (Von „modification“)

ctime speichert den Zeitpunkt der letzten Änderung an den Metadaten. (Von „change“) Dieser Zeitpunkt stimmt häufig mit *mtime* überein, da eine Änderung des Inhalts auch eine Änderung der Metadaten nach sich zieht.

size Die Dateilänge in Bytes.

blocks Die Anzahl der Blöcke die von einer Datei belegt werden.

blksize Die Größe eines Blocks in Bytes.

Die Methode `getattr(struct vfsmount *mnt, struct dentry *dentry, struct kstat *stat)` wurde erst mit der Kernversion 2.6 eingeführt und motiviert sich aus der Anwendung bei Netzwerkdateisystemen. Die Methode wird aufgerufen, bevor die Metadaten eines Inode-Objektes ausgelesen werden und ermöglicht es dem Dateisystem die Aktualität der Metadaten sicher zu stellen. Im Fall von *CLF* werden die Metadaten beim Metadatenmanager abgerufen und in der übergebenen `kstat`-Struktur zurück gegeben. Der Parameter `dentry` identifiziert die Inode[1] und der zusätzliche Parameter `mnt` erlaubt es der Methode auch allgemeine Informationen über das Dateisystem direkt auszulesen.

Nachrichtentyp `CLF_RQ_NOTIFY_CHANGE`: Um Änderungen an den Metdaten an den Metadatenmanager weiterzuleiten, verwendet *CLF* den Nachrichtentyp `CLF_RQ_NOTIFY_CHANGE`. Wenn das VFS Änderungen an den Metadaten einer Inode vornimmt, wird die Methode `setattr(struct dentry *dentry, struct iattr *attr)` in dem entsprechenden Inode-Objekt aufgerufen. Wie bei `getattr()` identifiziert `dentry` die Inode, während der zweite Parameter eine Struktur ist, die einerseits die geänderten Metadaten, aber andererseits auch ein Bit-Feld enthält, das anzeigt, welche der Attribute sich geändert haben. Ändern können sich: `mode`, `uid`, `gid`, `size`, `atime`, `mtime` und `ctime`. Wird beispielsweise eine Datei verkürzt (`truncate()`), dann wird neben der `truncate()`-Methode des Inode-Objektes auch `setattr()` aufgerufen, um anzuzeigen, dass sich Größe (`size`) und die Zeitpunkte `atime`, `mtime` und `ctime` geändert haben.

Nachrichtentyp `CLF_RQ_LOOKUP`: Die für die Pfadauflösung wichtigste implizite Anfrage ist `CLF_RQ_LOOKUP`. Sie wird sowohl in der Methode `lookup()` der Inode-Objekte, als auch in der Methode `d_revalidate()` der Dentry-Objekte verwendet. Beide Methoden spielen eine zentrale Rolle bei der schrittweisen Abarbeitung einer Pfad-Zeichenkette, wie sie in Abschnitt 3.5 beschrieben wurde. Die Anfrage an den Metadatenmanager stellt fest, ob sich in einem Verzeichnis eine Datei oder ein anderes Verzeichnis mit einem bestimmten Namen befindet und entspricht damit exakt der von der `lookup()` Methode zu implementierenden Funktionalität.

[1] Dentry- und Inode-Objekte sind miteinder verknüpft.

Die Methode `lookup(struct inode *dir, struct dentry *dentry, struct nameidata *nd)` repräsentiert einen einzelnen Schritt im Pfadauflösungsprozess und kommt nur dann zum Einsatz, wenn der gesuchte Eintrag nicht im Dentry-Cache gefunden werden konnte. Der Parameter `dir` verweist auf das Inode-Objekt des Verzeichnisses in dem gesucht wird und `dentry` auf ein von der Pfadauflösung angelegtes Dentry-Objekt, das mit dem Namen des gesuchten Eintrags initialisiert wurde. Der Parameter `nd` gehört zu der Auflösungs-Operation und speichert neben dem zuletzt gefunden Eintrag auch die Parameter der Suche. Es ist Aufgabe der Dateisystem-spezifischen Implementierung dieser Methode, festzustellen, ob der Verzeichniseintrag existiert und falls dem so ist, ein Inode-Objekt anzulegen und durch einen Aufruf der VFS-Funktion `d_add(struct dentry*, struct inode*)` die Verbindung zwischen beiden Objekten herzustellen. Das VFS erkennt daran, dass der Eintrag existiert und führt die Pfadauflösung entsprechend fort.

Auch die Methode `d_revalidate(struct dentry *dentry, struct nameidata *nd)` wird bei der Pfadauflösung verwendet. Sie wird von der VFS-Schicht aufgerufen, wenn der gesuchte Eintrag im Dentry-Cache gefunden wurde und soll sicherstellen, dass der Eintrag im Cache noch aktuell ist (siehe auch Abschnitt 3.5.2). Die Implementierung dieser Methode in *CLF* benutzt dazu wiederum eine Anfrage vom Typ `CLF_RQ_LOOKUP`, wobei sie zunächst zu dem übergebenen Dentry-Objekt das Verzeichnis ermittelt in dem das Objekt enthalten ist. Anschließend wird mit der Anfrage sichergestellt, dass der Eintrag in dem Verzeichnis noch existiert und das assoziierte Inode-Objekt das gleiche ist.

Das Inode-Objekt kann sich bei folgendem Szenario ändern: Ein Verzeichniseintrag wird im Dentry-Cache gespeichert. Von einem anderen Knoten wird diese Datei bzw. dieses Verzeichnis gelöscht und anschießend eine neue Datei oder ein neues Verzeichnis unter dem Namen des zuvor gelöschten erstellt. In diesem Fall würde die Anfrage an den Metadatenmanager die Existenz des Verzeichniseintrags bestätigen, aber die `revalidate` Methode von *CLF* muss in der Lage sein, festzustellen, dass sich hinter diesem Eintrag eine andere, als die zuvor bekannte Inode verbirgt und dann das alte Inode-Objekt durch das neue ersetzen. Unter anderem um dies festzustellen werden in *CLF Generationsnummern* eingesetzt.

5.2.3.3 *Generationsnummern*

Das VFS benutzt zur Identifikation von Inodes eine innerhalb des Dateisystems eindeutige Nummer, die so genannte *Inode-Nummer*. Bei lokalen Dateisystemen ist dies ausreichend, da sie volle Kontrolle über den Zustand jedes Inode-Objektes und den zugehörigen Dentry-Objekten besitzen. Anhand der *Inode-Nummer* ist es in Clusterdateisystemen nicht möglich, ein Dateisystemobjekt eindeutig zu identifizieren, weil sich das über die *Inode-Nummer* referenzierte Objekt ändern kann.

Diese Problematik tritt beispielsweise bei der Pfadauflösung auf. Die Pfadauflösung ist ein schrittweises Verfahren, bei dem eine Komponente des Pfades nach der anderen durchsucht wird. In lokalen Dateisystemen ist sie mit Hilfe von Abschließmechanismen so implementiert, dass sie praktisch atomar abläuft. Das bedeutet, dass nach jedem Schritt der Auflösung das zuletzt gefundene Inode-Objekt abgeschlossen wird und daher nicht gelöscht oder verschoben werden kann. Dies ist bei einem Netzwerkdateisystemen zwar theoretisch auch mög-

lich, würde aber einen erheblichen Kommunikationsaufwand erfordern, den die Dateisysteme üblicherweise nicht in Kauf nehmen. Beispiele dafür sind sowohl NFS, wie auch PVFS.

Wenn aber die einzelnen Pfadkomponenten nicht abgeschlossen werden, kann es durch Seiteneffekte während der Pfadauflösung zu einem Verhalten kommen, das nicht gewünscht ist: Wird nach einer Lookup-Anfrage das gefundene Verzeichnis von einem anderen Knoten gelöscht, dann ist das Ergebnis der Anfrage ungültig geworden. Befindet sich der anfragende Knoten aber mitten in einer Pfadauflösung, dann wird er im nächsten Schritt, also bei seiner nächsten Anfrage, die erhaltene *Inode-Nummer* als Identifikation für das Verzeichnis verwenden, in dem gesucht werden soll. Zu einem unerwünschten Verhalten kommt es in dem Fall, dass von einem anderen Knoten aus ein neues Verzeichnis an einem anderen Ort erstellt wird, das die gleiche *Inode-Nummer* erhält, wie das zuvor gelöschte Verzeichnis. In diesem Fall würde die Pfadauflösungkette gestört. Der Klient würde in einem falschen Verzeichnis seine Suche fortsetzen.

Die beschriebenen Probleme sind darauf zurück zu führen, dass die *Inode-Nummern*, die zur Identifizierung der Inodes dienen, wiederverwendet werden. Wenn zwischen zwei Anfragen eines Knotens, das durch eine Inode repräsentierte Dateisystemobjekt von einem anderen Knoten gelöscht wurde und in der Folge ein neues Objekt die mittlerweile unbenutzte Nummer erhält, kommt es zu dem ungewollten Verhalten. Diese Problematik kann bei allen Anfragen auftreten, die eine *Inode-Nummer* als Parameter einer Anfrage verwenden, was bei einem Großteil der Anfragen der Fall ist: `CLF_RQ_LOOKUP`, `CLF_RQ_READ_INODE`, `CLF_RQ_NOTIFY_CHANGE`, `CLF_RQ_UNLINK`, `CLF_RQ_MKDIR`, `CLF_RQ_RMDIR`, `CLF_RQ_RENAME` und `CLF_RQ_READDIR`.

Zur Lösung dieses Problems kommen in *CLF* so genannte *Generationsnummern* zum Einsatz. Eine *Generationsnummer* behebt die Schwierigkeiten durch die Wiederverwendung von *Inode-Nummern* dadurch, dass eine Inode über eine Kombination aus *Inode-Nummer* und *Generationsnummer* identifiziert wird. Wird ein Inode-Objekt mit einer bestimmten Nummer gelöscht und anschließend wieder neu instanziiert, wird die *Generationsnummer* hochgezählt, so dass ein Klient, der beim Metadatenmanager die alte Inode über die *Inode-Nummer* und die alte *Generationsnummer* anfordert nicht fälschlicherweise Daten der neuen Inode, sondern eine Fehlermeldung erhält. Die Verwendung von *Generationsnummern* ist ein Technik, die auch in NFS und in ähnlicher Form in beinahe jedem Netzwerkdateisystem zum Einsatz kommt.

Die Verwendung von *Generationsnummern* bei Identifikation von Inodes macht die beschriebenen Schwierigkeiten bei der Pfadauflösung entdeckbar, auch wenn dadurch die Semantik der Auflösung leicht verändert wird. Wird während der Suche ein gefundenes Verzeichnis durch ein gleichnamiges anderes ersetzt, dann muss die Pfadauflösung von *CLF* im nächsten Schritt mit einem Fehler abbrechen. Im Vergleich dazu würde die lokale Pfadauflösung die Ersetzung des Verzeichnisses so lange verzögern, bis die Pfadauflösung beendet ist. Trotz dieser Abweichung bieten Generationsnummern die bestmögliche Approximation an die lokale Semantik und sind Standard in verteilten Dateisystemen.

5.2.4 Kommunikation mit den Daten-Servern

Von dem Metadatenmanager kann der Klient die Struktur des Dateisystems und Informationen über Dateien und Verzeichnisse erfragen. Der eigentliche Inhalt

der Dateien ist aber auf den Daten-Servern gespeichert. Wie in Abbildung 5.1 (und Abbildung 5.3) dargestellt, besitzt ein *CLF* Dateisystem im Allgemeinen mehrere Daten-Server, die jeweils nur einen Teil der Daten speichern.

Dateien werden intern in Blöcke unterteilt, die reihum auf den Daten-Servern gespeichert werden. Bei *CLF* entscheidet der Metadatenmanager für jede Datei auf welchem Daten-Server der erste Block gespeichert wird um eine möglichst gleichmäßige Auslastung zu erreichen. Die Information auf welchem Daten-Server der erste Block liegt ist Teil der gespeicherten Metadaten.

Die Kommunikation mit den Daten-Servern findet nur in den Einschubmethoden `read()` und `write()` der File-Objekte statt. Die Parameter dieser Methoden sind ein Verweis auf das File-Objekt, der Speicherbereich, der gelesen (beziehungsweise geschrieben) werden soll und der Offset innerhalb der Datei. Die Kommunikation zwischen Klient und Daten-Server orientiert sich an den Blockgrenzen, daher berechnen die Methoden zunächst, welche Daten-Server kontaktiert werden und wieviele Blöcke gesendet oder empfangen werden müssen.

Durch den Einsatz von *Paradis-Net* Kooperationen ist es möglich, dass die Anfragen an alle Daten-Server abgeschickt werden können, ohne dass auf die Antworten einzeln gewartet werden muss. Hier kommt das in Abschnitt 4.4.3 bzw. 4.6.2 vorgestellte Verteilen/Sammeln-Kommunikationmuster zum Einsatz. Beim Lesen wird zunächst eine Kooperation definiert, die den Zielpuffer so aufteilt, dass die Antworten der einzelnen Daten-Server an die entsprechenden Positionen geschrieben werden. Dann werden individuelle Anfragen an die beteiligten Daten-Server geschickt und gewartet, bis alle Daten eingetroffen sind. Das Empfangen der Daten und das Schreiben in den Puffer wird anhand der Kooperationsdefinition von der *Paradis-Net*-Bibliothek im Hintergrund erledigt. Tritt während des Schreibens ein Fehler in einem der Daten-Server auf, so schickt dieser, unabhängig von der laufenden Kooperation, eine Fehlermeldung an den Klienten, die auch die Kooperationsmarke enthält. Mit Hilfe dieser Marke wird der an der Kooperation wartende Faden aufgeweckt, entdeckt die Fehlersituation und meldet den Fehler an das VFS zurück.

Beim Schreiben von Daten kommt eine phasenweises Protokoll zum Einsatz: In jeder Phase wird an jeden Daten-Server nur ein Block gesendet. Dadurch soll erreicht werden, dass die Daten-Server gleichmäßig stark belastet werden. Der Klient verschickt die Datenblöcke daher reihum an die Daten-Server und versendet erst dann einen weiteren Block an einen Server, wenn er in der Zwischenzeit auch einen an die anderen geschickt hat. Die Daten-Server erhalten mit jedem Block Informationen zu dem Block und der gesamten Schreiboperation. Hat ein Daten-Server alle für ihn bestimmten Blöcke erhalten schickt er eine Bestätigung an den Klienten. Tritt einen Fehler in einem Server auf, schickt dieser eine Meldung an den Klienten, der daraufhin die Operation abbricht und den Fehler per Rückgabewert an das VFS meldet. Der Klient definiert auch für diese Operation eine Kooperation. Sie ist dann beendet, wenn von jedem involvierten Daten-Server eine Nachricht eingetroffen ist.

Die Kommunikationsprotokolle stellen eine vereinfachte Variante der Protokolle dar, die in der für parallele Anwendung optimierten Version von *CLF* zur Anwendung gekommen kommen sind (siehe [44]).

5.2.5 Zusammenfassung

Dieser Abschnitt hat Entwurf und einige Details der Implementierung des Linux-Kern-Moduls beschrieben, das in *CLF* auf Klientenseite verwendet wird. Dieses Modul implementiert nach den Regeln des VFS ein Dateisystem, das sich, wie jedes andere Linux-Dateisystem montieren und benutzen lässt.

Besonderes Augenmerk wurde dabei auf die verwendeten Kommunikationsprotokolle und die Problematik der Konsistenzerhaltung in einem Netzwerkdateisystem gerichtet.

5.3 Metadatenmanager

In den vorangegangen Abschnitt wurden die Aufgaben des Metadatenmanagers aus Sicht des Klienten betrachtet. An dieser Stelle wird beschrieben, wie der Metadatenmanager diese Aufgaben erfüllt. Dazu wird sein modularer Aufbau beschrieben, bis hinunter zu Implementierungsdetails der einzelnen Module, die größtenteils im Metadaten-Surrogat (Abschnitt 5.4) wieder verwendet werden.

Es folgt nun zunächst eine Übersicht über die Architektur des Metadatenmanagers, daran schließt sich die Beschreibung der einzelnen Module an.

5.3.1 Übersicht

Der Metadatenmanager ist als Benutzerebenen-Prozess implementiert und stellt das Zentrum eines *CLF* Dateisystems dar, welches zudem noch aus Klienten (Abschnitt 5.2), Daten-Servern (Abschnitt A) und Metadaten-Surrogaten (Abschnitt 5.4) besteht. Der Metadatenmanager agiert gegenüber den Klienten als Dienstgeber und gegenüber den Daten-Servern als Auftraggeber.

Neben der Verwaltung der Metadaten, ist der Metadatenmanager auch für die Konfiguration des Dateisystems zuständig. Er befindet sich beim Start zunächst in einem *Konfigurationsmodus* und liest Konfigurationsdateien ein, welche die Verteilung der Daten-Server und Metadaten-Surrogate auf die Knoten des Clusters, sowie deren Konfiguration speichern. Diese Informationen sendet er an die Daten-Server und Surrogate und geht anschließend in den *Dienstmodus* über.

Die Konfigurationsdateien befinden sich in einem Verzeichnis des lokalen Dateisystems, das der Metadatenmanager auch dazu verwendet die Metadaten abzulegen. Im Dienstmodus verarbeitet der Metadatenmanager sequentiell die Anfragen der Klienten, die in Tabelle 5.2 und Tabelle 5.1 aufgelistet sind. Die Behandlungsmethoden werden durch *Paradis-Net* aufgerufen, welches auch die Serialisierung der Anfragen vornimmt. Jede Anfrage wird von einer Behandlungmethode komplett abgearbeitet, bevor die nächste aufgerufen wird. Dadurch ist die lokale Konsistenz der Metadaten gesichert.

Abbildung 5.4 zeigt den modularen Aufbau des Metadatenmanagers. In der obersten Schicht ist die eigentliche Funktionalität des Metadatenmanagers zu finden: Die Behandlungsmethoden der Anfragen, die Initialisierungsroutinen des Dateisystems und die Funktionen zur Manipulation der Metadaten (*Metadata Server*). Bei der Implementierung dieser Funktionalität greift die Schicht auf mehrere Module zurück.

Die Datenstrukturen zur Speicherung von Inode-Objekten und Verzeichnissen werden von spezialisierten Modulen (*Inode Cache* und *Directory Contents*)

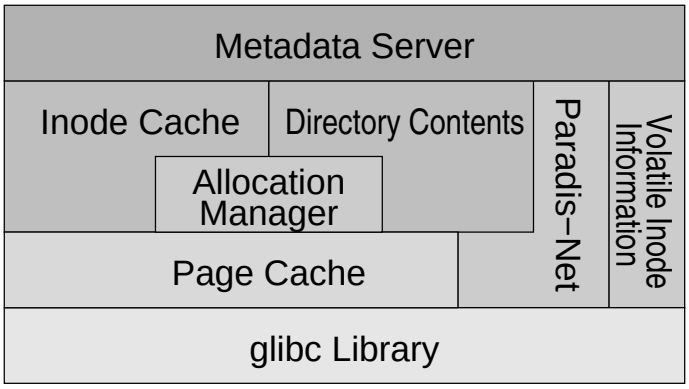


Abbildung 5.4: Modularer Aufbau des *CLF* Metadatenmanagers

definiert und persistent im lokalen Dateisystem abgelegt. Dazu verwenden beide Module ein weiteres Modul, das die verwendeten Dateien transparent in den Speicher abbildet und durch die Implementierung eines Seiten-Caches in der Benutzerebene die Zahl der Systemaufrufe vermindert (*Page Cache*).

Das Modul *Allocation Manager* wird von der Inode- und der Verzeichnisverwaltung verwendet und verwendet seinerseits das *Page Cache*-Modul zur Vermeidung von Systemaufrufen. Es implementiert ein persistentes Bitfeld, das einerseits die Benutzung von Inodes und andererseits die Verwendung von Verzeichnis-Seiten speichert. Die gesamte Netzwerkkommunikation des Metadatenmanagers basiert auf der *Paradis-Net* Bibliothek, wobei die Verzeichnisverwaltung (*Directory Contents*) sie direkt zum Versenden von Verzeichnissen verwendet.

Als letztes dient das Modul *Volatile Inode Information* der Speicherung von flüchtigen Daten über die Inodes des Dateisystems, wie beispielsweise die Information, welche Datei momentan von welchem Knoten geöffnet ist.

5.3.2 Die Seitenverwaltung (*Page Cache*)

CLF verwendet zur Speicherung aller persistenten Daten die Möglichkeit des Linux-Kerns, mittels der Funktion `mmap()` Dateien auf den Hauptspeicher abzubilden und diesen Speicher Prozessen in der Benutzerebene zur Verfügung zu stellen.

Die Abbildung ist effizienter als die Benutzung der `read()` und `write()` Funktionen, da der Kern durch Aufruf von `mmap()` seine internen Puffer in die Benutzerebene einblendet, wodurch Zwischenspeicherung der Daten in einem weiteren Speicherbereich eingespart wird. Die Speicherverwaltung des Kerns kann feststellen, wenn der Prozess schreibend auf den Speicher zugreift und aktualisiert in diesem Fall die Datei. Weiterhin werden größere eingeblendete Speicherbereiche nicht sofort geladen, sondern erst bei Bedarf, das heißt sie werden erst bei lesendem oder schreibendem Zugriff auf den Teilbereich von der Festplatte gelesen.

Die Funktion `mmap()` besitzt allerdings Einschränkungen, die hauptsächlich durch die Technik des Einblendens von Kern-Puffern bedingt ist. Zum einen

können Dateien nur Seiten-weise eingeblendet werden. Das bedeutet, dass der Versatz und die Länge eines Dateiausschnitts Vielfache der vom Kern verwendeten Seitengröße sein müssen. Die Seitengröße auf i386 Rechnern beträgt 4 KB, auf den für die Evaluation von *CLF* verwendeten Itanium2 Rechnern beträgt sie 16 KB. Zum anderen ist es mit dieser Funktion nicht möglich Dateien zu verlängern (d.h. sie über ihr aktuelles Ende hinaus zu beschreiben).

Aufgrund der ersten Einschränkung ist diese Technik besonders für die Verwaltung von Datensätzen fester Größe geeignet: Den Inode-Daten. Der Nachteil dieser Vorgehensweise ist die interne Fragmentierung, die entstehen kann, wenn die Größe einer Speicherseite kein Vielfaches der Größe eines Inode-Objektes ist. Dieser Nachteil wird allerdings durch die vielfach höhere Geschwindigkeit beim nicht-sequentiellen Zugriff auf die Datei aufgehoben.

Das *Page Cache* Modul wird mit dem Namen der Datei, aus der die Seiten eingeblendet werden sollen, und der Größe des selbst verwalteten Caches von Seiten initialisiert. Das Modul vereinfacht die Verwendung der `mmap()` Funktion so weit, dass der Benutzer mit Hilfe der Funktion `pc_load_page()`, unabhängig von der Länge der Datei, einzelne Seiten aus in den Speicher abbilden kann. Die Funktion blendet jeweils eine ganze Seite ein, die über einen Seitenindex (`mmap()` benutzt einen Byte-Index) angefordert wird; bei Bedarf verlängert das Modul die Datei. Die Funktion `pc_unload_page()` signalisiert dem Modul, dass eine bestimmte Seite nicht mehr benötigt wird. Durch diese Schnittstelle ist es möglich, zu jedem Zeitpunkt eine gewisse Zahl an Seiten innerhalb des Moduls vorzuhalten und mit Hilfe eines LRU-Verdrängungsalgorithmus Systemaufrufe zu sparen.

Die Module *Allocation Manager*, *Inode Cache* und *Directory Contents* basieren vollkommen auf der Funktionalität des *Page Cache* und konnten so ohne explizite EA-Aufrufe zum Laden oder zur persistenten Sicherung der von ihnen veränderten Metadaten implementiert werden. Sie benutzen die Funktion `pc_load_page()`, um einen Zeiger auf eine Seite zu erhalten, lesen die Daten aus oder manipulieren sie und melden dem Modul anschließend durch den Aufruf von `pc_unload_page()`, dass sie ihre Interaktion beendet haben und die Speicherseite bei Bedarf wieder freigegeben werden kann. Gleichzeitig sorgt das Betriebssystem im Hintergrund dafür, dass die im Speicher veränderten Seiten in die Dateien auf der lokalen Festplatte zurückgeschrieben werden.

5.3.3 Benutzt oder nicht? (*Allocation Manager*)

Das Modul *Allocation Manager* kommt sowohl bei der Verwaltung der Inodes, wie auch bei der Verzeichnisverwaltung zum Einsatz. Im Prinzip handelt es sich um nichts anderes als ein Bit-Feld, mit dem gespeichert wird, ob ein Element eine bestimmte Eigenschaft hat, oder nicht. Darüber hinaus ist es möglich, über eine Funktion das niedrigste Element zu finden, dessen Bit nicht gesetzt ist.

Konkret bedeutet das im Falle des Inode-Speichers, dass das Bit anzeigt, ob die entsprechende Inode momentan in Gebrauch ist oder nicht. Wird eine neue Inode angelegt, liefert das Modul die niedrigste bisher nicht benutzte Inode-Nummer zurück (d.h. den Index des niedrigsten, nicht gesetzten Bits) und markiert diese Inode-Nummer gleichzeitig als benutzt (d.h. setzt das Bit auf den Wert 1). Diese Eigenschaft könnte auch in den Metadaten selbst abgespeichert werden, mit Hilfe des *Allocation Manager*-Moduls ist aber eine besonders effiziente Suche nach freien Inode-Nummern möglich, die zusätzlich noch in einem

weiteren Modul zur Anwendung kommt.

Das Modul erlaubt neben dem Reservieren der niedrigsten noch freien Nummer auch die Zuteilung von bestimmten Nummern. Dies wird beispielsweise bei der Inode-Verwaltung dazu verwendet, die ersten 10 Inode-Nummern zu reservieren.

Der *Allocation Manager* benutzt das im vorangegangenen Abschnitt beschriebene Modul zur Seitenverwaltung, um das Bit-Feld persistent in einer Datei abzuspeichern. Das ermöglicht es dem Modul, intern nur mit Hauptspeicherseiten zu operieren und die Speicherung der Informationen dem Automatismus der Speicherabbildung zu überlassen. Das Bit-Feld wird dazu über die Speicherseiten verteilt, wobei jede Seite zusätzliche Informationen speichert, die das Auffinden von nicht gesetzten Bits erleichtern. Die Komplexität dieser Operation ist im ungünstigsten Fall $O(n)$, wobei n dem Index des höchsten gesetzten Bits entspricht. Die Zusatzinformationen ermöglichen in der Regeln jedoch das Auffinden des Bits in $O(1)$.

5.3.4 Inode-Speicher (*Inode Cache*)

Der Inode-Speicher bietet für Inode-Objekte eine ähnliche Funktionalität wie die Seitenverwaltung für Speicherseiten. Über einen Index können einzelne Inodes vor ihrer Benutzung geladen und nach Benutzung wieder abgemeldet werden. Implementiert ist dies durch die Abbildung der Inode-Objekte auf Speicherseiten, die wiederum von der Seitenverwaltung zur Verfügung gestellt werden.

Sowohl der Inode-Speicher, wie auch der *Allocation Manager* werden aus zwei verschiedenen Ausführungsfäden aufgerufen. Zum einen von dem Faden, der die Anfragen der Klienten bearbeitet, zum anderen von einem Verwaltungsfaden, der Dateien löscht, sobald sie auf keinem Klienten mehr geöffnet sind und auf den Daten-Servern entfernt wurden (siehe auch Abschnitt 5.3.6). Aus diesem Grund sind beide Module durch Semaphoren gesichert.

Die Inodes auf dem Metadatenmanager bestehen aus folgenden Elementen:

Linux-Metadaten: Bei diesen Daten handelt es sich um die Metadaten, die in den Inode-Objekten des Kerns gespeichert werden: Besitzer, Änderungzeiten, Zugriffsrechte, etc. (siehe Abschnitt 5.2.3.2).

***CLF*-Metadaten** beinhalten die Metadaten, die nicht zu den Standard-Metadaten des VFS gehören, aber für den Betrieb des Dateisystems notwendig sind. Dazu gehören unter anderem die Generationsnummer (siehe 5.2.3.3) und die Informationen über die Verteilung der Datei-Daten über die Daten-Server (siehe A.3). Die *CLF*-Metadaten werden bei entsprechenden Anfragen (zum Beispiel `CLF_RQ_READ_INODE`) zusammen mit den Linux-Metadaten an die Klienten gesendet.

Verzeichnis-ID : Repräsentiert die Inode ein Verzeichnis, dann stellt die „Verzeichnis-ID“ die Verknüpfung zu dem Modul *Directory Contents* dar. Siehe folgenden Abschnitt 5.3.5.

Der Inode-Speicher profitiert von der bereits in der Seitenverwaltung implementierten LRU-Cache-Funktionalität. Dadurch muss bei zeitlicher Lokalität der Zugriffe auf eine Inode, diese Inode nicht vom Betriebssystem geladen werden, weil sie noch in einer Seite des Seiten-Cache zu finden ist. Darüber hinaus

verwaltet der Linux-Kern auch noch eigene Caches, um Festplattenzugriffe zu vermeiden, womit die Verwendung der Seitenverwaltung der Einführung einer weiteren Cache-Schicht entspricht.

5.3.5 Verzeichnisse (Directory Contents)

Der Metadatenmanager folgt bei der Speicherung der Metadaten einem Konzept, das dem der VFS-Schicht im Linux-Kern entspricht. Das bedeutet, dass die Inode-Objekte (und damit die im vorherigen Abschnitt beschriebenen Metadaten) und die Verzeichnisinformationen (der Inhalt der Verzeichnisse) getrennt verwaltet werden.

Die Inodes stehen im Dateisystem für Dateien und Verzeichnisse, besitzen aber zunächst keinen Namen. Der Name wird dadurch bestimmt, dass sie in einem Verzeichnisses unter einem Namen durch ihre Inode-Nummer referenziert werden. Die Trennung der Inode-Objekte von ihrem Namen folgt dem Beispiel des VFS aus zwei Gründen:

Zum einen ist die Speicherung und Verwaltung von Objekten mit fester Größe einfacher als die von Objekten variabler Größe. Würden die Namen der Inode-Objekte zusammen mit den übrigen Inode-Daten gespeichert, würde das die Implementierung des im vorangegangen Abschnitt vorgestellten Inode-Speicher verkomplizieren, weil für einen Zugriff auf eine Inode mindestens eine Indirektion, beispielsweise über eine Indextabelle notwendig wäre. Gleichzeitig ist der Zugriff auf eine, über ihre Inode-Nummer referenzierte, Inode ist die am häufigsten ausgeführte Aktion auf den Metadaten des Dateisystems und erfordert daher eine möglichst effiziente Implementierung. Weiterhin wäre eine kompliziertere Freispeichersammlung (engl. Garbage Collection) notwendig.

Zum anderen lassen sich die vom VFS aufgerufenen Einschubmethoden besonders leicht implementieren, wenn die Metadatenverwaltung einem ähnlichen Konzept wie das VFS-Rahmenwerk folgt.

5.3.5.1 Funktionalität und Abhängigkeit zu der Inode-Verwaltung

Die Wurzel-Inode, also die Inode, die das Wurzelverzeichnis eines *CLF*-Dateisystems darstellt, ist per Konvention die Inode mit der Nummer „10“. Aufgrund dieser Festlegung kann jeder Metadatenmanager nur ein Dateisystem verwalten.

Die Verwaltung der Verzeichniseinträge innerhalb des Metadatenmanagers orientiert sich an der *Verzeichnis-ID*, die in den Inode-Daten gespeichert wird. Sie stellt die Verknüpfung zwischen einem Verzeichnis (der Inode) und seinem Inhalt dar. Die Kennung wird als Identifikation des Verzeichnisses bei dem *Directory Contents* Modul verwendet. Bei Anfragen wie dem Hinzufügen, Löschen oder Abfragen eines Verzeichniseintrags wird das betroffene Verzeichnis über die *Verzeichnis-ID* referenziert. Bei Initialisierung eines Verzeichnisses generiert die Verzeichnisverwaltung eine Kennung für das Verzeichnis, mit deren Hilfe in Zukunft der Verzeichnisinhalt referenziert werden kann.

Die Verzeichnisverwaltungsmodul bietet darüber hinaus Funktionen zum Hinzufügen (`dm_add_entry()`) und Löschen (`dm_rem_entry()`) von Verzeichniseinträgen, welche beim Erzeugen und Löschen von Dateien und Verzeichnissen in den Bearbeitungsfunktionen der Anfragen `CLF_RQ_CREATE`, `CLF_RQ_MKDIR`, `CLF_RQ_UNLINK` und `CLF_RQ_RMDIR` verwendet werden. Ein Verzeichniseintrag ist ein Tupel, bestehend aus dem Dateinamen und der Inode-Nummer.

Die wichtigste Aufgabe, die das Modul auszuführen hat, ist allerdings die Suche nach Dateien, die von den Anfragen des Typs `CLF_RQ_LOOKUP` (siehe Tabelle 5.2) gefordert wird. Die Funktion `dm_find_entry` erfüllt diese Funktionalität; sie erwartet die *Verzeichnis-ID* und den Namen des Eintrags als Parameter gibt die Inode-Nummer zurück, falls der Eintrag gefunden wurde.

5.3.5.2 Erweiterbares Hashen (Extendible Hashing)

Bei der Implementierung des Moduls kam eine Variante des aus dem Datenbankbereich bekannten *Erweiterbaren Hashen* (*EH*, engl. Extendible Hashing, siehe [31, 73]) zum Einsatz. Neben dem erweiterbaren Hashen wurde dieses Modul versuchsweise auch unter Benutzung von B-Bäumen [20] implementiert. Obwohl beide Verfahren auf die Verwendung von Block- oder Seiten-weise angeordnetem Speicher zugeschnitten sind, besitzt das *EH* zwei Vorteile. Zum einen sind bei der Suche nach einem Eintrag immer nur zwei Seitenzugriffe notwendig, während bei B-Bäumen, je nach Größe des Verzeichnisses mehrere Seiten von der Festplatte geladen werden müssen. Zum anderen ist es für die Beantwortung einer `CLF_RQ_READDIR`-Anfrage möglich, ohne Kopieroperationen den gesamten Verzeichnisinhalt an den Klienten zu schicken.

Grundidee des Erweiterbaren Hashens Das *erweiterbare Hashen* ist eine externe Suchmethode, die 1978 von R. Fagin et al. entwickelt wurde. Das Suchen in dieser Datenstruktur erfordert bei typischen Anwendungen zwei Plattenzugriffe für jede Suche, und ermöglicht gleichzeitig ein effizientes Einfügen. Die Daten werden in Seiten gespeichert, die auf einem Plattenspeicher abgelegt werden. Bei *CLF* übernimmt das *Page Cache*-Modul die Abbildung des Festplatten- (genauer: Datei-) Inhalts in den Speicher.

Innerhalb einer *Daten-Seite* sind die Daten nach dem Suchschlüssel sortiert. Wird eine Seite so voll, dass sie keine weiteren Einträge mehr aufnehmen kann, wird sie auf zwei Seiten aufgespalten. Zusätzlich zu den *Daten-Seiten* gibt es eine (oder mehrere) *Index-Seiten*, die einen Index enthalten, mit dessen Hilfe die Datenseite gefunden werden kann, die den mit unserem Suchschlüssel übereinstimmenden Eintrag enthält. Im Allgemeinen sieht die Datenstruktur auch mehrere Datensätze mit gleichem Suchschlüssel vor, im Fall von Verzeichnissen ist jedoch jeder Schlüssel (= Dateiname) einzigartig.

Um eine gleichmäßige Verteilung der Schlüsselwerte zu erreichen, werden die Schüssel mit einer Hash-Funktion auf einen Wert abgebildet, dessen führende Bits möglichst gleichverteilt sind. Die Suche nach einem Datensatz erfolgt in zwei Schritten: Zunächst wird innerhalb einer *Index-Seite* nach der *Daten-Seite* gesucht, die den Schlüssel enthält. Der Schlüssel wird dazu mit einer Hash-Funktion auf einen Wert abgebildet, dessen Bitfolge als Index einer in der *Index-Seite* gespeicherten Tabelle verwendet wird. Aus dem Tabelleneintrag kann nun abgelesen werden, welche *Daten-Seite* den gesuchten Datensatz enthält. Innerhalb der *Daten-Seite* kann der gesuchte Datensatz durch eine binäre Suche gefunden werden, da die Einträge nach dem Schlüssel (oder dessen Hash-Wert) sortiert sind.

Das Einfügen eines Eintrags in die Datenstruktur erfordert zunächst wiederum das Auffinden der *Daten-Seite*, die dem Schlüssel (genauer: dem Hash-Wert) des Schlüssels zugeordnet ist. Falls auf der *Daten-Seite* genug Platz frei ist, wird der Datensatz dort eingefügt, andernfalls wird die *Daten-Seite* in zwei

Teile gespalten. Dazu werden die Datensätze, die sich in einem weiteren Bit des Hash-Wertes unterscheiden, auf eine neue angelegte Seite verschoben. Weiterhin muss die Tabelle auf der *Index-Seite* angepasst werden, so dass korrekt auf die neue und alte *Daten-Seite* verwiesen wird.

Die *Index-Seite* enthält eine Tabelle deren Länge einer Zweier-Potenz entspricht und die über die höchstwertigen Bits des Hash-Wertes des Schlüssels indiziert wird. Die Größe dieser Tabelle ist dynamisch und passt sich der Zahl der referenzierten *Daten-Seiten* an. Daher ist es in bestimmten Fällen notwendig, die Tabellengröße zu verdoppeln, um nach Aufspaltung einer *Daten-Seite*, die neu entstandene *Daten-Seite* zu referenzieren.[2] Bei einer Verdoppelung des Index erhöht sich die zur Indizierung verwendete Bitzahl um eins.

Das Löschen von Einträgen erfolgt analog zum Einfügen, wobei hier im Anschluss nicht geprüft wird, ob Seiten aufgeteilt werden, sondern ob sich eine Verschmelzung von *Daten-Seiten* lohnt. Wird die Verschmelzung durchgeführt, muss sie entsprechende Änderungen an der *Index-Seite* nach sich ziehen. Dies kann unter bestimmten Bedingungen auch zu einer Verkleinerung der Tabelle führen.

Die folgenden Abschnitte beschreiben die konkrete Implementierung des *Erweiterbaren Hashens* im Modul *Directory Contents*. Für eine ausführliche Diskussion des Algorithmus verweise ich auf die zu Beginn dieses Abschnitts angegebenen Referenzen.

Die konkrete Implementierung unterscheidet sich von dem klassischen Algorithmus durch eine zusätzliche Indizierung der Datensätze innerhalb der *Daten-Seiten*, die durch die flexible Größe der Datensätze motiviert wird. Eine weitere Besonderheit ist die Serialisierung aller in einem Verzeichnis enthaltenen Daten, die bei der Abfrage des Verzeichnisinhalts erforderlich ist. Um diese Besonderheiten beschreiben zu können, ist eine detaillierte Besprechung der Datenstruktur notwendig.

5.3.5.3 Hash-Wert

Der in den folgenden Abschnitten verwendete Begriff *Dateiname* ist als Synonym für den Namen eines Verzeichniseintrags zu verstehen und schließt damit auch Verzeichnisnamen ein.

Der Dateiname stellt in den Verzeichnissen den Suchschlüssel dar. Anfragen an das Modul sollen feststellen, ob ein bestimmter Dateiname in einem Verzeichnis existiert und im Falle der Existenz die Inode-Nummer des Eintrags bestimmen. Als Hash-Wert wird ein 16 Bit Wert benutzt, der durch folgende Iteration aus dem Dateinamen berechnet wird:

```
uint16_t full_name_hash(const char *name) {
  uint16_t hash = 0;
  while (*name)
    hash = partial_name_hash(*name++, hash);
  return hash;
}
```

Der Hash-Wert berechnet sich damit aus der Anwendung der Funktion `partial_name_hash(char*, uint16_t)` auf jedes Zeichen der Zeichenkette, wobei das letzte Ergebnis des Funktionsaufrufs bei jedem weiteren Schritt der

[2] Diese Erweiterung des Index ist namensgebend für den Algorithmus.

Funktion als Parameter übergeben wird. Die Funktion `partial_name_hash` ist folgendermaßen definiert:

```
uint16_t partial_name_hash(uint16_t c, uint16_t prevhash) {
  return (prevhash + (c << 4) + (c >> 4)) * 11;
}
```

Die Berechnung des Hash-Wertes wurde hier als C-Quelltext angegeben, da eine abstrakte Definition aufgrund der vorzeichenlosen 16-Bit Arithmetik unangemessen kompliziert ausgefallen wäre. Die Berechnungsvorschrift wurde in Anlehnung an die im Reiser Dateisystem[55] verwendete *R5 Hash*-Funktion definiert.

5.3.5.4 Such-Algorithmus

Die Verzeichnisverwaltung basiert auf der Seitenverwaltung und speichert ihre Datenstrukturen auf den Seiten, die von dieser zur Verfügung gestellt werden. Die in den Inode-Daten eines Verzeichnisses abgelegte Verzeichnis-ID verweist dabei auf eine *Index-Seite*. Die *Index-Seite* enthält Verweise auf *Daten-Seiten*, welche die Verzeichniseinträge enthalten.

Die Suche ist die zentrale Funktion des Moduls. Anhand eines Beispiels verfolgen wir nun die Suche nach dem Dateinamen „file.txt“ in dem Verzeichnis mit der Verzeichnis-ID (*Index-Seite*) „21“. Die relevanten Teile der Datenstruktur sind in Abbildung 5.5 dargestellt.

1. Der Hash-Wert des Dateinamens wird berechnet: `full_name_hash("file.txt")= 44019`. Der Wert „44019“ ist in Binärdarstellung „1010 1011 1111 0011“.

2. Wie in Abschnitt 5.3.5.2 beschrieben wurde, ist die Größe des Index flexibel. Die Anzahl der verwendeten Bits und damit die Größe des Index wird in dem letzten Eintrag (2^n) der *Index-Seite* gespeichert. (n folgt aus der Seitengröße der verwendeten Hardware-Architektur; auf i386-Rechner ist $n = 10$, auf Itanium2-Rechner ist $n = 12$.)

   Auf der Abbildung 5.5 enthält der Eintrag 2^n der *Index-Seite* den Wert 2. Dieser Wert bedeutet, dass der Index eine Länge von $2^2 = 4$ besitzt und die 2 höchstwertigen Bits (Abkürzung *HWB*) des Hash-Wertes als Index verwendet werden.

3. Die beiden HWBs des Hash-Wertes sind 10, daher ist die entsprechende *Daten-Seite* über den Indexeintrag 2 zu finden, dieser verweist auf die Seite mit der Nummer 42.

4. Eine *Daten-Seite* setzt sich aus zwei Teilen zusammen: Einem Datenteil, der vom Beginn der Seite bis zum Ende wächst und einem Indexteil, der vom Ende der Seite in Richtung des Anfangs wächst. Der Indexteil enthält die Hash-Werte der Einträge im Datenteil sowie Verweise auf diese. Er ist nach den Hash-Werten sortiert und ermöglicht damit das schnelle Auffinden von Einträgen. Die *Daten-Seite* 42 enthält 2 Einträge und auf der Suche nach dem Hash-Wert 44109 im Indexteil wird der binäre Suchalgorithmus beim zweiten Eintrag fündig. (siehe Abbildung 5.5)

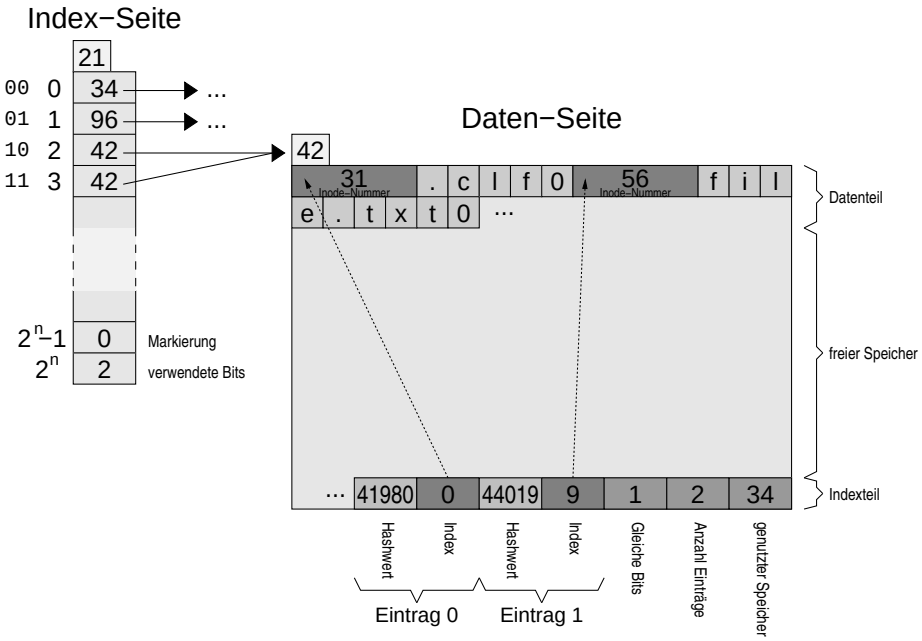


Abbildung 5.5: Beispiel einer Index- und einer *Daten-Seite* im „Directory Contents“ Modul

5. Der zu dem Hash-Wert 44109 gehörende Eintrag wird über den im Indexteil gespeicherten Byteindex 9 aufgefunden. Der Index verweist im Datenteil der Seite auf den Beginn der Inode-Nummer. Obwohl die Hash-Werte übereinstimmen, muss in einer letzten Prüfung festgestellt werden, ob der gesamte Dateiname übereinstimmt. Dies ist hier der Fall: Der Eintrag „file.txt“ wurde gefunden und besitzt die Inode-Nummer 56.

Bemerkungen zu dem Suchalgorithmus:

- Die Suche nach einem Verzeichniselement führt zu genau zwei Seitenzugriffen und damit auch zu maximal zwei Seitenfehlern.
- Die Suche besitzt eine Komplexität von $O(1)$ - unter der Annahme, dass sich alle Seiten im Speicher befinden. Im ungünstigsten Fall muss zwar eine Seite komplett durchsucht werden, aber da die Seitengröße eine Konstante ist, folgt auch in diesem Fall die beschriebene Komplexität.

  In der Praxis zeigt die durchschnittliche Laufzeit dieses Algorithmus eine große Abhängigkeit von der Verzeichnisgröße. Bei größeren Verzeichnissen steigt die Wahrscheinlichkeit für Hash-Kollisionen und damit erfordert der Algorithmus auf der Suche nach dem richtigen Eintrag aufwändigere Vergleiche. In Abschnitt 5.3.5.7 werden die praktischen Laufzeiten der Verzeichnis-Suche vorgestellt.
- In Schritt 4 kommt als Suchalgorithmus die Binäre Suche [74] zum Einsatz. Aufgrund der Eigenschaften des Algorithmus führt die Suche im Falle von Hash-Kollisionen aber nicht zwangsläufig zu dem gesuchten Eintrag,

da mehrere Einträge den gleichen Hash-Wert besitzen, aber nur einer dieser Einträge gefunden wird. Daher werden nach Ende der Binär-Suche nicht nur der aktuelle Eintrag, sondern auch die benachbarten Einträge untersucht, falls diese den gleichen Hash-Wert besitzen.

5.3.5.5 Einfügen und Entfernen von Einträgen

Die im letzten Abschnitt beschriebene und beispielhaft in Abbildung 5.5 dargestellte Datenstruktur ist zwar gut für die Suche nach Verzeichniseinträgen geeignet, während die Pflege relativ aufwändig ist. Der Grund für den Einsatz der Datenstruktur liegt in der Häufigkeit der Anfragen an den Metadatenmanager: Der überwiegende Teil der Anfragen (siehe auch Kapitel 6) sind vom Typ `CLF_RQ_READ_INODE` und `CLF_RQ_LOOKUP`. Dabei erfordert die `CLF_RQ_LOOKUP`-Anfrage das Suchen eines bestimmten Dateinamens in einem Verzeichnis. Demgegenüber sind das Erstellen von Verzeichnissen und Dateien, sowie das Löschen derselben relativ seltene Operationen und das spiegelt sich auch in der Implementierung der Verzeichnisverwaltung wider. Das Hinzufügen und das Löschen von Einträgen ist ungleich aufwändiger als das Suchen, auch wenn im schlechtesten Fall nur 3 Seiten verändert werden müssen.

Einfügen Das Hinzufügen eines Verzeichniseintrags zu einem Verzeichnis als Folge einer `CLF_RQ_MKDIR`- oder `CLF_RQ_CREATE`-Anfrage, verläuft in folgenden Schritten:

1. Zunächst werden alle Schritte des zuvor beschriebenen Suchalgorithmus ausgeführt. Wird dabei festgestellt, dass bereits ein Eintrag unter dem neuen Namen existiert, wird ein Fehler an die aufrufende Funktion zurück gegeben. Andrenfalls fährt der Algorithmus mit Schritt 2 fort.

2. Es wird geprüft, ob auf der *Daten-Seite* genug freier Speicher vorhanden ist, um den neuen Eintrag zu speichern. Ist dies der Fall, fährt er bei Schritt 5 fort.

3. Auf der *Daten-Seite* ist nicht genug Platz für einen weiteren Eintrag, daher muss die *Daten-Seite* auf zwei Seiten aufgeteilt werden. Der Algorithmus prüft zunächst, ob im Index der *Index-Seite* der Verweis auf eine weitere *Daten-Seite* gespeichert werden kann. Ist dies nicht der Fall, wird die Größe des Index verdoppelt. Das bedeutet, dass zur Indizierung der Seiten ein weiteres Bit verwendet wird.

   Im Fall der in Abbildung 5.5 dargestellten Seiten wäre beispielsweise eine Verdoppelung des Index bei Aufteilung der *Daten-Seite* 42 nicht notwendig, da auf diese von zwei Index-Einträgen verwiesen wird. Nach einer Aufteilung der *Daten-Seite* würde der Index 2 weiterhin auf die *Daten-Seite* 42 und der Index 3 auf die neue *Daten-Seite* verweisen, wobei die Seite 42 dann nur noch Einträge enthalten würde, deren HWBs „10“ sind und die neue Seite diejenigen mit den HWBs „11“.

4. Auf einer *Daten-Seite* befinden sich Einträge deren Hash-Werte in einer bestimmten Anztahl HWBs übereinstimmen. Die Anzahl der gleichen Bits wird Indexteil der *Daten-Seite* im Feld *GleicheBits* gespeichert. (vergleiche Abbildung 5.5)

Bei einer Aufteilung wird der Inhalt der *Daten-Seite* so auf zwei Seiten aufgeteilt, dass die Einträge, deren Hash-Werte im $(GleicheBits+1)-$ten Bit den Wert 0 enthalten auf der ursprünglichen Seite verbleiben und die mit dem Wert 1 auf eine neu Seite kopiert werden. Innerhalb einer *Daten-Seite* sind die Einträge nach Hash-Werten sortiert, daher sind die Einträge im Daten- und Indexteil bereits so angeordnet, dass sie Block-weise mit zwei Kopieroperationen auf die neue Seite übertragen werden können, wobei im Indexteil noch die Verschiebung des Datenteils an den Anfang der Seite verrechnet werden muss. Die Verwaltungsinformationen *AnzahlEinträge* und *GenutzterSpeicher* müssen auf beiden Seiten angepasst werden und der Wert von *GleicheBits* wird auf der ursprünglichen Seite um 1 erhöht und auf die neue Seite übertragen.

5. In den Schritten 1–4 wurde die *Daten-Seite* ermittelt, die den neuen Verzeichniseintrag speichern soll und sichergestellt, dass auf der Seite Platz für diesen vorhanden ist.

   Auf der *Daten-Seite* wird der Eintrag wie folgt gespeichert: Mit Hilfe einer binären Suche wird die Einfügeposition des neuen Elementes in den Einträgen des Indexteils bestimmt. Anhand der Informationen des Indexteils werden die Einträge in Datenteil so verschoben, dass sie eine Lücke bilden, in die der neue Eintrag passt; das gleiche passiert mit den Einträgen im Indexteil, wobei das Feld zum Beginn der Seite hin wächst und daher die vorderen Einträge nach vorne verschoben werden, während im Datenteil die hinteren Einträge nach hinten verschoben werden. Anschließend wird der neue Eintrag in die Lücke im Dateiteil kopiert und im Indexteil referenziert. Die Referenzen der im Dateiteil verschobenen Einträge werde aktualisiert.

Das in Schritt 5 beschriebene Verschieben der Einträge im Datenteil sorgt dafür, dass diese aufsteigend nach ihren Hast-Werten sortiert sind und ermöglicht das in Schritt 4 beschriebene Block-weise Kopieren. Dadurch ist es im Falle einer Aufteilung nicht notwendig, jeden Eintrag einzeln zu betrachten, bei Bedarf auf die neue Seite zu kopieren und dabei auch die frei gewordenen Lücken in der ursprünglichen Seite zu entfernen. Es ist auch nicht notwendig, die freigewordenen Speicherblöcke bei der Aufteilung zu kompaktieren, da sich alle verschobenen Verzeichniseinträge am Ende des Datenteils befinden und dort gelöscht werden können.

Entfernen Das Entfernen eines Verzeichniseintrags als Folge einer `CLF_RQ_RMDIR`- oder `CLF_RQ_UNLINK`-Anfrage, läuft wie folgt ab:

1. Zunächst wird der zu Beginn dieses Abschnitts beschriebene Suchalgorithmus ausgeführt um den zu löschenden Verzeichniseintrag in der zugehörigen *Daten-Seite* aufzufinden.

2. Mit Hilfe des Indexes auf der *Daten-Seite* kann der Algorithmus feststellen, welchen Bereich des Datenteils der Eintrag belegt. Die nachfolgenden Einträge werden innerhalb der Datenteils nach vorne verschoben und überschreiben dabei den Eintrag vollständig. Der Verweis auf den Eintrag wird im Indexfeld gelöscht indem die Einträge mit kleinerem Hash-Wert nach hinten verschoben werden und die nachfolgenden Einträge, deren Verweise

aufgrund der Verschiebung im Datenteil nicht mehr aktuell sind, angepasst werden. (vergleiche Abbildung 5.5)

Nach Abschluss dieses Schrittes wurde der Eintrag vollständig gelöscht. Die weiteren Schritte dienen der Pflege und Kompaktierung der Datenstruktur.

3. Nach dem Löschen eines Eintrags wird überprüft, ob die *Daten-Seite* mit ihrer *Schwester-Seite* verschmolzen werden kann. Unter einer *Schwester-Seite* versteht man die Seite, eine identische Anzahl *GleicheBits* besitzt und in den ersten $GleicheBits - 1$ Bits mit der betrachteten Seite überein stimmt. Eine Verschmelzung ist dann möglich, wenn die Summe des genutzten Speichers ($genutzterSpeicher(S_1) + genutzterSpeicher(S_2)$) beider Seiten, abzüglich des für die auf beiden Seiten vorhandenen Felder (*GleicheBits*, *AnzahlEinträge* und *genutzterSpeicher*) belegten Speichers kleiner als die Seitengröße ist.

   Obwohl eine Verschmelzung beider Seiten schon unter dieser Bedingung möglich wäre, besteht die Gefahr, dass ein auf diese Verschmelzung folgendes Hinzufügen eines Eintrags zur erneuten Aufteilung der *Daten-Seite* führen würde. Um dies zu verhindern werden zwei Seiten erst dann verschmolzen, wenn auf der verschmolzenen Seite noch 10% des Speicherplatzes ungenutzt sein werden.

   Wird keine Verschmelzung durchgeführt, ist der Algorithmus an dieser Stelle beendet.

4. Die Verschmelzung von zwei Seiten verläuft umgekehrt zur Aufteilung: Der Datenteil der Seite mit den höheren Hash-Werten (S_2) wird Blockweise hinter den Datenteil der Seite mit den niedrigeren Hash-Werten (S_1) kopiert. Der Indexteil von S_1 wird so weit nach vorne verschoben, dass der Indexteil von S_2 hineinkopiert werden kann. Anschließend werden die Indizes im ehemaligen Indexteils von S_2 angepasst, sowie die Zahl der neu hinzugefügten Elemente (*AnzahlEinträge*) und der belegte Speicher (*GenutzterSpeicher*) angepasst. Das Feld *GleicheBits* wird um den Wert 1 erniedrigt. (vergleiche Abbildung 5.5)

5. Alle Referenzen auf S_2 auf der *Index-Seite* werden durch Referenzen auf S_1 ersetzt und S_2 wird frei gegeben.

6. Auf der *Index-Seite* wird geprüft, ob eine Verkleinerung des Indexes möglich ist. Dies ist genau dann möglich, wenn alle Verweise auf *Daten-Seiten* an Indexpositionen, die sich nur im niedrigwertigstem Bit unterscheiden, gleich sind. Durch eine Verkleinerung des Index verringert sich die Anzahl der Bits über die indiziert wird um eins und die Größe der Tabelle halbiert sich.

5.3.5.6 Lesen von Verzeichnissen

Neben dem Suchen eines Dateinamens und dem Hinzufügen und Entfernen von Einträgen, ist das Auflisten des gesamten Verzeichnisinhalts die vierte Operation, die das Modul ausführen muss. Ein Klient schickt eine Anfrage vom Typ `CLF_RQ_READDIR` an den Metadatenmanager, sobald ein Verzeichnis geöffnet

wird. Als Antwort auf die Anfrage erwartet der Klient eine Liste aller Verzeichniseinträge und der zugehörigen Inode-Nummern. Der Klienten iteriert in nachfolgenden `readdir()`-Aufrufen wird über diese Liste (siehe auch Abschnitt 5.2.3.1).

Der Metadatenmanager schickt die Liste in der Form, wie sie im Datenteil der *Daten-Seiten* gespeichert wird: In Form eines großen Puffers, der eine Folge von Inode-Nummern, jeweils gefolgt von dem zugehörigen Dateinamen enthält (siehe Abbildung 5.5). Zur Beantwortung einer `readdir()` Anfrage lädt der Manager alle zu einem Verzeichnis gehörenden *Daten-Seiten* und schickt die Datenteile dieser Seiten an den Klienten.

Aufgrund des Aufbaus der *Daten-Seiten*, insbesondere aufgrund der Trennung von Daten- und Indexteil, ist für die Beantwortung dieser Anfrage keine Berechnung notwendig, sondern nur die Versendung der Datenteile aller in der *Index-Seite* referenzierten *Daten-Seiten*. Somit lässt sich die Operation aufgrund der passend gewählten Datenstruktur besonders effizient implementieren.

5.3.5.7 Performanz der Verzeichnisverwaltung

Wie bereits erwähnt, stand bei der Implementierung dieses Moduls die Performanz der `lookup()`- und `readdir()`-Operationen im Vordergrund, da diese den Großteil der Klienten-Anfragen bilden. In diesem Abschnitt werden Messergebnisse präsentiert, welche die durchschnittliche Ausführungsdauer der *Suche*-(`lookup()`), *Einfüge*- und *Lösche*-Operationen in Abhängigkeit von der Verzeichnisgröße darstellen.

Versuchsaufbau Das Modul wurde unter Linux 2.6.9 auf einem *Intel ItaniumII*-Doppelprozessor-Knoten (2x1.3GHz) mit 1 GB Hauptspeicher und einer 73 GB Ultra320 SCSI Festplatte vermessen. Die Namen der Einträge wurden zufällig unter Benutzung des Glib-Zufallsgenerators (`rand()`) erzeugt: Für jeden Namen wurde zunächst eine zufällige Länge zwischen 4 und 56 Zeichen festgelegt und anschließend wurden die Namen aus Zeichen mit einem ASCII-Wert zwischen 32 und 126 gebildet. Die Vermessung einer Schleife, die nur die zufällige Bestimmung eines Dateinamens enthält, ergab eine Schleifen- und Namensberechnungs-Dauer von $3,04\mu sec$ pro Iteration. Daher wurde in den drei nachfolgenden Abbildungen der Beginn der Zeitachse bei $3\mu sec$ gewählt. Der Anteil der Namens-Berechnung wurde allerdings nicht aus den Ergebnissen herausgerechnet.

Suchen Abbildung 5.6 zeigt die Dauer einer `lookup()`-Operation in Abhängigkeit zu der Größe des Verzeichnisses. Für die Messung wurde ein Verzeichnis erstellt und eine gewisse Menge zufällige Einträge hinzugefügt. Für jeden Datenpunkt wurden in dem Verzeichnis 50000 Suchoperationen durchgeführt, wobei für den ersten Datensatz (*existierende Dateien*), der Zufallsgenerator regelmäßig so zurückgesetzt wurde, so dass ausschließlich Dateien gesucht wurden, die in dem Verzeichnis vorhandenen sind, während für den zweiten Datensatz (*nicht-existierende Dateien*) der Zufallsgenerator weiter lief und auf diese Weise nicht im Verzeichnis vorhandene Dateinamen gebildet hat.

In der Abbildung ist zu sehen, dass die für die Suche nach einem Verzeichniseintrag notwendige Zeit im relevanten Bereich (weniger als 1000 Einträgen pro Verzeichnis) beinahe konstant bleibt. Die benötigte Zeit für die Suche nach existierenden Dateien beträgt dabei maximal $3,92\mu sec$, die für nicht-existierende

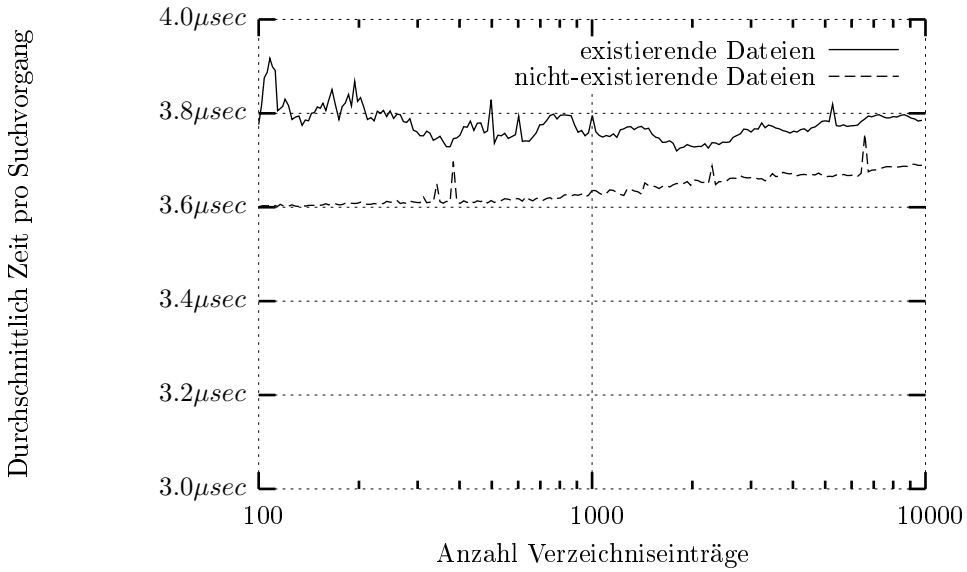


Abbildung 5.6: Dauer der Suche nach einem Verzeichniseintrag in Abhängigkeit von der Verzeichnisgröße

Dateien etwas mehr als $3,6\mu sec$. Aufgrund des verwendeten Caches sind diese Zeiten nahe an reinen Speicherzugriffen auf die verwendeten Seiten. Zu beachten ist weiterhin, dass die Zeit für die Erzeugung des Zufallsnamens nicht herausgerecht wurde.

Die leicht abnehmende Dauer der Suche nach existierenden Dateien bei steigender Verzeichnisgröße ist darauf zurückzuführen, dass bei größeren Verzeichnissen der Zufallsgenerator seltener zurückgesetzt werden muss. Der Bibliotheksaufruf zum Zurücksetzen des Zufallsgenerators dauert relativ lange und weitere Experimente haben gezeigt, dass der Kurvenverlauf ohne diesen Seiteneffekt, wie bei der Suche nach nicht-existierenden Dateien verlaufen würde.

Der Grund für die steigende Dauer der Suche nach nicht-exisiterenden Dateien bei steigender Verzeichnisgröße sind die Hash-Kollisionen, die sich bei größeren Verzeichnissen immer stärker auswirken, wie man am Verlauf des zweiten Datensatzes (Suche nach nicht-existierenden Dateien, Bereich 1000 - 10000) erkennen kann. Dass die Suche nach existierenden Dateien länger dauert als die nach nicht existierenden erklärt sich daraus, dass bei existierenden Dateien die einzelnen Zeichen des Eintrags verglichen werden müssen, während im anderen Fall meist schon der Vergleich der Hash-Werte zeigt, dass das Verzeichnis keinen Eintrag mit dem gesuchten Namen enthält. Diese Effekt schwächt sich aufgrund von Hash-Kollisionen bei größeren Verzeichnissen immer mehr ab.

Einfügen Abbildung 5.7 zeigt die Messergebnisse aus den Testläufen mit der Funktion zum Hinzufügen von Verzeichniseinträgen. Jeder Datenpunkt zeigt die durchschnittliche Zeit, die zum Einfügen eines Verzeichniseintrags notwendig war, wenn der Vorgang bei einer bestimmten Verzeichnisgröße abgebrochen wurde. Es wurde die Gesamtzeit $t_{ges}(n)$ gemessen, die für das Hinzufügen von

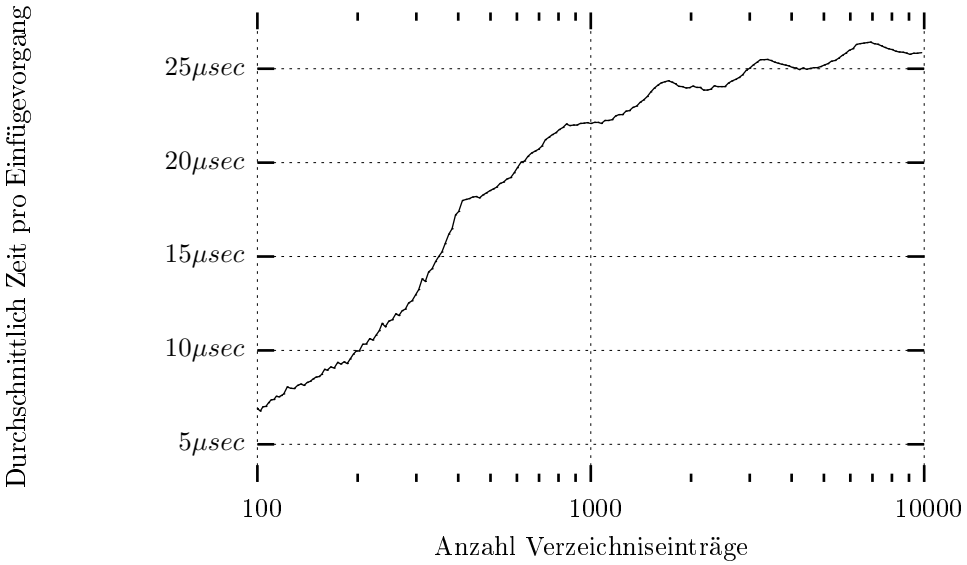


Abbildung 5.7: Dauer des Einfügens eines Verzeichniseintrags in Abhängigkeit von der Verzeichnisgröße

n Verzeichniseinträgen zu einem anfangs leeren Verzeichnis notwendig war und anschließend die durchschnittlich für das Einfügen eines Eintrags benötigte Zeit $t(n) = \frac{t_{ges}(n)}{n}$ berechnet.

Im relevanten Bereich bis 1000 Einträge beträgt die durchschnittlich für das Einfügen eines Eintrags benötigte Zeit unter $22\mu sec$, im Bereich von 1000 bis 10000 Einträgen steigt der Zeitbedarf bis maximal $26,4\mu sec$ an. Die für das Einfügen benötigte Zeit setzt sich hauptsächlich aus 2 Komponenten zusammen: Einerseits der Zeit, die für das *Auffinden des Einfügepunktes* und andererseits die Zeit, die für das *Einfügen* selbst notwendig ist.

Den Anteil der ersten Komponente entspricht im wesentlichen dem zweiten Datensatz (nicht-existierende Dateien) in Abbildung 5.6. Wie bei der reinen Suche, wächst auch hier der aufgrund der höher werdenden Wahrscheinlichkeiten von Hash-Kollisionen bei größeren Verzeichnissen, die Suchzeit. Den größeren Anteil an der Gesamtzeit stellt mit der zweiten Komponente das Einfügen des Eintrags im Daten- und Indexteil der *Daten-Seite* dar. Dabei spielt es eine wichtige Rolle, wieviel des Speichers einer *Daten-Seite* benutzt wird (der *Füllungsgrad* einer *Daten-Seite*), denn bei volleren Seiten muss durchschnittlich mehr Speicher verschoben werden, um Platz für den neuen Eintrag zu machen.

Der Einfluss des Füllungsgrads auf die durchschnittliche Einfügezeit, kann man in Abbildung 5.7 an dem wellenförmigen Kurvenverlauf ablesen. Bei wiederholtem Hinzufügen von Einträgen ist es in regelmäßigen Abständen notwendig, volle *Daten-Seiten* aufzuteilen und eventuell auch den Index der *Index-Seite* zu vergrößern. Wenn eine Seite aufgeteilt wird, werden nachfolgende Einfügevorgänge auf den geteilten Seiten schneller ausgeführt, da diese einen geringeren Füllungsgrad als die Ursprungs-Seite haben. Weil aufgrund der Hash-Funktion der Füllungsgrad der *Daten-Seiten* ungefähr gleichmäßig steigt, werden diese

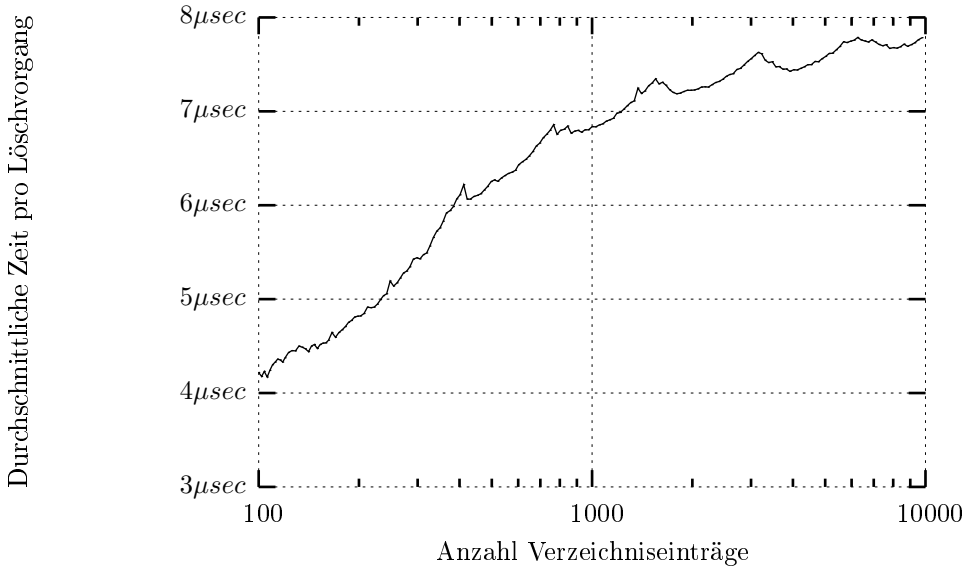


Abbildung 5.8: Dauer des Löschens eines Verzeichniseintrags in Abhängigkeit von der Verzeichnisgröße

auch ungefähr zum gleichen Zeitpunkt voll und damit aufgeteilt. Dieser Zeitpunkt der Aufteilung vieler *Daten-Seiten* lässt sich an den lokalen Maximalwerten des Kurvenverlaufs ablesen.

Löschen Die Messergebnisse für das Löschen von Einträgen sind in Abbildung 5.8 zu sehen. Das Messverfahren entspricht dem zuvor beschriebenen, das für die Messung des Einfügevorgangs verwendet wurde. In dieser Messung wird allerdings erst ein Verzeichnis erstellt und mit zufälligen Einträgen bis zu der zu messenden Größe gefüllt, dann beginnt der Messvorgang in dem die zuvor hinzugefügten Einträge wieder gelöscht werden. Dazu wird jeweils vor dem Einfüge- und Löschvorgang der Zufallsgenerator zurück gesetzt, damit pseudo-zufällig in beiden Durchläufen die gleichen Namen generiert werden.

Die Messergebnisse in Abbildung 5.8 ähneln im Verlauf denen auf Abbildung 5.7, sind jedoch zirka um den Faktor 3 geringer. Der ähnliche Verlauf ist nicht verwunderlich, da sich beide Verfahren (von eventuell durchgeführten Aufteilungs- und Verschmelzungs-Operationen abgesehen) sehr ähneln. Zunächst suchen beiden die Einfügestelle, beziehungsweise den zu löschenden Eintrag, anschließend wird der Eintrag eingefügt oder gelöscht.

Das Einfügen und Löschen eines Eintrags läuft auch vergleichbar ab: Im ersten Fall wird im Daten- und Indexteil der *Daten-Seite* durch Verschieben von Einträgen Platz geschaffen, im zweiten Fall wird der Eintrag durch Verschiebung der anderen Elemente überschrieben, es entfällt dabei das Kopieren des Eintrags. In beiden Fällen müssen Verweise im Indexteil angepasst werden; das betrifft im Durschnitt die Hälfte der Verweise.

Für das Verschieben des Speichers wird die Funktion `memmove()` verwendet, die es erlaubt Speicher korrekt zu verschieben, selbst dann, wenn Ziel und Quell-

speicher überlappen. In Abhängigkeit davon, wie sich diese Bereiche überlappen, muss `memmove()` den Bereich entweder beginnend vom Anfang oder beginnend vom Ende kopieren. Tests mit `memmove()` auf dem Testrechner haben gezeigt, dass Verschieben mit negativem Offset (das bedeutet: der Zielspeicher beginnt vor dem Quellspeicher, den er überlappt) um eine Größenordnung schneller ist als das Verschieben mit positivem Offset. Beim Verschieben mit negativem Offset muss der Puffer von vorne nach hinten durchlaufen werden, beim Verschieben mit positivem Offset von hinten nach vorne.

Die große Differenz zwischen dem Zeitbedarf der Einfüge- und Lösch-Operationen ist darauf zurückzuführen, dass bei der Lösch-Operation, die Verschiebung im Dateiteil der *Daten-Seite* mit negativem Offset durchgeführt wird, während beim Einfügen ein positiver Offset zum Einsatz kommt. Gleichzeitig müssen im Indexteil zwar Verschiebungen in der jeweils umgekehrten Richtung durchgeführt werden, diese Verschiebungen betreffen allerdings im Durchschnitt weniger Speicher, da jeder Eintrag im Datenbereich $5 + Länge(Dateiname)$ Bytes, im Indexbereich jedoch immer nur 4 Bytes belegt. Der Zeitbedarf der Gesamtoperationen wird daher deutlich von der Verschiebung im Datenteil dominiert, die bei Löschen von Einträgen, aufgrund des negativen Offsets, um ein vielfaches schneller abläuft.

Fazit Insgesamt zeigen die Operationen das erwartete Laufzeitverhalten. Insbesondere die Suche zeigt im relevanten Bereich bis 1000 Einträgen annähernd konstanten Zeitbedarf und bleibt auch im Bereich bis 10000 Einträgen, trotz vermehrter Hash-Kollisionen noch unter einer Suchzeit von $0,88\mu sec$ pro Suchanfrage ($3,92\mu sec$ *Gesamtzeit* - $3,04\mu sec$ *benötigte Zeit zum Generieren eines Dateinamens*). Erwartungsgemäß ist auch das Laufzeitverhalten der Funktionen zum Hinzufügen und Löschen von Einträgen: Der Zeitbedarf für diese Operationen steigt mit wachsender Verzeichnisgröße an, bleibt jedoch im relevanten Bereich bei unter $22\mu sec$ und zeigt auch oberhalb dieses Bereichs akzeptables Verhalten.

5.3.5.8 Zusammenfassung

In diesem Abschnitt wurde die Verzeichnisverwaltung des Metadatenmanagers vorgestellt. Dabei wurden die intern verwendeten Datenstrukturen und Algorithmen vorgestellt, sowie deren Laufzeitverhalten analysiert. Zu den wichtigsten Eigenschaften gehört dabei die geringe Dauer der Suchanfragen und eine Speicherung der Daten, welche eine sehr effiziente Beantwortung von `CLF_RQ_READDIR`-Anfragen erlaubt.

5.3.6 Flüchtige Informationen *(Volatile Inode Information)*

Unter „flüchtigen Informationen über Inodes“ versteht man Informationen, die für den Betrieb des Dateisystems zwar wichtig sind, aber nicht auf einem Datenspeicher abgelegt werden müssen, weil sie ihre Bedeutung verlieren, wenn das Dateisystem heruntergefahren wird.

Die Informationen, die derzeit verwaltet werden, sind diejenigen zu offenen Dateien und veränderten Inodes, obwohl das Modul (VII-Modul, *Volatile Inode*

Information) bereits in der Lage ist Replikate von Metadaten zu verwalten (siehe 7.2 *Zukünftige Arbeiten*).

Die Verwaltung der Informationen zu offenen Dateien ist notwendig, damit die UNIX-Semantik beim Löschens von offenen Dateien eingehalten werden kann: Dateien, die von einem Prozess geöffnet sind, können gelöscht werden, allerdings verschwindet damit zunächst nur der Verzeichniseintrag der Datei, während die Datei noch so lange bestehen bleibt, wie sie von mindestens einem Prozess verwendet wird. Damit diese Semantik auch bei Prozessen auf unterschiedlichen Knoten eines Cluster-Dateisystems funktioniert, muss der Metadatenmanager über geöffnete Dateien Buch führen.

Der Manager benachrichtigt das Modul bei den drei Anfragearten: `CLF_RQ_OPEN`, `CLF_RQ_CLOSE` und `CLF_RQ_UNLINK`. Öffnet einer der Klienten eine Datei, dann wird dies im VII-Modul gespeichert, beim Schließen der Datei wird der Eintrag wieder gelöscht. Bei einer Lösch-Anfrage veranlasst der Metadatenmanager die Verzeichnisverwaltung zur Löschung des Dateinamens aus dem Verzeichnis und benachrichtigt dann das VII-Modul. Handelt es sich um eine Datei, über die keine Informationen vorliegen (die also nicht geöffnet ist), werden auch die Metadaten und die zugehörige Inode-Nummer frei gegeben. Handelt es sich aber um eine Datei, die von mindestens einem Klienten geöffnet ist, markiert das Modul den Eintrag entsprechend und verschiebt die Freigabe der Inode bis zu dem Zeitpunkt an dem der letzte Klient die Datei schließt.

Bei seiner Verwendung im Metadaten-Surrogat verwaltet das Modul auch Informationen zu veränderten Inode-Objekten und Verzeichnissen. Diese Funktionalität wird in Abschnitt 5.4.3 behandelt.

5.3.7 Zusammenfassung

In diesem Abschnitt wurde der Metadatenmanager vorgestellt und dabei sowohl seine Zusammenarbeit mit den anderen Komponenten des Dateisystems beschrieben, wie auch seine Architektur und Details zu den einzelnen Modulen. Das Modul zur Verzeichnisverwaltung und die darin verwendeten Algorithmen und Datenstrukturen wurden besonders detailliert betrachtet.

5.4 Metadaten-Surrogat (Proxy)

Das Ziel bei der Einführung von Metadaten-Surrogaten ist die Entlastung des Metadatenmanagers durch die Surrogate, welche die Aufgaben des Managers übernehmen und unter sich aufteilen. Daher sind die Surrogate in Abbildung 5.3 zwischen den Klienten und dem Metadatenmanager angeordnet.

Die bisher in diesem Kapitel betrachteten Dateisystemkomponenten bilden zusammen bereits ein funktionierendes Dateisystem (siehe auch Abbildung 5.1). Die Komponenten sind um den Metadatenmanager angeordnet, der für die Beantwortung der Anfragen der Klienten verantwortlich ist. In Abschnitt 5.2.3 wurde beschrieben, welche Anfragen der Metadatenmanager zu bearbeiten hat. Bei Betrachtung der VFS-Schnittstelle (Abschnitt 3.4) gab es nur wenige Aufrufe, die keine Interaktion mit einer konkreten Dateisystemimplementierung benötigen (Kategorie 1, Tabelle 3.2). Von den übrigen Systemaufrufen erfordern alle Operationen eine direkte Interaktion mit dem Metadatenmanager. Betrachtet

man die impliziten Anfragen, die zur Auflösung eines Pfades benötigt werden, dann sind für einen einzelnen Systemaufruf sogar mehrere Anfragen notwendig.

Während die zentrale Verwaltung der Metadaten die Konsistenzerhaltung bei ihrer Verarbeitung vereinfacht, so schränkt sie gleichzeitig auch die Skalierbarkeit des Dateisystems ein. Mit der Einführung von Metadaten-Surrogaten soll die Skalierbarkeit verbessert werden. Gleichzeitig werden einzelne, in Bezug auf die Konsistenz kritische, Operationen, wie beispielsweise die Vergabe von Inode-Nummern, weiterhin von dem zentralen Metadatenmanager bearbeitet.

Der folgende Abschnitt gibt zunächst einen Überblick über die neue Rolle der Metadaten-Surrogaten und deren Aufgabenteilung. Dann folgt eine Diskussion des Aktualisierungsprotokolls zwischen Surrogat und Metadatenmanager. Als letztes werden zwei Klassen von Operationen vorgestellt, die eine Kooperation von bis zu vier Surrogate erfordern.

5.4.1 Übersicht

Bei Benutzung von Metadaten-Surrogaten richten die Klienten ihre Anfragen nicht mehr an den Metadatenmanager, sondern an eines der Surrogate.[3]

Verteilung der Metadaten Die Surrogate unterscheiden sich untereinander durch ihre Zuständigkeit für die Metadaten: Für die Verwaltung einer Inode ist immer genau ein Surrogat verantwortlich; handelt es sich bei der Inode um ein Verzeichnis, ist das Surrogat auch für die Verwaltung des Verzeichnisinhalts verantwortlich.

Weil es Querverweise zwischen Metadaten gibt, ist es notwendig, dass die Surrogate kommunizieren, um diese Verweise zu aktualisieren, zu löschen oder neue hinzu zu fügen. Für die Kommunikation untereinander, wie auch für die Kommunikation mit den Klienten ist notwendig, den für eine bestimmte Inode verantwortlichen Knoten mit möglichst geringem Aufwand zu finden.

Die größte Flexibilität würde eine individuelle Zuweisung von Inodes zu Surrogaten bieten, die sich zur Laufzeit ändern kann. Auf diese Weise könnten die Inodes so verteilt werden, dass eine möglichst gleichmäßige Auslastung der Surrogate erreicht wird. Allerdings müsste dazu ein „Adressbuch“ geführt werden, bei dem die Komponenten erfragen können, welches Surrogat für eine bestimmte Inode verantwortlich ist. Daher wäre bei jedem Zugriff auf Metadaten mindestens eine weitere Netzwerkkommunikation notwendig, damit der Klient oder das Surrogat zunächst erstmal das für eine bestimmte Inode verantwortliche Surrogat auffinden kann. Dieser Nachteil kann unter Umständen durch die zeitweise Zwischenspeicherung der Zuständigkeits-Informationen bei den beteiligten Komponenten kompensiert werden.

Es bleibt zweifelhaft, ob der Vorteil der dynamischen Umverteilung von Inodes die Nachteile, die durch den erhöhten Kommunikations- und Verwaltungsaufwand entstehen, aufheben kann. Aus diesem Grund wird bei *CLF* eine feste Verteilungfunktion in Abhängigkeit von der *Inode-Nummer* verwendet. Die Funktion teilt die Inodes reihum anhand der Nummer einem Surrogat zu. Mathematisch gesprochen: Jedes Surrogat verwaltet Inodes deren Nummern kongruent

[3] Eine Ausnahme bildet die Montier-Anfrage (`CLF_RQ_MOUNT`), die nach wie vor an den Metadatenmanager gerichtet wird.

bezüglich der Gesamtzahl der Surrogate sind. Der Hauptvorteil einer solch regelmäßigen Verteilung ist es, dass jede Komponente eigenständig feststellen kann, wer für die Verwaltung einer bestimmten Inode verantwortlich ist, wenn sie die Zahl der Surrogate und ihre Adressen kennt.

Rolle des Metadatenmanagers Mit der Einführung von Surrogaten verändert sich der Aufgabenbereich des Metadatenmanagers. Er ist nicht mehr für die Beantwortung der Klienten-Anfragen verantwortlich[3] sondern dient der Koordination der Surrogate. Er speichert weiterhin alle Metadateninformationen: Beim Hochfahren des Dateisystem versendet der Manager diese Informationen an die verantwortlichen Surrogaten und erhält anschließend von diesen Aktualisierungsnachrichten, falls auf den Metadaten Veränderungen vorgenommen wurden. Details zu dem verwendeten Aktualisierungsprotokoll folgen in Abschnitt 5.4.3.2.

Der Metadatenmanager übernimmt auch weiterhin die Verwaltung einer zentralen Liste der benutzten Inode-Nummern (das Modul *Allocation Manager*, Abschnitt 5.3.3) sowie das Löschen von Dateiinhalten auf den Daten-Servern, das direkt damit zusammen hängt. Die Surrogate besitzen eine Kopie der Information ob die Inodes, für die sie zuständig sind, benutzt werden oder nicht, dürfen diese Informationen aber nur unter bestimmten Bedingungen ändern.

Anhand der Inode-Nummer wird festgelegt welches Surrogat für das Inode-Objekt verantwortlich ist und da sich die Inode-Nummer während der Lebenszeit des Objektes (und damit der Datei oder des Verzeichnisses) nicht ändert, wird bei Erstellung einer neuen Datei oder eines Verzeichnis durch die Vergabe der Nummer festgelegt, welches Surrogat für dieses Objekt zuständig ist. Durch die zentrale Verwaltung der Inode-Nummern soll eine möglichst gleichmäßige Verteilung der Dateien über die Surrogate erreicht werden.

Anmerkung: Tatsächlich wäre es über die gezielte Zuweisung von Inode-Nummern möglich, die Verteilung der Metadaten zu beeinflussen. Allerdings ist unklar anhand welcher Kriterien bereits bei der Erstellung einer Datei oder eines Verzeichnisses die Zuweisung beeinflusst werden sollte. Daher werden in der aktuellen Implementierung von *CLF* die Nummern quasi zufällig verteilt.

Klienten Auf Seite der Klienten ändert sich nur wenig: Beim Versenden einer Anfrage wird der Adressat in Abhängigkeit von der betroffenen Inode-Nummer aus der Liste der Surrogate gewählt. Diese Liste erhält der Klient beim Montieren des Dateisystems vom Metadatenmanager auf die gleiche Weise, wie er auch die Adressen der Daten-Server erhält.

Bei Anfragen, die mehr als ein Inode-Objekt betreffen, sind für die Beantwortung unter Umständen mehrere Surrogate verantwortlich. In diesen Fällen erhält der Klient die Antwort auf seine Anfrage nicht von dem Surrogat an das er die Anfrage geschickt hat, sondern von einem anderen. Durch den Einsatz von *Paradis-Net* stellt dies kein Problem dar, da die Antwort anhand ihrer Kooperationsmarke (siehe Abschnitt 4.5.5) erkannt und dem richtigen Prozess zugeführt wird. Es handelt sich dabei um das in Abschnitt 4.6 vorgestellte *Weiterleitungs-Muster*.

Zusammenfassung Die Metadaten-Surrogate bearbeiten an Stelle des Metadatenmanagers alle[3] Anfragen der Klienten, wobei jedes Metadaten-Surrogat

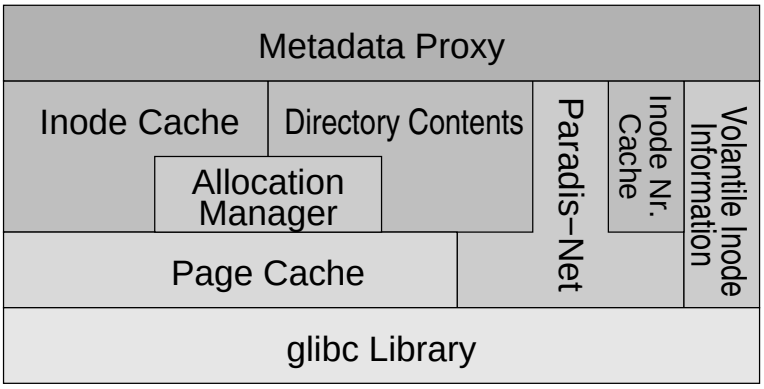


Abbildung 5.9: Modularer Aufbau des *CLF* Metadaten-Surrogats

fest für bestimmte Metadaten verantwortlich ist. Diese Zuständigkeit entscheidet sich mittels eine Hash-Funktion über die *Inode-Nummer*.

Der Metadatenmanager nimmt eine koordinierende Rolle ein. Er verwaltet wenige zentrale Datenstrukturen (z.B. die Liste der benutzen Inodes) und speichert weiterhin alle Metadaten. Bei Änderungen erhält er Aktualisierungsnachrichten von den Surrogaten.

Aus Sicht des Klienten besteht der einzige Unterschied zwischen der Benutzung von Metadaten-Surrogaten und einem zentralen Metadatenmanager in der Adressierung der Anfragen, die im ersten Fall an das jeweils zuständige Surrogate und im zweiten Fall grundsätzlich an den Metadatenmanager gesendet werden.

Die folgenden Abschnitte betrachten den modularen Aufbau der Surrogate (Abschnitt 5.4.2), sowie die in der Kommunikation mit Metadatenmanager (Abschnitt 5.4.3) und Klient (Abschnitt 5.4.4) verwendeten Kommunikationsmuster.

5.4.2 Architektur

Der interne Aufbau der Surrogate (Abbildung 5.9) ähnelt dem des Metadatenmanagers. Alle für den Manager entwickelten Module kommen hier wieder zum Einsatz. Dies ist aber auch nicht überraschend, da die Surrogate ähnliche Aufgaben haben, wie der Manager selbst. Das neue Modul *Inode Number Cache* wird in Abschnitt 5.4.3.3 behandelt.

Einige Module mussten aufgrund der speziellen Anforderungen der Surrogate im Vergleich zu der im Metadatenmanager verwendeten Funktionalität erweitert werden. Die Module *Allocation Manager*, *Directory Contents* und *Volatile Inode Information* haben zusätzliche Funktionen erhalten, um den Informationsaustausch zwischen dem Metadatenmanager und dem Surrogat zu vereinfachen. Die zusätzlichen Funktionen dienen beispielsweise dazu, die *Daten-Seiten* der Verzeichnisverwaltung komplett zu verschicken oder bestimmte Inode-Nummern zu reservieren.

Die oberste Schicht (*Metdata Proxy*) in Abbildung 5.9 unterscheidet sich allerdings vollständig von der in Abbildung 5.4 (*Metadata Server*): Während der

Metadatenmanager nach der Initialisierungsphase, in der er Konfigurationsinformationen an die Daten-Server und Surrogate versendet, in die Dienstphase übergeht, in der alle ankommende Anfragen beantwortet werden, benötigt ein Surrogat zusätzliche Informationen über das Dateisystem und sich selber, um Anfragen beantworten zu können:

1. Anzahl n_{Sur} und Adresse $adr(i) \quad i \in \{0, \ldots, n_{Sur}\}$ aller Surrogate
2. Eigene Position (i_{Sur}) in dieser Liste

Diese Informationen erhält ein Surrogat beim Start von Metadatenmanager und kann damit bestimmen, für welche Inodes es verantwortlich ist. Der Index i des für die Inode mit der Nummer $InodeNr$ verantwortlichen Surrogats S und dessen Adresse lässt sich wie folgt bestimmen:

$$\begin{aligned} i &= InodeNr \mod n_{Sur} \\ S &= adr(i) \end{aligned}$$

Mit den Informationen ist es auch möglich die bestehenden Module zur Verwaltung der Inodes weiter zu verwenden, obwohl nur ein Teil der Inodes auf jedem Knoten gespeichert werden sollen. Wie in Abschnitt 5.3.4 beschrieben, werden in den Modulen die Inodes durch ihre Nummer adressiert und hintereinander in einer Datei gespeichert. Auf den Surrogaten würde daher nur $(1/n_{Sur})$-tel der Einträge benutzt werden, die übrigens würden leer bleiben. Um das zu vermeiden, werden die Inodes bei Benutzung der Module *Inode Cache* und *Allocation Manager* nicht über ihre Inode-Nummer sondern über den Index $\lfloor \frac{InodeNummer}{n_{Sur}} \rfloor$ adressiert.

Der Rest des Abschnitts 5.4 behandelt die oberste Schicht der Architektur des Surrogats und wie diese die in diesem Kapitel beschriebenen Module einsetzt. Der folgende Abschnitt betrachtet zunächst die Kommunikation mit dem Metadatenmanager, in Abschnitt 5.4.4 steht die Kommunikation mit den Klienten im Vordergrund.

5.4.3 Kommunikation mit dem Metadatenmanager

Die Kommunikation zwischen Surrogat und Metadatenmanager ist dominiert von den Aktualisierungsnachrichten der Surrogate. Darüber hinaus lassen sich aber noch zwei weitere von insgesamt drei Teilbereichen identifizieren:

1. **Initialisierung** der Daten auf den Surrogaten, (5.4.3.1)
2. **Aktualisierung** der Daten des Metadatenmanagers und (5.4.3.2)
3. **Inode-Nummern Verwaltung** des Metadatenmanagers. (5.4.3.3)

5.4.3.1 Initialisierung

Die Surrogate befinden sich, wie auch die Daten-Server (Abschnitt A) nach ihrem Start in einem *Konfigurationsmodus*, in dem sie zunächst keine Anfragen beantworten, sondern auf die Initialisierung durch den Metadatenmanager warten. Die Initialisierung ist im Vergleich zu den Daten-Servern aufwändiger, weil nicht nur die Konfiguration des Dateisystems übermittelt wird, sondern auch alle Metadaten, für die das Surrogat verantwortlich ist.

Die Initialisierung erfolgt in zwei Phasen: In der ersten sendet der Metadatenmanager zunächst einmal die Grundinformationen (Nachrichtentyp `CLF_RQ_PROXY_INIT_START`) über das Dateisystem an die Surrogate:

Index des Adressaten und Zahl aller Surrogate. Dies sind, zusammen mit den Adressen aller Surrogate, die wichtigsten Daten zur Berechnung der Zuständigkeit für ein Inode-Objekt.

Lokaler Pfad. Der Pfad unter dem das Surrogat seine Daten speichert. Eventuell vorhandene Daten in diesem Pfad werden beim Start gelöscht, da das Surrogat alle Daten vom Metadatenmanager erhält.

Die *Paradis-Net*-Adresse aller Surrogate. Diese Liste wird benötigt, da die Surrogate bei manchen Operationen zusammen arbeiten müssen.

Nachdem der Metadatenmanager von allen Surrogaten eine Quittung erhalten hat, geht er in die zweite Phase der Initialisierung über. Dabei durchläuft er alle benutzten Inode-Objekte und sendet diese einzeln an das jeweils verantwortliche Surrogat.

Handelt es sich bei einem Inode-Objekt um ein Verzeichnis, dann muss auch der Inhalt des Verzeichnisses an das Surrogat gesendet werden. Zu diesem Zweck werden die Informationen in Form der *Daten-Seiten* (Abschnitt 5.3.5.2) an die Nachricht angehangen. Auf diese Weise können die Seiten bei den Surrogaten direkt in das Verzeichnisverwaltungs-Modul hineinkopiert werden, ohne dass eine weitere Bearbeitung notwendig ist. Einzig die *Index-Seiten* des Verzeichnisses müssen bearbeitet werden, da diese absolute Verweise auf die *Daten-Seiten* des Managers enthalten.

Auf jedem Surrogat wird nur ein der Teil der gesamten Verzeichnisinformationen gespeichert. Im Gegensatz zu den Inode-Objekten, die aufgrund ihrer Nummer, die als Schlüssel für den Zugriff verwendet wird, an einem festgelegten Platz gespeichert werden müssen, ist es außerhalb des Verzeichnisverwaltung-Moduls irrelevant, welche konkrete Seitennummer eine bestimmte *Index-Seite* belegt, solange diese Nummer in dem Inode-Objekt korrekt gespeichert wird. Daher erfolgt die Verwaltung der Verzeichnissinformationen auf jedem Surrogat lokal, das bedeutet, dass die Verzeichnis- und *Index-Seiten*, die der Manager dem Surrogat bei der Initialisierung sendet, dort in anderen Seiten gespeichert werden. Die absoluten Verweise auf den *Index-Seiten* werden vom Manager zunächst in relative Verweise umgewandelt, die der Reihenfolge der *Daten-Seiten* in der Nachricht entspricht, und auf der Empfängerseite werde diese wieder in absolute Verweise umgewandelt, sobald den *Daten-Seiten* lokale Seitennummern zugewiesen wurden.

Sind alle Metadaten an die Surrogate verteilt worden, sendet der Metadatenmanager diesen eine Nachricht vom Typ `CLF_RQ_PROXY_INIT_END`, die sie in den *Dienstmodus* versetzt.

5.4.3.2 Aktualisierung

Nach der Initialisierung durch den Metadatenmanager besteht zwischen den Metadaten auf den Surrogaten und auf dem Manager kein Unterschied. Dies ändert sich, sobald die Metadaten durch Zugriffe der Klienten manipuliert werden. Um die Daten auf dem Manager zu aktualisieren, senden die Klienten in regelmäßigen Abständen Aktualisierungsnachrichten an diesen.

Das *Volatile Inode Information*-Modul (VII-Modul) führt Buch darüber, welche Inodes und welche Verzeichnisinhalte auf einem Surrogat geändert wurden. Die Veränderungen werden von den Bearbeitungsfunktionen der Anfragen an das VII-Modul gemeldet. Ein zusätzlicher Ausführungsfaden überprüft durch eine Anfrage an das Modul (standardmäßig einmal pro Sekunde), ob sich die Metadaten geändert haben und eine Aktualisierung des Managers notwendig ist.

Zu Aktualisierung des Metadatenmanagers verwenden die Surrogate zwei Nachrichtentypen: Zum einen werden geänderte Verzeichnisse einzeln mit dem in Abschnitt 5.4.3.1 beschriebenen Verfahren an den Manager verschickt. Zum anderen werden, um bei den geänderten Inode-Daten eine große Menge von Nachrichten zu vermeiden, die geänderten Inodes in Blöcken von 314 Inodes ($\approx$ 32 KB pro Nachricht) an den Manager geschickt. Bei Empfang der Nachrichten überscheibt der Metadatenmanager die alten Daten mit den neuen.

Der Metadatenmanager erhält mit jeder Aktualisierungsnachricht nur eine Moment-Aufnahme des jeweiligen Surrogats, der Gesamtzustand muss jedoch in sich nicht konsistent sein, da die Surrogate zu unterschiedlichen Zeitpunkten ihre Aktualisierungen schicken. Ein konsistenter Stand wird erst beim Herunterfahren des Dateisystems erreicht, oder wenn während eines gesamten Aktualisierungszyklus keine weiteren Änderungen an den Metadaten durchgeführt wurden.

Für die Entwicklung von *CLF* wurde nur eine zeitlich regelmäßige Aktualisierung implementiert. Diese Form führt unter Umständen zu unnötigen Aktualisierungen, also mehrmaligen aufeinander folgenden Aktualisierungen der gleichen Inodes, die zu größerem Netzwerkverkehr führen. Das Finden eines günstigen Zeitpunktes für die Aktualisierung ist ein Optimierungsproblem: Auf der einen Seite soll möglichst lange mit der Aktualisierung gewartet werden, um mehrmalige Aktualisierungen und damit unnötigen Netzwerkverkehr zu vermeiden. Auf der anderen Seite muss das VII-Modul jedes Surrogaten eine Liste der geänderten Inodes und Verzeichnisse führen, die einen gewissen Speicherplatz und Verwaltungsaufwand erfordern, der mit der Größe der Liste steigt. Es erscheint also sinnvoll, den Zeitpunkt der Aktualisierung einerseits von der seit der letzten Aktualisierung vergangen Zeit und andererseits von der Anzahl der geänderten Metadaten abhängig zu machen. Im Rahmen dieser Arbeit wurde dieses Problem nicht weiter untersucht.

5.4.3.3 Inode-Nummern Verwaltung

Auch wenn die *Verantwortung für Inodes* auf die Surrogate verteilt wurde, verbleibt die *Verwaltung der Inode-Nummern* beim Metadatenmanager. Dazu müssen zwei Arten von Anfragen beantwortet werden:

1. **Die Anforderung einer freien Inode-Nummer**, wenn eine neue Datei oder ein neues Verzeichnis angelegt wird.
2. **Die Rückgabe einer nicht mehr verwendeten Inode-Nummer**, wenn eine Datei oder ein Verzeichnis gelöscht wird.

Wir betrachten den Einsatz dieser Anfragen anhand des Lebenszyklus einer Datei D: Die Datei D soll in einem Verzeichnis V angelegt werden. Diese Anfrage erhält das Surrogat S, das für die Verwaltung des Verzeichnisses verantwortlich ist. Dieses Surrogat fordert für die neue Datei eine freie Inode-Nummer

beim Metadatenmanager an. Nach Erhalt der Nummer $InNr$, wird die Datei initialisiert, in das Verzeichnis V eingetragen und kann dann von allen Klienten benutzt werden. (Wir nehmen an dieser Stelle der Einfachheit halber an, dass die Metadaten der Datei auch von S verwaltet werden.)

Das Löschen der Datei erfolgt durch den Klienten über eine Anfrage an S, dem Verantwortlichen für D. Daraufhin wird zunächst der Verzeichniseintrag von D aus dem Verzeichnis V entfernt, anschließend wird das zu löschende Inode-Objekt dem VII-Modul gemeldet, das (wie in Abschnitt 5.3.6 für den Metadatenmanager beschrieben) eine Liste der auf Klienten geöffneten Dateien führt. Die Anforderung der Löschung der Metadaten von D wird so lange verzögert, wie die Datei D auf einem Klienten geöffnet ist. Ist dies nicht (mehr) der Fall, kann die Datei endgültig gelöscht werden, indem die frei gewordene Inode-Nummer an den Manager geschickt und auf dem Surrogat als ungültig markiert wird. Der Metadatenmanager gibt schließlich die empfangene Inode-Nummer $InNr$ in der zentralen Datenstruktur frei, nachdem der Dateiinhalt auf den Daten-Servern gelöscht wurde. Ab diesem Zeitpunkt ist $InNr$ wieder verfügbar und wird bei Bedarf an ein Surrogat vergeben, das eine Nummer anfordert.

Um dem Kommunikationsaufwand und gleichzeitig die Verzögerungen durch die zusätzliche Anfrage beim Anlegen von Dateien und Verzeichnissen zu vermeiden, wird das Modul *Inode Number Cache* eingesetzt. Es speichert eine größere Menge Inode-Nummern, die es mittels einer speziellen Anfrage (`CLF_RQ_PROXY_ALLOC_INODES`) vom Metadatenmanager erhalten hat. Benötigt das Surrogat eine neue Inode-Nummer, kann das Modul die Anfrage lokal aus den zuvor beantragten und vorgehaltenen Inode-Nummern beantworten. Sobald die Zahl der vorgehaltenen Nummern einen gewissen Wert unterschreitet, wird im Hintergrund ein neuer Inode-Nummern-Block vom Metadatenmanager angefordert. Durch die Zwischenspeicherung muss einerseits weniger kommuniziert werden (eine Anfrage für eine große Menge Nummern) und andererseits wird die Verzögerung, die bei individuellen Anfragen an den Metadatenmanager entstehen würde, vollständig verborgen.

Ähnlich gehen die Surrogate auch bei der Rückgabe der frei gewordenen Nummern vor: Diese werden nicht sofort gemeldet, sondern zunächst gesammelt und zum Aktualisierungszeitpunkt an den Metadatenmanager gesendet.

Hierbei ist zu beachten, dass für die blockweise Herausgabe der Inode-Nummern der Metadatenmanager zusätzliche Informationen speichern muss. Die angeforderten Inode-Nummern werden nicht sofort verwendet, sind also nicht in Gebrauch. Um den Zustand einer Inode, deren Nummer bereits rausgegeben wurde, die aber noch keine Datei und kein Verzeichnis repräsentiert, innerhalb der Managers zu markieren, werden die Inode-Daten in einen ungültigen Zustand versetzt, anhand dessen der Manager erkennen kann, dass die Inode-Nummer zwar schon an ein Surrogat vergeben wurde, aber dieser sie noch nicht verwendet. Erst wenn die Inode-Nummer verwendet wird wird diese im Zuge der Aktualisierung in einen gültigen Zustand versetzt.

5.4.3.4 Zusammenfassung Aktualisierung

Das VII-Modul verwaltet drei Arten Informationen über Inodes, die auf dem zentralen Metadatenmanager aktualisiert werden müssen: geänderte Verzeichnisse, geänderte Inodes (siehe 5.4.3.2) und gelöschte Inodes (siehe 5.4.3.3).

Wie die zu aktualisierenden Daten an den Manager gesendet werden, ist in

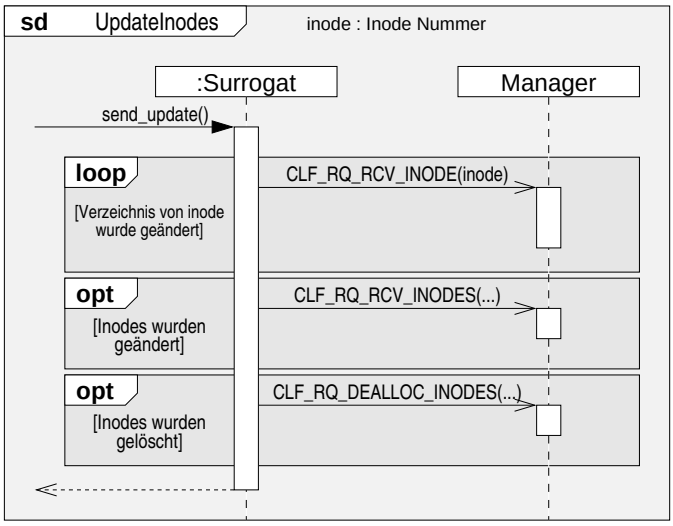


Abbildung 5.10: Aktualisierung geänderter und gelöschter Inodes auf dem Metadatenmanager

Abbildung 5.10 zu sehen. Das Sequenzdiagramm stellt den Verlauf der Kommunikation vereinfacht dar und beschreibt einen vollständigen Aktualisierungsprozess (`send_update()`). Zunächst wird geprüft, ob Verzeichnisse geändert wurden. Ist das der Fall, werden die Inode und Verzeichnisinformationen einzeln als `CLF_RQ_RCV_INODE`-Anfragen an den Manager gesendet. Wurden Inodes verändert, die im ersten Schritt noch nicht aktualisiert wurden, werden diese im zweiten Schritt zusammen als `CLF_RQ_RCV_INODES`-Anfrage an den Manager geschickt.

Im letzten Schritt werden schließlich alle gelöschten Inode-Nummern als `CLF_RQ_DEALLOC_INODES`-Nachricht an den Manager gemeldet. Anschließend ist die Aktualisierung beendet. Die `send_update()`-Funktion wird auf jedem Surrogat in regelmäßigen Abständen aufgerufen und blockiert während ihres Ablaufs die Beantwortung von Anfragen, die Metadaten verändern. Wie aus dem Sequenzdiagramm zu ersehen ist, warten die Surrogate allerdings nicht auf eine Bestätigung des Managers, denn die Aktualisierungsnachrichten sind asynchron implementiert. Für das Surrogat ist es nicht wichtig, wann der Manager seine Daten aktualisiert und weil *Paradis-Net* garantiert, dass die Daten dort eintreffen, kann das Surrogat auf den Erhalt einer Quittung verzichten.

Bezüglich der Aktualisierung von Verzeichnisinhalten ist das Protokoll sehr grob, da ein geändertes Verzeichnis immer komplett an den Metadatenmanager gesendet wird. An dieser Stelle gibt es weitere Optimierungsmöglichkeiten, die allerdings im Rahmen dieser Arbeit nicht untersucht wurden.

5.4.4 Kommunikationsmuster bei Klientenanfragen

Eine gleichmäßige Verteilung der Metadaten wird bei *CLF* dadurch erreicht, dass die Inodes anhand ihrer Nummer und unabhängig von ihrem Inhalt, reihum auf die Surrogate verteilt werden (siehe 5.4.1). Nachteil dieses Verteilungssche-

Beantwortung durch...	*Anfragetypen*	*siehe*
Metadatenmanager	`CLF_RQ_MOUNT, CLF_RQ_UMOUNT, CLF_RQ_STATFS`	-
1 Surrogat	`CLF_RQ_LOOKUP, CLF_RQ_READ_INODE, CLF_RQ_NOTIFY_CHANGE, CLF_RQ_OPEN, CLF_RQ_CLOSE, CLF_RQ_READDIR`	5.4.4.1 (Seite 120)
≤ 2 Surrogate	`CLF_RQ_CREATE, CLF_RQ_UNLINK, CLF_RQ_MKDIR, CLF_RQ_RMDIR`	5.4.4.2 (Seite 122)
≤ 4 Surrogate	`CLF_RQ_RENAME`	5.4.4.3 (Seite 124)

Tabelle 5.3: Die an der Beantwortung einer Anfrage beteiligten Komponenten

mas ist es, dass es mit hoher Wahrscheinlichkeit Unterverzeichnisbeziehungen zwischen Inodes gibt, die auf verschiedenen Surrogaten verwaltet werden. Dies ist Folge der quasi zufälligen Verteilung der Inodes anhand ihrer Nummer und erfordert bei bestimmten Anfragen die Zusammenarbeit mehrerer Surrogate. Die Anfragen, die eine Zusammenarbeit mehrerer Surrogate erfordern, werden *potentiell-kooperativ* genannt.

Zunächst werden drei Kategorien von Anfragen unterschieden, die sich in der Zahl der beteiligten Inode-Objekte unterscheiden. Die Zahl der betroffenen Inode-Objekte entspricht direkt der potentiellen Zahl an Surrogaten, die an der Erfüllung der Operation beteiligt sein können. Die Kommunikationsmuster, die dadurch entstehen, werden in den folgenden Abschnitten genauer betrachtet.

Tabelle 5.3 teilt die *CLF* Anfragen nach der Zahl der beteiligten Surrogate ein und enthält Verweise auf die Abschnitte, die sich mit der jeweiligen Kategorie befassen. Die Tabelle enthält aus Gründen der Vollständigkeit auch die Kategorie der Anfragen, die der Metadatenmanager beantwortetet. Diese Anfragen betreffen keine einzelnen Inodes; sie dienen der Initialisierung (`CLF_RQ_MOUNT`) bzw. dem Abmelden (`CLF_RQ_UMOUNT`) des Klienten oder fragen allgemeine Informationen über das Dateisystem ab (`CLF_RQ_STATFS`) und werden daher im Folgenden nicht weiter behandelt.

5.4.4.1 Einfache Anfragen: 1 Surrogat

Die *einfachen Anfragen* können von einem einzelnen Surrogat beantwortet werden, da sie nur ein Inode-Objekt betreffen. Die Anfragen unterscheiden sich kaum von einer Anfrage an den Metadatenmanager. Die Anfragen, die in diese Kategorie fallen (siehe Tabelle 5.3) machen gleichzeitig auch bis zu 95% der gesamten Anfragen aus [27]. Vor allem die CLF_RQ_LOOKUP- und CLF_RQ_READ_INODE-Anfragen können daher besonders effizient bearbeitet werden.

Als Beispiel für eine Anfrage dieser Kategorie betrachten wir eine *Notify Change*-Anfrage, die der Klient bei einer Änderung der Inode-Daten versendet, wie beispielsweise aufgrund eines chmod-Befehls. Abbildung 5.11 zeigt ein Sequenzdiagramm [39], das den Verlauf einer solchen Anfrage darstellt. Dieses und die folgenden Sequenzdiagramme dienen dazu den Ablauf einer typischen Anfrage zu veranschaulichen. Daher wurden sie so vereinfacht, dass sie nur die für das Verständnis notwendigen Funktionsaufrufe und Daten enthalten und Fehlerbehandlung weitgehend ausgeklammert wird. Die Interaktionspartner (in Abbildung 5.11: Surrogat, :Klient und Manager) stellen jeweils eine Komponente

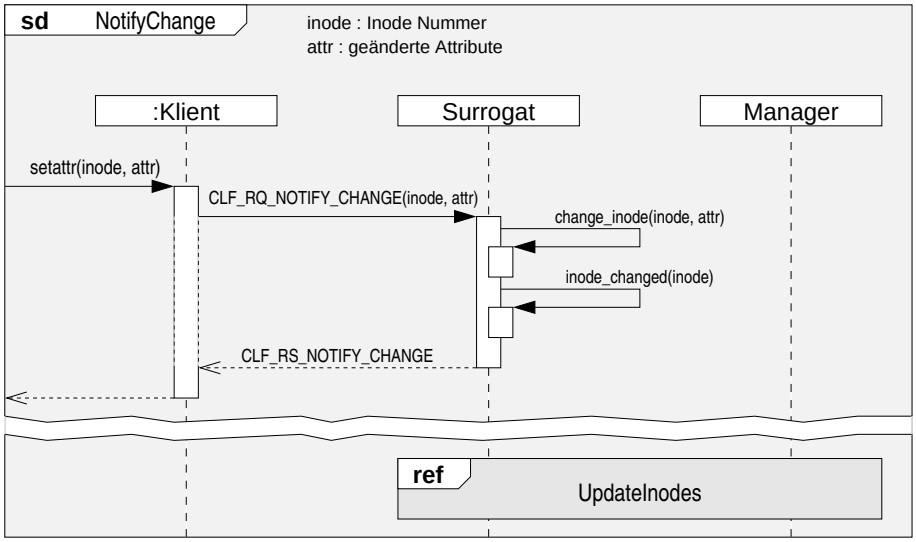


Abbildung 5.11: Das Sequenzdiagramm einer *Notify Change*-Anfrage

des *CLF*-Dateisystems dar. Die Komponenten können miteinander nur über das Netzwerk kommunizieren, die Nachrichten werden in Anlehnung an die Nachrichtentypen mit Großbuchstaben bezeichnet, die enthaltenen Daten folgen als Parameter.

Für die Kommunikation miteinander verwenden die Komponenten *Paradis-Net* und dabei kommen Kooperationen und die Fähigkeit von *Paradis-Net*, Nachrichten an einen Partner weiterzuschicken, zum Einsatz. Diese Implementationsdetails werden in den Sequenzdiagrammen nicht thematisiert und nur im Text erwähnt.

In Abbildung 5.11 wird die Anfrage CLF_RQ_NOTIFY_CHANGE aufgrund einer expliziten Metadatenänderungen, die den Aufruf der setattr() Einschubmethode zur Folge hat, abgesetzt. Die Anfrage wird an das für die Inode verantwortliche Surrogat gesendet, das daraufhin die Metadaten ändert (change_inode()). Weiterhin wird die Veränderung an der Inode registriert, im Sequenzdiagramm ist dies durch einen Aufruf der inode_changed()-Funktion dargestellt. Diese Information wird in dem *Volatile Inode Information* Modul gespeichert, wie in Abschnitt 5.4.3.2 beschrieben wurde. Der Aktualisierungsprozess (UpdateInodes) wird in dem Sequenzdiagramm auf Abbildung 5.10 dargestellt.

Der Aktualisierungsprozess muss nicht direkt auf die Anfrage des Klienten folgen. In dem Sequenzdiagramm wurde die Aktualisierung daher durch einen „Riss“ von der Anfrage abgetrennt um auf diese Weise den unbekannten zeitlichen Abstand darzustellen. Diese Aktualisierung stellt, zusammen mit dem Aufruf von inode_changed() den Unterschied zur traditionellen Bearbeitung der Anfrage auf dem Metadatenmanager dar.

Fehler Im Fehlerfall wird ein Fehler-Code in der CLF_RS_NOTIFY_CHANGE Nachricht an den Klienten geschickt. Ein Fehler kann nur in der change_inode()-Methode auftreten, beispielsweise, wenn ein anderer Knoten die Inode zwi-

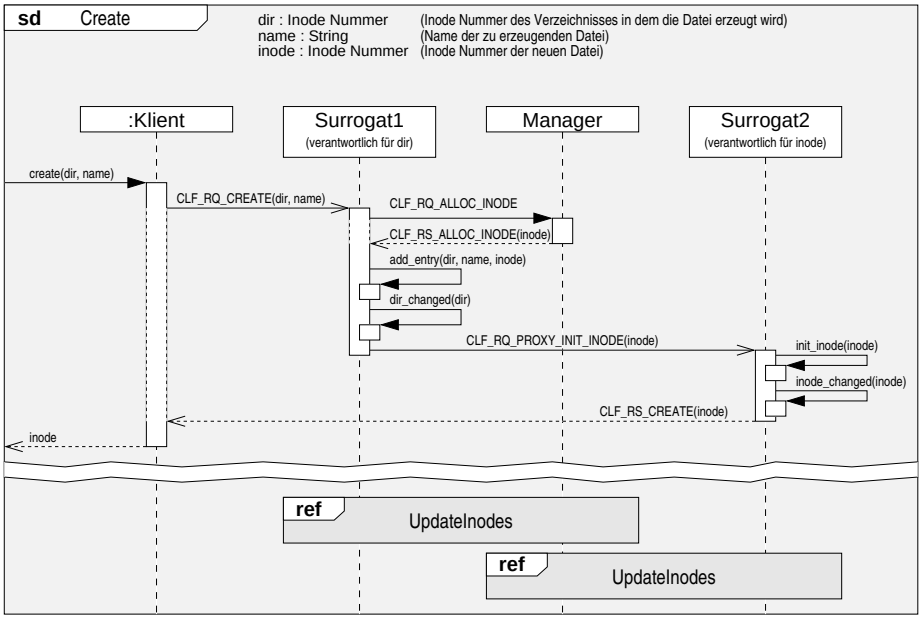


Abbildung 5.12: Das Sequenzdiagramm einer *Create*-Anfrage

schenzeitlich gelöscht hat.

5.4.4.2 Erzeugen und Löschen: bis zu 2 Surrogate

Die in diesem und im folgenden Abschnitt beschriebenen *potentiell-kooperativen* Anfragen benötigen im Allgemeinen mehr als ein Surrogat für ihre Bearbeitung. Die tatsächliche Anzahl der Surrogate ist neben dem Anfragetyp jedoch auch abhängig von den Anfrageparametern, genauer gesagt: von den betroffenen Inodes. Die in Tabelle 5.3 und in diesen Abschnitten genannte Zahl an Surrogaten stellt eine obere Grenze dar, die dann erreicht wird, wenn alle betroffenen Inodes von unterschiedlichen Surrogaten verwaltet werden.

Für den Fall, dass zwei an einer Anfrage beteiligte Inodes von dem gleichen Surrogat verwaltet werden, handelt es sich bei den Nachrichten zwischen den beiden vermeidlich unterschiedlichen Surrogaten lediglich um lokale Funktionsaufrufe. Die *potentiell-kooperative* Anfrage wird in diesem Fall wie eine *einfache Anfrage* von einem Surrogat bearbeitet. (Daher auch *potentiell-kooperativ*.) In den Sequenzdiagrammen 5.12 und 5.13 wird jedoch vereinfachend vom ungünstigsten Fall ausgegangen, bei dem alle Inodes von unterschiedlichen Surrogaten verwaltet werden. Alle potentiellen Netzwerknachrichten werden daher als Nachrichten zwischen unterschiedlichen Komponenten dargestellt.

Die Anfragen zum Erzeugen von Dateien betreffen immer zwei Inodes: Zum einen die Inode, die das Verzeichnis repräsentiert, in dem der Eintrag erstellt wird (dir) und zum anderen die Inode der neu erstellten Datei (inode). Das Sequenzdiagramm in Abbildung 5.12 zeigt den Ablauf einer create()-Anfrage zum Erzeugen einer Datei. Diese Anfrage gleicht der zum Anlegen eines Verzeichnisses, der einzige Unterschied besteht in den verwendeten Nachrichtentypen.

Die wichtigsten Parameter der create()-Anfrage sind das Verzeichnis, in dem die Datei angelegt werden soll (dir) und der Name der neu zu erstellenden Datei (name). Darüber hinaus gehören zu der Anfrage auch noch die Zugriffsrechte der neuen Datei, sowie deren Besitzer- und Gruppenzugehörigkeit. Aus Gründen der Übersichtlichkeit sind die zuletzt genannten Parameter nicht im Sequenzdiagramm abgebildet.

Nachdem die Einschubmethode create() auf dem Klienten aufgerufen wurde, stellt dieser eine CLF_RQ_CREATE-Anfrage mit den im letzten Absatz genannten Parametern an Surrogat1, das für die Verwaltung des Verzeichnisses verantwortlich ist. Das Surrogat stellt zunächst sicher, dass kein Verzeichniseintrag unter dem neuen Namen existiert (ansonsten meldet es einen Fehler an den Klienten und bricht die Bearbeitung der Anfrage ab). Das Surrogat fordert daraufhin eine freie Inode-Nummer inode beim Metadatenmanager an. Dies erfolgt im Prinzip so, wie es im Sequenzdiagramm dargestellt ist, allerdings schickt das Surrogat nicht einzelne Anfragen, sondern hält zu jedem Zeitpunkt eine gewisse Menge an freien Inode-Nummern bereit, wie in Abschnitt 5.4.3.3 beschrieben wurde.

Surrogat1 fügt nun die neue Datei name mit Inode inode zu dem Verzeichnis hinzu. Aufgrund der neuen Inode-Nummer stellt Surrogat1 fest, dass die Metadaten der Inode von Surrogat2 verwaltet werden. Surrogat1 schickt eine CLF_RQ_INIT_INODE-Anfrage an Surrogat2, das daraufhin die Metadaten anhand der mitgesendeten Parameter initialisiert. Die Nachricht wird per `pdn_forward()` (siehe Abschnitt 4.5.4) verschickt, wodurch das Surrogat mit der Anfrage auch die Kooperationsmarke erhält. Nachdem es die Metadaten initialisiert hat richtet es seine Antwort nicht an Surrogat1, von dem es die Anfrage erhalten hat, sondern schickt dem Klienten direkt die Erfolgsbestätigung der *Create*-Anfrage zu.

Beide Surrogate nehmen Änderungen an ihren Metadaten vor: Auf Surrogat1 wird das Verzeichnis dir durch das Hinzufügen des neuen Eintrags verändert, auf Surrogat2 werden die Metadaten der Inode inode initialisiert. Diese Änderungen werden durch Aufruf der Funktionen dir_changed() und inode_changed() an das jeweilige VII-Modul gemeldet und im späteren Verlauf an den Metadatenmanager gemeldet. In Abbildung 5.12 ist die Aktualisierung durch die Verweise auf das UpdateInodes-Sequenzdiagramm angedeutet. Die Reihenfolge der Aktualisierungsoperationen von beiden Surrogaten ist allerdings nicht fest und nur indirekt an die Anfrage gekoppelt. Der unbestimmte zeitlich Abstand zwischen Anfrage und Aktualisierung wird im Sequenzdiagramm durch einen Riss dargestellt.

Fehler Es können Fehler beim Anfordern der Inode-Nummer und beim Hinzufügen des Verzeichniseintrags auftreten. Diese Fehler werden direkt von Surrogat1 an den Klienten gemeldet. Tritt beim Hinzufügen des Verzeichniseintrags ein Fehler auf, wird zunächst die zuvor angeforderte Inode-Nummer zurück gegeben.

Die Initialisierung der Inode auf Surrogat2 kann nicht fehlschlagen, daher muss nach der Weitergabe der Anfrage an Surrogat2 auch keine weitere Fehlerbehandlung vorgesehen werden.

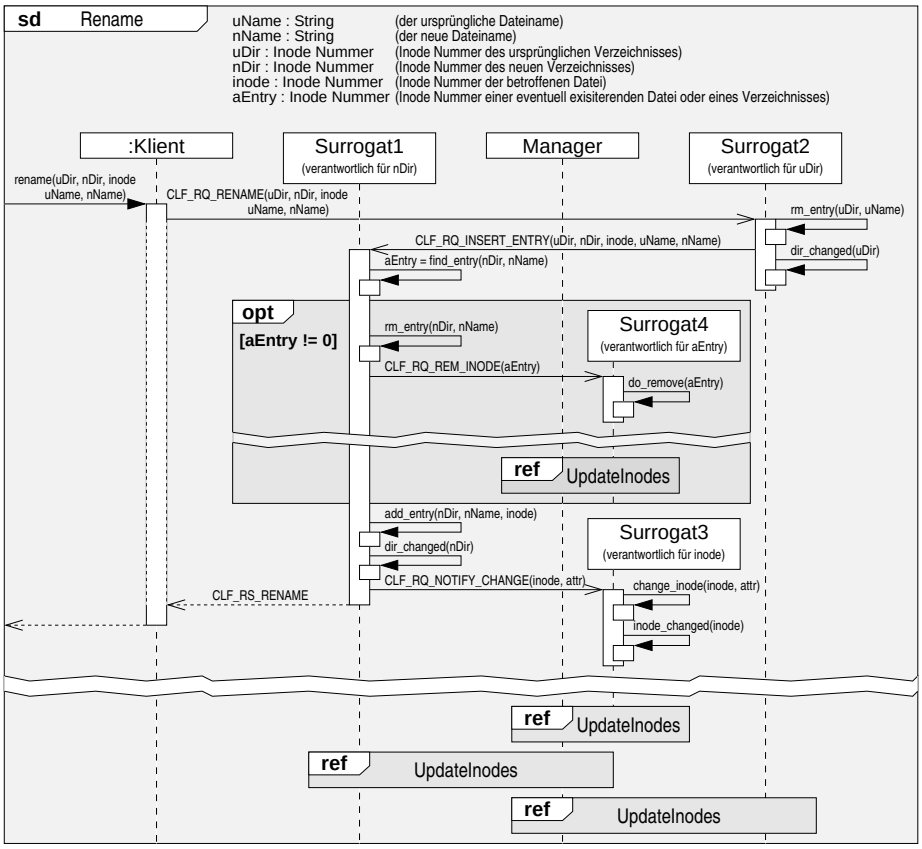


Abbildung 5.13: Für eine *Rename*-Anfrage müssen bis zu 4 Surrogate zusammen arbeiten.

5.4.4.3 Umbenennen: bis zu 4 Surrogate

Die *Rename*-Operation des VFS[4] ist eine komplexe Operation: Sie dient, wie ihr Name andeutet, zum Umbenennen von Dateien oder Verzeichnissen. Darüber hinaus ist es aber gleichzeitig möglich, die Datei oder das Verzeichnis in ein anderes Verzeichnis zu verschieben. Parameter der Operation sind daher neben dem ursprünglichen und dem neuen Namen des betroffenen Eintrags (uName, nName) auch das ursprüngliche und das neue Verzeichnis, in dem die Datei oder das Verzeichnis abgelegt ist/wird (uDir, nDir).

Für die folgende Betrachtung der *potentiell-kooperativen* Operation werden wir annehmen, dass eine gleichzeitige Umbenennung und Verschiebung einer Datei mit der Inode-Nummer inode durchgeführt wird (uName $\neq$ nName und uDir $\neq$ nDir). Weiterhin wird angenommen, dass die Metadaten des betroffenen Eintrags und die Metadaten der beiden Verzeichnisse auf unterschiedlichen Surrogaten verwaltet werden. Unter diesen Annahmen sind genau drei Surrogate an der Operation beteiligt. Ein weiteres Surrogat kommt noch hinzu, wenn wir

[4] Auf der Kommandozeile von UNIX ist die Operation unter dem Namen `mv` (Abkürzung für *move*) bekannt.

annehmen, dass sich im Zielverzeichnis nDir bereits ein Eintrag (aEntry) unter dem Namen nName befindet, der von einem vierten Surrogat verwaltet wird. Ist dies der Fall, wird der bestehende Eintrag gelöscht und durch den neuen ersetzt.

Abbildung 5.13 zeigt das Sequenzdiagramm der Umbenennungsoperation. Die Anfrage wird von einem Klienten gestellt und sie beginnt mit einem Aufruf der rename() Einschubmethode mit den bereits vorgestellten Parametern. Die Methode sendet diese Parameter in einer Nachricht vom Typ CLF_RQ_RENAME an Surrogat2, das für die Verwaltung des Quellverzeichnisses uDir verantwortlich ist. Das Surrogat entfernt zunächst einmal den Eintrag uName aus diesem Verzeichnis (rm_entry(uDir, uName)). An dieser Stelle erfolgt eine wichtige Fehlerbehandlung, die jedoch nicht im Sequenzdiagramm dargestellt ist: Für den Fall, dass der Eintrag der Datei nicht (mehr) existiert, kann die Operation als Ganzes nicht durchgeführt werden, weil eine Verschiebung und/-oder Umbenennung ohne die ursprüngliche Datei nicht durchgeführt werden. In diesem Fall wird die Operation an dieser Stelle abgebrochen und eine Fehlermeldung an den Klienten geschickt.

Ist das Entfernen des Eintrags erfolgreich verlaufen, sendet das Surrogat eine CLF_RQ_INSERT_ENTRY-Anfrage an Surrogat1, das für die Verwaltung des Zielverzeichnisses nDir zuständig ist. Mit dieser Anfrage überträgt es gleichzeitig auch die Verantwortung für die Beantwortung der Anfrage des Klienten an Surrogat1 (per pdn_forward()). Nach Versenden der Nachricht meldet Surrogat2 die Änderung des Verzeichnisinhalts von uDir an sein VII-Modul und hat damit seine Aufgabe erfüllt.

Surrogat1 prüft zunächst durch Aufruf der find_entry()-Funktion, ob bereits ein Eintrag unter dem gewünschten Namen nName vorhanden ist. Ist dies der Fall, wird der Bereich ausgeführt, der im Sequenzdiagramm als **opt**ional gekennzeichnet ist.

Innerhalb des optionalen Bereichs wird die bereits unter dem Namen bestehende Datei entfernt, indem zunächst der Verzeichniseintrag durch Aufruf der rm_entry()-Methode entfernt wird. Anschließend muss auch die zu der Datei gehörenden Inode gelöscht werden. Diese Inode (aEntry) wird von einem weiteren Surrogat (Surrogat4) verwaltet, dem Surrogat1 zu diesem Zweck eine Nachricht vom Typ CLF_RQ_REM_INODE schickt. Das Surrogat4 löscht durch Aufruf der do_remove()-Methode daraufhin die Inode (sobald sie von keinem der Klienten mehr geöffnet gehalten wird). Während des nächsten Aktualisierungszyklus wird das Löschen der Inode an den Metadatenmanager gemeldet. Weil der genaue Zeitpunkt der Aktualisierung nicht fest steht, wurde dieser Teil des **opt**-Blockes durch einen Riss vom Rest des Diagramms abgetrennt.

Anschließend wird von Surrogat1 der Verzeichniseintrag der umbenannten / verschobenen Datei hinzugefügt (add_entry()) und das Verzeichnis als „geändert“ beim VII-Modul gemeldet (dir_changed()).

Weil sich mit der Verschiebung des Eintrags von inode auch der Änderungszeitpunkt (ctime) der Inode ändert, muss Surrogat1 dem für die Verwaltung von inode verantwortlichen Surrogat3 eine entsprechende Änderungsnachricht (CLF_RQ_NOTIFY_CHANGE) schicken. Das Surrogat3 ändert daraufhin die den Zeitstempel (change_inode()) und registriert die Änderung der Inode mittels inode_changed(). Surrogat1 meldet gleichzeitig den erfolgreichen Abschluss der Operation mit einer CLF_RS_RENAME-Nachricht an den Klienten.

Im Rahmen der Umbenennung werden auf allen beteiligten Surrogaten Metadaten geändert, so dass in der jeweils folgenden Aktualisierungsphase (Update-

Inodes) die Veränderungen an den Metadatenmanager gemeldet werden. Im Sequenzdiagramm 5.13 ist dies, wie auch schon in Abbildung 5.12, durch einen Riss dargestellt, weil der Zeitpunkt und die Reihenfolge der Aktualisierung der Surrogate nicht fest steht. Der Zeitpunkt der Aktualisierungen wird nicht durch die betrachtete Operation, sondern durch die Konfiguration des Dateisystems bestimmt.

Fehler Neben der bereits betrachteten Fehlersituation auf Surrogat2, kann es zu einer weiteren Fehlersituation kommen, wenn das Verzeichnis nDir bereits voll ist. In diesem Fall schlägt der Aufruf add_entry(nDir, nName, inode) fehl.

Aufgrund dieses Fehlers muss Surrogat1 wiederum Surrogat2 kontaktieren, um die Datei wieder in ihrem ursprünglichen Verzeichnis uDir einzutragen. Surrogat1 meldet anschließend den Fehler an den Klienten.

In Abschnitt 5.4.4.1 wird ein Fehlerzustand bei der *Notify Change*-Anfrage beschrieben. Das in dem Abschnitt beschriebene Fehler-Szenario setzt vorraus, dass die Inode von einem anderen Klienten gelöscht wird. Weil aber die veränderte Inode zum Zeitpunkt der Aktualisierung nach der Umbenennung in keinem Verzeichnis aufgeführt ist, kann es bei der Umbenennung nicht zu dieser Situation kommen.

5.4.5 Aufwandsbetrachtung

In diesem Abschnitt wurde das Konzept der Surrogate vorgestellt. Zunächst wurde die, leicht vom Metadatenmanager abweichende, Architektur sowie die Modifikationen und Erweiterungen der gemeinsam verwendeten Module beschrieben. Weiterhin wurde die Kommunikation mit dem Metadatenmanager mit dem Fokus auf die Aktualisierungsprotokolle und schließlich die Zusammenarbeit der Surrogate bei Beantwortung bestimmter Klienten-Anfragen betrachtet.

Durch die Verteilung der Inodes auf eine Reihe von Surrogaten ist der Ablauf der Anfragen zum Erzeugen, Löschen und Umbenennen von Dateien und Verzeichnissen aufwändiger geworden, da diese Anfragen in der Regel mehr als ein Surrogat für ihre Bearbeitung benötigen. Solche *potentiell-kooperativen* Anfragen müssen zunächst zu einem der Surrogate geschickt werden, welches sie seinerseits an ein weiteres Surrogat weitergibt, das schließlich die Antwort an den Klienten schickt.

Insgesamt sind also maximal drei Netzwerkoperationen notwendig, während bei Einsatz eines zentralen Metadatenmanagers und bei den übrigen Anfragetypen, die nur ein Surrogat betreffen, nur zwei Netzwerktransfers notwendig sind. Unter der (sehr pessimistischen) Annahme, dass Netzwerkoperationen den Zeitbedarf einer Anfrage dominieren, ist die Bearbeitung einer einzelnen Anfrage durch Surrogate also um bis zu 50% zeitaufwändiger.

Auf der anderen Seite ist zu beachten, dass die betroffenen Anfragen relativ selten auftreten und die vielfach häufigeren einfachen Anfragen (aus Abschnitt 5.4.4.1) durch die Verteilung auf mehrere Dienstgeber schneller bearbeitet werden können. Dem vorliegenden Entwurf liegt die Annahme zu Grunde, dass gerade bei großer Anfragelast die Gesamtleistung des Dateisystems, trotz des höheren Aufwands für Anfragen dieser Kategorie, steigen wird. Diese Annahme wird durch die Messungen in Kapitel 6 bestätigt.

5.5 Zusammenfassung

CLF besitzt im Vergleich zu anderen Cluster-Dateisystemen, wie beispielsweise PVFS [17] und Lustre [9], neben den üblichen Komponenten *Klient*, *Daten-Server* und *Metadatenmanager*, als neuartige Komponente die *Metadaten-Surrogate*. Surrogate werden eingesetzt um den zentralen Metadatenmanager zu entlasten, indem sie stellvertretend für den Manager die Anfragen der Klienten beantworten. Die Surrogate teilen dazu die Metadaten mittels eines Hash-Wertes untereinander auf und sind jeweils nur für die auf ihnen gespeicherten Metadaten verantwortlich.

Die Klienten wurden so modifiziert, dass sie ihre Anfragen entsprechend des Hash-Wertes der betroffenen Metadaten an das zuständige Surrogat richten. Abgesehen von der variablen Adressierung des Dienstgebers, besteht für sie allerdings kein Unterschied zwischen der Benutzung des zentralen Managers und den verteilten Surrogaten.

Aufgrund der Abhängigkeiten zwischen Metadaten auf unterschiedlichen Surrogaten ist es notwendig, dass sie sich einerseits über den Manager und andererseits auch untereinander abstimmen. Bei bestimmten *potentiell-kooperativen* Klienten-Anfragen ist es notwendig, dass bis zu vier Surrogate an der Ausführung der Operation beteiligt sind, normalerweise genügt jedoch ein einzelnes Surrogat. Die Anzahl der Surrogate ist zur Laufzeit des Dateisystems fest, kann jedoch vor jedem Start neu festgelegt werden.

In diesem Kapitel wurde der Entwurf und die Implementierung der vier Komponenten des Dateisystems beschrieben, sowie die verwendeten Kommunikationsprotokolle. Das folgende Kapitel wird sowohl die Effektivität der Implementierung als Ganzes, wie auch die Leistungsverbesserungen durch den Einsatz der Surrogate anhand von Benchmarks untersuchen.

Kapitel 6

Auswertung

Dieses Kapitel präsentiert die Ergebnisse der quantitativen Evaluierung des Dateisystems *CLF* im Allgemeinen und dem Einsatz von Surrogaten im Besonderen. Die Evaluierung erfolgt anhand von *Medabench*, einem Benchmark, der die Metadaten-Leistung von Cluster-Dateisystemen misst. Im folgenden Abschnitt wird zunächst der *Medabench* vorgestellt, im Anschluss daran wird der verwendete Cluster beschrieben und die Ergebnisse der Messungen vorgestellt.

6.1 *Medabench*

Obwohl es eine große Zahl von Benchmarks für Dateisysteme gibt, sind nur wenige für Cluster-Dateisysteme verfügbar. In der Regel handelt es sich dabei um Erweiterungen lokaler Benchmarks, die in der Lage sind die Tests auf mehreren Knoten eines Clusters gleichzeitig auszuführen und die Einzelergebnisse zu einem Gesamtergebnis zu kombinieren. Noch seltener sind Benchmarks zur Messung der Metadatenleistung in Clustern. Tatsächlich konnte trotz intensiver Recherche nur ein Programm gefunden werden das dieses Ziel verfolgt. Leider ist *Metabench* [80] aufgrund verschiedener Schwächen nicht für die Evaluierung der allgemeinen Metadatenleistung eines Dateisystems geeignet. In Anhang B wird dieser Benchmark zusammen mit einigen Messungen von *CLF* und *PVFS* vorgestellt.

Aufgrund der Mängel von *Metabench* wurde *Medabench* (Metadata benchmark) im Rahmen einer Studienarbeit [27] am Institut für Programmstrukturen und Datenorganisation implementiert. Ziel war die Entwicklung eines Benchmarks, der die praktisch relevante Metadatenleistung einer Cluster-Dateisystems besser erfasst als *Metabench*.

6.1.1 Beschreibung

Bei *Medabench* handelt es sich um einen *Trace-basierten Benchmark*. Unter *Traces* versteht man Informationen, die in einem System während des laufenden Betriebs aufgezeichnet werden. Für *Medabench* wurden die in der Literatur als *Berkeley-Traces* bezeichneten Aufzeichnungen [1] als Grundlage verwendet. Sie sind frei verfügbar und können über das Internet heruntergeladen werden. Bei

der Analyse wurden neben den durchgeführten Operationen auch Klassen von Dateigrößen und Zugriffsmodi untersucht.

Die Dateien werden in einem an den Linux-Dateibaum angelehnten Pseudo-Dateibaum platziert, der drei Arten von Dateien anhand der Position im Baum unterscheidet:

1. Dateien die nur gelesen werden. (Verzeichnis: „`/etc`“)
2. Dateien die nur geschrieben werden. (Verzeichnis: „`/var`“)
3. Dateien die geschrieben und gelesen werden. (Verzeichnis: „`/tmp`“)

Innerhalb der Verzeichnisse existiert eine weitere Unterteilung der Dateien in *prozessglobale* und *prozesslokale* Dateien. Auf Erstere wird von allen Prozessen zugegriffen, während auf die zweitgenannten nur von einzelnen Prozessen zugegriffen wird. Da in einem Linux-Dateibaum nicht alle Dateien in einem Verzeichnis angeordnet sind, sondern in einer großen Zahl hierarchisch angeordneter Unterverzeichnisse, modelliert der Benchmark auch diese Eigenschaft:

Jedes der drei Basisverzeichnisse enthält eine Verzeichnishierarchie, wobei eine Mischung aus prozessglobale und prozesslokale Dateien nur in den Basisverzeichnissen zu finden sind und die Unterverzeichnisse, sowie die darin enthaltenen Dateien, entweder prozesslokal oder prozessglobal verwendet werden. Durch den Pseudo-Dateibaum soll erreicht werden, dass die Benchmark-Operationen auf Dateien und Verzeichnissen operieren, die einem realen Dateisystem möglichst nahe kommen.

Medabench läuft in drei Phasen ab, von denen die erste den Dateibaum vorbereitet, die zweite die eigentliche Messung durchführt und die dritte den benutzten Baum wieder löscht. Während der zweiten Phase wird mit verschiedenen Operationen, die zufällig anhand der aus den Traces bestimmten Wahrscheinlichkeiten bestimmt werden, auf die Dateien zugegriffen.

Diese drei Phasen werden für unterschiedliche Klientenzahlen wiederholt und jeweils der Metadaten-Durchsatz als Anzahl der Operationen pro Sekunde gemessen. Der Benchmark stellt seine Ergebnisse mit Hilfe von generierten HTML-Seiten dar. In Grafiken wird der ermittelte Durchsatz in Relation zu der Anzahl der Klientenprozesse, sowie für jeden einzelnen Benchmarklauf der Durchsatz jedes einzelnen Klientenprozesses und die Abweichung über die Zeit dargestellt. Anhand des Durchsatzes bei verschiedenen Klientenzahlen wird in diesem Kapitel die Skalierbarkeit der Dateisysteme untersucht.

Bei der Berechnung des Metadaten-Durchsatzes eines Dateisystems werden zwei unterschiedliche Formeln verwendet: Zum einen gibt es den *Gesamtdurchsatz* (`total`), der als Quotient der Anzahl aller ausgeführten Operationen und der längsten, von einem Prozess benötigten Zeit berechnet wird. Zum anderen wird der *kumulierte Durchsatz* (`accumulated`) aus der Summe der Durchsätze der einzelnen Klienten berechnet. Der *Gesamtdurchsatz* gibt Aufschluss über die Leistungsfähigkeit der Metadatenverarbeitung des Dateisystems als ganzes, weicht der *kumulierte Durchsatz* stark von diesem ab, ist das ein Zeichen dafür, dass die Leistung nicht auf allen Knoten gleichmäßig erbracht wurde. Bei den folgenden Messungen wird generell nur der Gesamtdurchsatz (`total`) verwendet.

Die Konfiguration von *Medabench* erfolgt über eine Konfigurationsdatei, die den Messergebnissen beigefügt wird, um eine einfache Reproduzierung der Mes-

sung zu ermöglichen. Die Konfigurationsdatei erlaubt eine weitgehende Beeinflussung der Messung, von der Definition der Abbruchkriterien bis zur Zahl der verwendeten Klienten und der verwendeten Schrittweite bei Messreihen mit unterschiedlichen Klientenzahlen.

Optional führt *Medabench* auch Lese- und Schreib-Operationen aus, die in ihrer Länge aus einer von sieben Klassen entstammen, deren Wahrscheinlichkeit aus der Analyse der Traces bestimmt wurde. Bei ausgeschalteten Daten-Operationen werden zwar nach wie vor die `read()`- und `write()`-Befehle aufgerufen, jedoch mit einer Länge von 0.

6.1.2 Auswertung der Traces

In diesem Abschnitt werden kurz die Ergebnisse der Auswertung der Berkeley-Trace-Dateien [1] zusammengefasst, welche die Zusammensetzung der Operationen von *Medabench* bestimmen. Mehr Details zu der Analyse der Trace-Dateien sind in der Studienarbeit von Oliver Denninger [27] zu finden. (Die in diesem Abschnitt angegebenen Tabellen entstammen dieser Arbeit.)

Aufgrund ihrer Häufigkeitsverteilung wurden fünf Dateisystemsaufrufe ausgewählt, die ausgeführt werden. Das Auslassen der restlichen Systemaufrufe wird durch die geringe Wahrscheinlichkeit dieser Aufrufe und der Tatsache, dass die meisten von ihnen auf die fünf ausgewählten Aufrufe abgebildet werden, gerechtfertigt. Die Operationen sind: *Status abrufen*, *Lesen*, *Schreiben*, *Erzeugen* und *Löschen* von Dateien. Öffnen und Schließen sind in den Lese- und Schreibaufrufen implizit enthalten. Dies ist kein Problem, da sich die Wahrscheinlichkeiten von Lese-/Schreibzugriffen und Öffnen-/Schließenoperationen in den Traces ungefähr entsprechen. Nach Reduktion auf die fünf genannten Operationen ergibt sich die Verteilung, die in Tabelle 6.1 zusammengefasst wurde.[1]

stat	0,75
read	0,15
write	0,05
create	0,03
delete	0,02

Tabelle 6.1: Verteilung der Operationen (Schritt 1)

	lesend	**schreibend**	**beides**
stat	0,3285	0,4000	0,2715
read	0,5530		0,4470
write		0,6575	0,3425
create			1,0000
delete			1,0000

Tabelle 6.2: Verteilung der Zugriffsmodi (Schritt 2)

Die Dateien unterscheidet der Benchmark nach Zugriffsmodus und Kontext. Die verwendeten Zugriffsmodi sind *nur lesender*, *nur schreibender* und *sowohl*

[1] Die um ein Prozent höhere Wahrscheinlichkeit für die Erzeugung einer Datei, im Vergleich zum Löschen einer Datei wird durch die Implementierung nicht umgesetzt. Der Benchmark verwendet eine vorgegebene Obergrenze an Dateien.

	lesend		schreibend		beides	
	global	lokal	global	lokal	global	lokal
stat	0,846	0,154	0,682	0,318	0,905	0,095
read	0,585	0,415			0,589	0,411
write			0,193	0,807	0,279	0,721
create					0,170	0,830
delete					0,134	0,866

Tabelle 6.3: Verteilung der Kontexte (Schritt 3)

lesender als auch schreibender Zugriff. Zur Vereinfachung bezieht sich der Zugriffsmodus auf die gesamte Lebensdauer einer Datei, d.h. auf eine Datei des Typs nur lesend wird während ihrer gesamten Lebensdauer nur lesend zugegriffen. Der Kontext einer Datei ist entweder *prozesslokal* oder *prozessglobal*. Auf eine *prozesslokale* Datei wird nur von einem Prozesses, auf eine *prozessglobale* Datei dürfen alle Prozesse des Benchmarks zugreifen.

Um die Skalierbarkeit zu messen, variiert der Benchmark die Anzahl der aktiven MPI-Prozesse, welche durch die MPI-Laufzeitumgebung auf die Knoten des Clusters verteilt werden. Je mehr Prozesse gleichzeitig aktiv sind, desto höher ist die Belastung des Dateisystems. Im Folgenden werden diese Prozesse *Klienten* oder *Klienten-Prozesse* genannt.

Der Benchmark wählt die auszuführende Operation mit Hilfe eines Zufallsgenerators und den angegebenen Tabellen in einem stufenartigen Auswahlprozess. Nachdem aufgrund der in Tabelle 6.1 angegeben Wahrscheinlichkeiten eine Operation ausgewählt wurden, wird mit Tabelle 6.2 der Zugriffsmodus und anschließend mit Tabelle 6.3 der Kontext bestimmt.

Der Benchmark kann Lese- und Schreiboperationen sowohl ohne Dateiinhalte als auch mit Pseudodateiinhalten ausführen. Ohne Dateiinhalte bekommt der entsprechende Systemaufruf als Lese- bzw. Schreibgrößenargument „0“ übergeben. Bei der Verwendung von Pseudodateiinhalten stammen die Daten aus einem mit Zufallswerten initialisierten Puffer. Für die bei den Pseudodateiinhalten zu verwendenden Größen wurden wiederum die Traces analysiert. Die Größen wurden dazu in sieben Klassen unterteilt, die das gesamte beobachtete Größenspektrum einschließen.

Tabelle 6.4 gibt die Verteilung der Größe der Leseoperationen aufgeschlüsselt nach Zugriffsmodus und Kontext an, während Tabelle 6.5 die Verteilung der Schreibgrößen zeigt. Die Spalte für nur schreibenden Zugriff entfällt bei Leseoperationen. Analog zu den Lesegrößen fehlt hier die Spalte nur lesender Zugriff. Die Tabellen werden als letzter Schritt der stufenweisen Operationsauswahl eingesetzt.

6.1.3 Kritik

Die größte potentielle Schwachstelle des Benchmarks liegt bei den zugrunde liegenden Trace-Dateien. Die Berkeley-Traces wurden auf Arbeitsplatzrechnern aufgezeichnet, die von Studenten, Dozenten und Mitarbeitern für Arbeiten wie Textverarbeitung, Softwareentwicklung und wissenschaftliche Programme verwendet werden. Es ist daher nicht klar, ob die Aufzeichnungen auf die Knoten eines Clusters übertragbar sind. Andererseits liegt die aus der Analyse resultierende Häufigkeitsverteilung der Operationen, im Vergleich zu der von *Meta-*

	lesend		beides	
	global	lokal	global	lokal
0 - 1 KB	0,3053	0,3469	0,4623	0,5091
1 - 4 KB	0,4131	0,4266	0,2053	0,2838
4 - 16 KB	0,2233	0,1675	0,2770	0,1672
16 - 64 KB	0,0462	0,0506	0,0455	0,0343
64 - 256 KB	0,0103	0,0080	0,0094	0,0048
256 KB - 1 MB	0,0018	0,0005	0,0004	0,0006
1 - 4 MB	0,0001	0,0000	0,0000	0,0001

Tabelle 6.4: Verteilung der Größe von Leseoperationen (Schritt 4 bei Leseoperationen)

	schreibend		beides	
	global	lokal	global	lokal
0 - 1 KB	0,2506	0,4197	0,4312	0,5140
1 - 4 KB	0,3366	0,2974	0,1810	0,2828
4 - 16 KB	0,3580	0,2260	0,2979	0,1678
16 - 64 KB	0,0498	0,0428	0,0723	0,0274
64 - 256 KB	0,0037	0,0086	0,0168	0,0059
256 KB - 1 MB	0,0011	0,0010	0,0006	0,0004
1 - 4 MB	0,0002	0,0045	0,0000	0,0017

Tabelle 6.5: Verteilung der Größe von Schreiboperationen (Schritt 4 bei Schreiboperationen)

bench, zumindest näher an der realen Verteilung.

Die Verwendung von Unterverzeichnissen ist besser gelöst als bei *Metabench*, allerdings basiert die Strukturierung und Größe der Verzeichnisse nicht auf der Untersuchung realer Dateisysteme, sondern orientiert sich an dem Kontext einer Datei.

Als Schwachstelle kann weiterhin gesehen werden, dass *Medabench* unter der Betreuung des Autors von *CLF* entstanden ist. Durch Einflussnahme auf die Entwicklung könnte er den Entwurf des Benchmarks so beeinflusst haben, dass *CLF* bei der Messung besonders gut abschneidet.

6.2 Der Kia-Cluster

Die Messungen wurden auf dem Kia-Cluster des Instituts für Programmstrukturen und Datenorganistation durchgeführt. Der Cluster besteht aus mit 16 HP rx2600 Rechnerknoten. Jeder Knoten ist mit zwei *Intel Itanium II*-Prozessoren ausgestattet, die mit jeweils 1.3 GHz getaktet sind und über 1 GB Speicher verfügen. Weiterhin verfügt jeder Knoten über eine 73 GB Ultra320 SCSI Festplatte mit einer Transferrate von 320 MB/s.

Als Betriebssystem kommt 64-Bit Linux aus der Redhat Advanced Server Distribution zum Einsatz (Kern-Version 2.6.9). Zur Kommunikation verwenden die Benchmark-Programme die MPI-Umgebung Parastation 4.1.1.

Die Knoten besitzen eine administrative und eine normale Fast-Ethernet-Schnittstelle, sowie das Hochgeschwindigkeits-Netzwerk *Infiniband*. Die Dateisystemkomponenten benutzen für die Kommunikation die TCP/IP-Schnittstelle dieses Netzwerks, weil die Verwendung der nativen Schnittstellen nicht nur eine

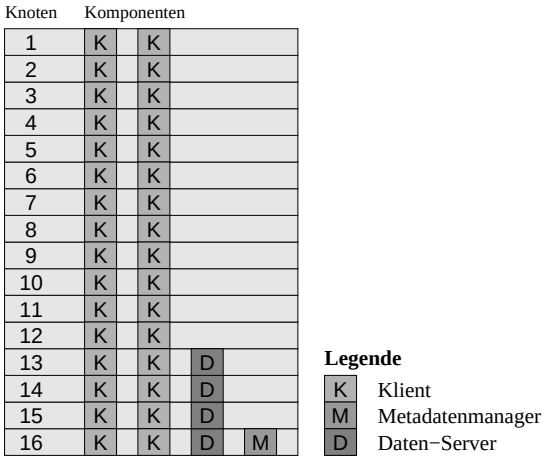


Abbildung 6.1: Die Standard-Konfiguration mit einem zentralen Metadatenmanager

Benutzerebenen-Implementierung, sondern auch eine Kern-Implementierung des *Paradis-Net*-Netzwerkprotokolls erfordert hätte. Zwar wurde die *Paradis-Net*-Unterstützung von Infiniband für die Benutzerebene im Rahmen einer Diplomarbeit implementiert [88], aufgrund der fehlenden Implementierung des Kern-Moduls wurde diese bei den Messungen jedoch nicht eingesetzt.

6.3 Vorüberlegungen und Parametrisierung von *Medabench*

Ziel der folgenden Messungen ist es, die Skalierbarkeit von bestimmten Dateisystem-Konfigurationen zu vergleichen. Dazu wird jede Messung mit einer unterschiedlichen Zahl an Klienten wiederholt um die Leistungsfähigkeit der Konfiguration bei unterschiedlichen Belastungen zu untersuchen. Der zur Verfügung stehende Cluster hat mit 16 Knoten zwar eine relativ geringe Größe, besitzt allerdings pro Knoten zwei Prozessoren, so dass es möglich ist, auf jedem Knoten zwei Klientenprozesse laufen zu lassen und auf diese Weise Messungen mit bis zu 32 Klienten durchzuführen.

Die Messungen sind sehr Metadaten-intensiv. Daher ist es schon mit dieser relativ geringen Knotenzahl möglich, den Metadatenmanager beziehungsweise die Surrogate voll auszulasten. Die Benchmarks bestehen ausschließlich aus Dateisystemoperationen und enthalten keine weiteren Berechnungen. Daher lässt sich bei den Messungen schon im messbaren Bereich, das heißt bei weniger als 32 Klienten, eine Sättigung erzielen und damit die maximale Metadatenleistung eines Dateisystems bestimmen.

Es ist weiterhin zu beachten, dass sich jeweils zwei Prozessoren eine Netzwerkschnittstelle teilen, wobei diese jedoch keinen Flaschenhals darstellt, wie die Messungen zeigen werden. Aufgrund der Tatsache, dass auf jedem Knoten zwei Klientenprozesse laufen, wurden die Messungen nur mit einer geraden Anzahl Klienten durchgeführt und dabei sicher gestellt, dass auf jedem beteiligten Kno-

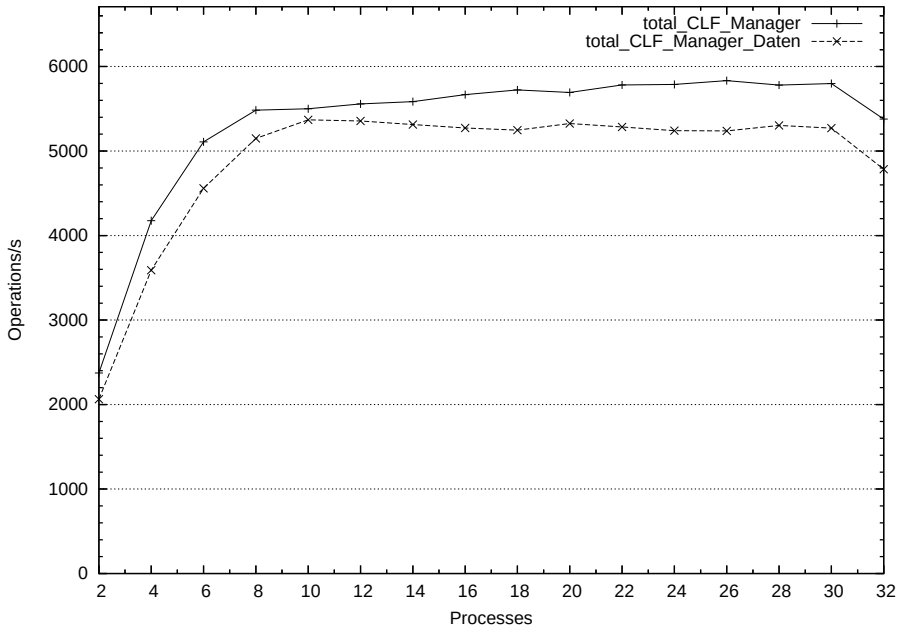


Abbildung 6.2: Vergleich zweier *Medabench*-Läufe: mit und ohne Datenoperationen

Konfiguration: Abb. 6.1

ten entweder keine oder zwei Klienten an der Messung beteiligt waren. Damit soll verhindert werden, dass ein Ungleichgewicht zwischen Klienten, die allein auf einem Knoten laufen und denen, die sich einen Knoten teilen, entsteht.

Die geringe Knotenzahl führt auch dazu, dass auf einigen Knoten sowohl Dienstgeber (Metadatenmanager, Daten-Server und Surrogate) und Dienstnehmer (Klienten) ablaufen. Die zusätzliche Belastung der vier Daten-Server- oder der Surrogat-Knoten durch Klientenprozesse könnte zu einer Verfälschung der Messung führen.

6.3.1 Voruntersuchung

Medabench führt neben den Metadatenoperationen auch Lese- und Schreib-Operationen durch (siehe Abschnitt 6.1). Aufgrund der geringen Größe des Clusters müssen die Knoten, auf denen die Daten-Server ablaufen gleichzeitig auch für Klienten-Prozesse verwendet werden. In diesem Abschnitt soll im Rahmen einer Voruntersuchung der Einfluss der Lese- und Schreib-Operationen auf die Messung untersucht werden.

Dazu werden zwei ähnliche Dateisystemkonfigurationen von *CLF* verwendet, die keine Surrogate einsetzen und der Konfiguration aus Abbildung 6.1 entsprechen. (Diese Konfiguration wird in den folgenden Abschnitten *Standard-Konfiguration* genannt.) Für die Messungen mit Lese- und Schreib-Operationen werden, abweichend von der *Standard-Konfiguration*, 16 Daten-Server verwendet, um die erhöhter Last auf diese Komponenten möglichst gleichmäßig auf die

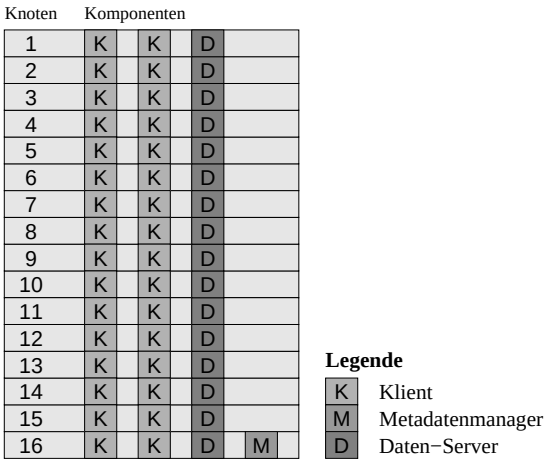


Abbildung 6.3: Die Standard-Konfiguration mit einem zentralen Metadatenmanager und 16 Daten-Servern

Knoten des Clusters zu verteilen. Diese Konfiguration ist in Abbildung 6.3 zu sehen.

Der andere Unterschied zwischen den beiden Messreihen ist der *Medabench*-Parameter `write_data`, der die Länge der Daten-Operationen beeinflusst. Alle Knoten, einschließlich der Knoten auf denen Dienstgeber laufen, werden für jeweils zwei Klienten verwendet. Bei Messungen mit weniger als 32 Klienten werden zuerst die Klienten auf Knoten mit geringerer Ordnungszahl verwendet, während die übrigen Klienten nicht zum Einsatz kommen.

Abbildung 6.2 zeigt das Ergebnis beider Messungen im Vergleich. Die Messreihe `CLF_Manager` wurde ohne Daten-Operationen durchgeführt, während bei `CLF_Manager_data` die Daten-Operationen mit in die Messung einfließen. (Der Präfix `total_` wird von dem Medabench-Benchmark vor den Namen der Messung gesetzt und bezeichnet den *Gesamtdurchsatz*, siehe Abschnitt 6.1.)

Erwartungsgemäß ist die Leistung des Dateisystems im zweiten Fall schlechter als im ersten. Während bei `CLF_Manager` ein maximaler Durchsatz von 5834 Operationen pro Sekunde erreicht wurde, konnte `CLF_Manager_Daten` nur 5369 Operationen pro Sekunde erreichen. Die erste Messung weist dieses Maximum bei einer hohen Klientenzahl auf, wogegen es bei Durchführung der Daten-Operationen bei 10 Klienten erreicht wird.

Beachtenswert ist der Abstand zwischen den Kurven, der bei steigender Klientenzahl immer mehr wächst. Daran lässt sich ablesen, dass der Einfluss der Daten-Operationen die Messung der Metadatenleistung des Dateisystems negativ beeinflussen.

Aufgrund des Einflusses der Daten-Operationen auf die Messung der Metadaten-Leistung werden die nachfolgenden Messungen mit *Medabench* zunächst *ohne Daten-Operationen* durchgeführt. In Abschnitt 6.8 wird der Einfluss der Daten-Operationen auf die Gesamtleistung des Dateisystems bei Verwendung von Surrogaten untersucht.

6.4 Messungen

Die nun folgenden Messungen gliedern sich in drei Teile:

Abschnitt 6.5 vergleicht *CLF* in seiner Standard-Konfiguration (ohne Surrogate) mit *NFS* und PVFS2.

Abschnitt 6.6 untersucht, wie sich die Metadatenleistung von *CLF* bei Einsatz von zwei Surrogaten im Vergleich zu einem zentralen Metadatenmanager verhält.

Abschnitt 6.7 präsentiert die Ergebnisse von *Medabench* beim Einsatz von mehr als zwei Surrogaten.

Abschnitt 6.8 untersucht den Einfluss von Daten-Operationen beim Einsatz von mehr als zwei Surrogaten.

6.5 Vergleich von NFS, PVFS und *CLF*

In diesem Abschnitt wird *CLF* anhand des Benchmarks mit NFS (siehe Abschnitt 2.2.2 oder [70]) und PVFS2 (siehe Abschnitt 2.2.5 oder [17]), Version 1.5.1 verglichen. Bei *CLF* wird eine Konfiguration mit einem zentralen Metadatenmanager verwendet, die dem Layout einer PVFS-Installation entspricht; Bei NFS agiert ein Knoten, der einen Teil des lokalen Dateisystems exportiert, als alleiniger Dienstgeber. Mit den Messungen soll zunächst untersucht werden, ob die Dateisysteme in ähnlicher Konfiguration eine vergleichbare Leistung zeigen.

Abbildung 6.1 zeigt die Verteilung der Komponenten auf die Knoten. Der Metadatenmanager bei *CLF* und PVFS kommt auf Knoten 16 zur Ausführung, die Daten-Server werden auf den Knoten 13, 14, 15 und 16 platziert. Für die NFS-Messungen wurde Knoten 16 als Dienstgeber verwendet. Alle Knoten, einschließlich der Dienstgeber, werden für Klientenprozesse verwendet, wir zuvor schon in der Voruntersuchung.

6.5.1 Diskussion

Abbildung 6.4 zeigt den Gesamt-Metadaten-Durchsatz (`total`) der Dateisysteme.

Der Metadatendurchsatz von *CLF* ist dem der anderen beiden Dateisysteme überlegen. Bei nur 2 Klienten erreicht *CLF* einen Durchsatz von 2372 Operationen Sekunde, während NFS 1318 und PVFS 243 Operationen pro Sekunde erzielt. Bei einer größeren Zahl von Klienten steigt dieser Wert bei allen Dateisystemen an, bis er sich schließlich mit 5834 (*CLF*), 4128 (NFS) und 445 (PVFS) Operationen pro Sekunde sein Maximum erreicht. Damit liegt die maximale Metadatenleistung von *CLF* um 41% über der von NFS und 13 Mal höher als die von PVFS.

Bei *CLF* und NFS ist zu beobachten, dass die Leistung bei der Verwendung von 32 Klienten-Prozessen im Vergleich mit 30 Klienten-Prozessen stark absinkt. Dies ist darauf zurück zu führen, dass die Klienten-Prozesse 31 und 32 auf Knoten 16 platziert werden. Weil Knoten 16 bei *CLF* als Metadatenmanager und bei NFS als Dienstgeber verwendet wird ist bei der Hinzunahme der

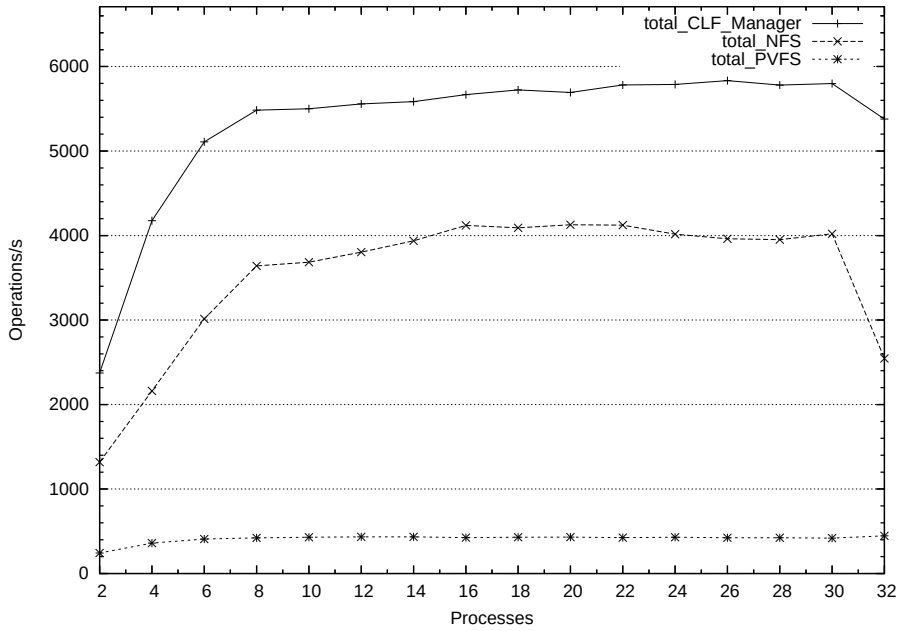


Abbildung 6.4: Vergleich von *CLF*, NFS und PVFS bei gleicher Konfiguration
Konfiguration: Abb. 6.1

beiden Klienten ein starker Leistungsabfall zu verzeichnen. Der in der Grafik dargestellte Durchsatz wird als Quotient aus der Gesamtzahl an Operationen und der vom langsamsten Klienten benötigten Zeit berechnet. Daher führt die schlechtere Leistung der beiden Klienten zu einem Absinken der Gesamtleistung. Die Metadatenleistung von PVFS bleibt bei 32 Klienten auf niedrigem Niveau stabil, weil der Metadatenmanager selbst nur geringe Last erzeugt und durch zusätzliche Klienten-Prozesse nicht eingeschränkt wird.

Bei allen drei Dateisystemen ist zu beobachten, dass die Metadatenverarbeitung bereits bei der Verwendung von acht Klienten ausgelastet ist. Bei mehr als acht Klienten steigt der Durchsatz weiterhin leicht an; dieser Anstieg ist auf eine noch bessere Auslastung der Dienstgeber zurückzuführen. Im Fall von NFS ist bei 16 Klienten die absolute Leistungsgrenze erreicht, bei mehr als 22 Klienten ist sogar ein Abfall der Leistung zu verzeichnen, dessen Ursache ein ungünstiges Überlastverhalten des NFS-Dienstgebers ist. *CLF* erreicht seine maximale Leistung bei 26 Klienten uns stagniert bei 28 und 30 Knoten. Aufgrund der geringen Größe des Kia-Clusters kann nicht festgestellt werden, ob bei weiter steigender Klientenzahl auch bei *CLF* ein Rückgang der Leistung eintritt.

6.5.2 *CLF* vs. NFS

Der Metadatendurchsatz von *CLF* liegt deutlich über dem der beiden Vergleichssysteme. Bei NFS ist dies hauptsächlich darauf zurück zu führen, dass NFS genaugenommen kein eigenständiges verteiltes Dateisystem ist, sondern eine Softwareschicht, die ein lokales Dateisystem entfernt zugreifbar macht (Ab-

schnitt 2.2.2). Jeder Zugriff eines Klienten wird auf dem Dienstgeber durch einen Zugriff auf das lokale Dateisystem des Dienstgebers ausgeführt. Diese Vorgehensweise ist dem Einsatz einer optimierten Metadatenverwaltung, die bei *CLF* zum Einsatz kommt (Abschnitt 5.3), unterlegen.

Dennoch ist überraschend, dass NFS, trotz des Einsatzes von Caches, nicht an die Leistung von *CLF* heranreichend kann.

6.5.3 *CLF* vs. PVFS

PVFS besitzt eine um eine Größenordnung schlechtere Leistung als *CLF*. Der Grund dafür ist in der Implementierung des Metadatenmanagers zu finden.

Die Messungen mit PVFS zeigen ab sechs Klienten ein asymptotisches Verhalten wie bei *CLF*. Aufgrund dessen kann der Klient als Ursache ausgeschlossen werden, weil ansonsten bei Hinzunahme von weiteren Klienten auch ein entsprechender Leistungszuwachs zu verzeichnen wäre. Die Messungen legen nahe, dass der Metadatenmanager die Leistung limitiert. Diese Sättigung ist bereits bei ca. 450 Anfragen pro Sekunde erreicht.

Der Grund, warum der Metadatenmanager von PVFS keinen mit *CLF* vergleichbaren Durchsatz zeigt, ist in der Architektur zu finden. In Kapitel 5 wurde der Aufbau des Managers von *CLF* ausführlich vorgestellt. Bei der Implementierung stand vor allem die Leistungsfähigkeit im Vordergrund, weshalb alle Funktionen auf relativ niedriger Ebene und unter Benutzung von Cache- und Speicher-Abbildungtechniken implementiert wurden. Durch die Benutzung von *Paradis-Net* ist die Implementierung auch implizit mehrfädig, da der Empfang der Nachrichten und die Verwaltung der Netzwerkverbindungen in einem eigenen Faden vorgenommen wird.

Demgegenüber folgt PVFS einem anderen Konzept, bei dem die Leistung nicht im Vordergrund stand. Statt einer spezialisierten Implementierung zur Verwaltung der Metadaten wird eine Universal-Datenbank eingesetzt. Die Berkeley Datenbank [64] wurde 1991 an der Universität Berkeley und später von *Sleepycat Software Inc.* entwickelt[2] und ist für OpenSource-Projekte wie PVFS lizenzfrei verfügbar. Obwohl die Datenbank angeblich sehr leistungsfähig ist, so bestehen doch Grund zu der Annahme, dass sie die Ursache für die vergleichsweise schlechte Leistung der Metadatenverwaltung von PVFS ist.

6.5.4 Zusammenfassung

Die Metadatenleistung von *CLF* ist den beiden Vergleichssystemen bereits ohne Einsatz von Surrogaten überlegen. Um den Nutzen der Surrogate zu bestimmen, wird in den folgenden Kapiteln die in Abbildung 6.1 dargestellte Standard-Konfiguration von *CLF* (mit einem Metadatenmanager, aber ohne Surrogate) als Vergleich herangezogen. Zum einen wurde in diesem Abschnitt gezeigt, dass *CLF* in dieser Konfiguration anderen verteilten Dateisystemen überlegen und daher eine hohe Messlatte für die Leistungsfähigkeit ist. Andererseits ermöglicht der Vergleich unterschiedlicher Konfigurationen des gleichen Dateisystems eine genauere Untersuchung der Leistungsänderungen, weil Einflussfaktoren, die nicht zwischen den Konfigurationen variiert werden, ausgeschlossen werden können.

[2] Im Februar 2006 wurde Sleepycat von der *Oracle Corporation* aufgekauft.

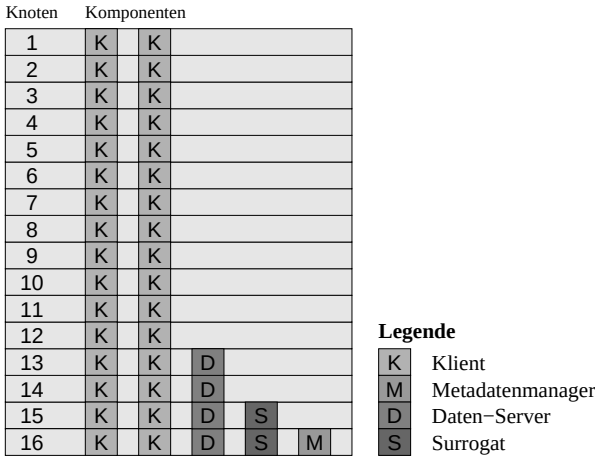


Abbildung 6.5: Die Konfiguration mit 2 Metadaten-Surrogaten

6.6 Zwei *CLF* Surrogate

In diesem Abschnitt wird untersucht, wie sich der Einsatz von zwei Surrogaten auf die Leistungsfähigkeit der Metadatenverwaltung auswirkt. In Abschnitt 6.5 wurde *CLF* mit NFS und PVFS verglichen und dabei wurde fest gestellt, dass die Metadatenverwaltung von *CLF*, auch ohne den Einsatz von Surrogaten, um ein vielfaches leistungsfähiger als die der Vergleichssysteme ist. Aus diesem Grund und zum Ausschluss anderer Faktoren als dem Einsatz von Metadaten-Surrogaten werden in diesem Abschnitt verschiedene Konfigurationen von *CLF* miteinander verglichen. Der Vergleich findet mit der in Abschnitt 6.3 beschriebenen Parametrisierung statt.

Die Ergebnisse des Vergleichs der Konfiguration mit zwei Surrogaten (`CLF_2_Surrogate`, Abbildung 6.5) mit der Standard-Konfiguration (`CLF_Manager`, Abbildung 6.1) sind in Abbildung 6.6 zu sehen. Um möglichst viele Knoten exklusiv für Klienten-Prozesse zu Verfügung zu haben, kommen bei den Konfigurationen mit Surrogaten, auf Knoten 16 sowohl ein Surrogat, wie auch der Metadatenmanager zur Ausführung. Metadatenmanager und Surrogat beeinflussen sich auf diesem Knoten allerdings nur geringfügig, da der Metadatenmanager mit seinem kleinen Aufgabenbereich nur geringe Last erzeugt.

Obwohl `CLF_2_Surrogate` bei weniger als 8 Klienten eine geringfügig schwächere Leistung als `CLF_Manager` zeigt (ca. 250 Operationen pro Sekunde weniger), erreicht die Konfiguration bei 8 Klienten erstmals einen höheren Durchsatz. Bei weiter steigenden Klientenzahlen verhält sich die Leistung zwar nicht so stabil wie bei Verwendung des Metadatenmanagers, erreicht aber ein Maximum von 7348 Operationen pro Sekunde und damit 26% mehr als `CLF_Manager`.

Beide Kurven zeigen einen ähnlichen Verlauf: Bei `CLF_Manager` flacht die Kurve bei 8 Klienten deutlich ab und strebt mit steigender Knotenzahl einem Maximum zu. Dieser Effekt ist auch bei `CLF_2_Surrogate` zu beobachten, allerdings flacht die Kurve erst bei 14 Klienten ab und weist ein lokales Maximum bei 24 Klienten auf. Dieses Maximum ist vermutlich dadurch bedingt, dass bei den Messungen mit 26, 28, 30 und 32 Klienten auf den Daten-Servern Klien-

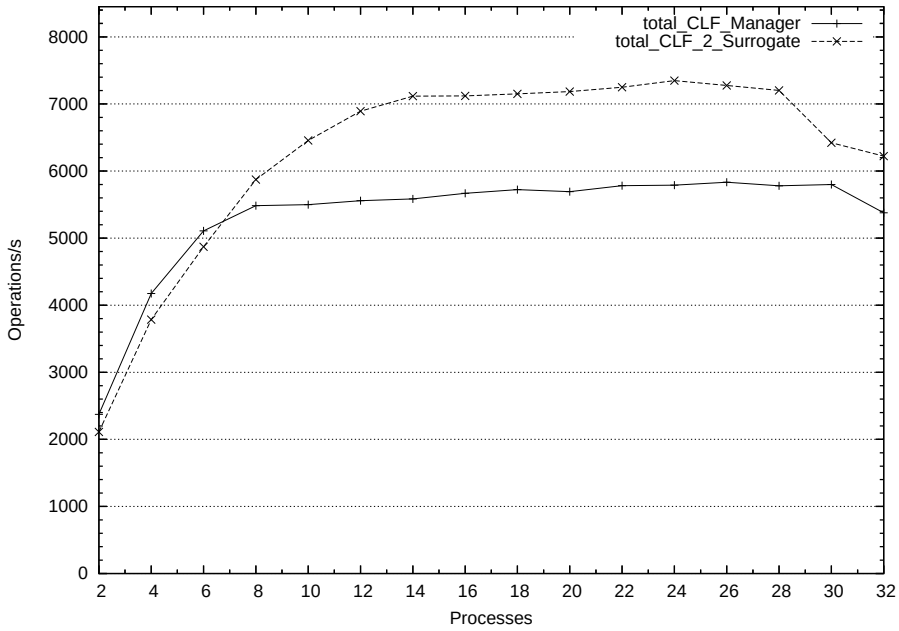


Abbildung 6.6: *CLF* mit Metadatenmanager vs. 2 Surrogate
Konfigurationen: *CLF Manager*: Abb. 6.1, *CLF 2 Surrogate*: Abb. 6.5

ten ablaufen, welche die Gesamtleistung des Dateisystems senken. Obwohl der Benchmark Datenoperationen (`read` und `write`) mit der Länge 0 durchführt, kontaktiert *CLF* dennoch die betroffenen Daten-Server. Diese Belastung ist für den leichten Abfall der Metadatenleistung bei den Datenpunkten 26 und 28 verantwortlich.

Für den starken Abfall der Leistung an den Punkten 30 und 32 sind die Metadaten-Surrogate verantwortlich. Sie sind voll ausgelastet und werden durch die gleichzeitige Ausführung der Klientenprozesse gebremst und bremsen umgekehrt auch die Klientenprozesse, wodurch die Metadatenleistung des gesamten Dateisystems um ca. 12% absinkt.

6.6.1 Diskussion

Insgesamt ist die Leistungssteigerung bei Verwendung von zwei Surrogaten zwar deutlich, fällt aber mit 26% geringer aus, als die erhoffte, annähernde Verdoppelung des Durchsatzes. Eine Verdoppelung ist jedoch aus folgenden Gründen unrealistisch:

Zum einen müssen die Surrogate bei bestimmten Aufgaben, wie dem Erzeugen und Löschen von Dateien zusammen arbeiten. Bei zwei Surrogaten liegt die Wahrscheinlichkeit für die Notwendigkeit einer solchen Kooperation bei 50%. Unter der Annahme, dass die Bearbeitung einer Anfrage auf einem Surrogat, unabhängig vom Typ der Anfrage, eine konstante Zeit in Anspruch nimmt, würde sich in diesem Fall die Antwortzeit verdoppeln, weil die Anfrage von beiden Surrogaten bearbeitet werden muss.

Im Durchschnitt steigt also die Bearbeitungszeit für eine potentiell-kooperative Anfrage um 50%, wodurch der Durchsatz auf $66,\overline{6}\%$ gesenkt würde. (Weil die Surrogate ausgelastet sind, wurde in der Berechnung die für die zusätzliche Kommunikation notwendige Zeit ausgeklammert.)

Allerdings haben die potentiell kooperativen Anfragen in *Medabench* nur eine Wahrscheinlichkeit von 5%, während eine Anfrage mit 95%iger Wahrscheinlichkeit nicht-kooperativ ist. Insgesamt würde daher der Durchsatz auf $98,\overline{3}\%$ pro Surrogat sinken, wobei die zwei Surrogate potentiell die doppelte Menge an Anfragen bearbeiten könnten wie der einzelne Metadatenmanager. Also könnte auch bei Einberechnung der potentiell-kooperativen Anfragen maximal mit einer Steigerung des Durchsatzes um $96,\overline{6}\%$ gerechnet werden.

Zum anderen spielt auch die zusätzliche Kommunikation zwischen den Surrogaten und dem Metadatenmanager eine Rolle. Die Surrogate schicken im Abstand von einer Sekunde eine Aktualisierungsnachricht zum Metadatenmanager. Geänderte Inodes werden zusammen in großen Paketen, während geänderte Verzeichnisse einzeln und vollständig an den Manager geschickt werden.

Die Aktualisierungen können der Leistung des Dateisystems auf zwei Arten schaden: Einerseits können die Nachrichten zu einer zusätzlichen Belastung des Netzwerks führen und andererseits können die Surrogate während der Aktualisierung keine Anfragen beantworten. Anfragen, die während der Aktualisierung auf einem Surrogat eintreffen, müssen warten, bis diese abgeschlossen ist.

Wie groß die Auswirkungen der Aktualisierungen sind, ist nur schwer einzuschätzen. Daher wurden versuchsweise die Registrierungsfunktionen des VII-Moduls, die bei Änderungen von Inodes und Verzeichnissen aufgerufen werden, vollständig ausgeschaltet. Messungen mit zwei Surrogaten zeigten allerdings keinen messbaren Unterschied zu der in Abbildung 6.6 dargestellten Leistungskurve. Der zusätzliche Aufwand für die Aktualisierungsnachrichten scheint also vernachlässigbar klein zu sein.

Nach Ausschluss möglicher Erklärungen für die relativ geringe Leistungssteigerung beim Einsatz von 2 Surrogaten (zusätzliche Kommunikation zwischen den Surrogaten und zusätzliche Kommunikation zwischen den Surrogaten und dem Manager) bleibt offen, warum die Leistung nicht in dem erwarteten Maße ansteigt. Um die Ursachen weiter zu erforschen wurde der Benchmark modifiziert, um das Verhalten von *CLF* bei verschiedenen Anfragetypen zu untersuchen.

6.6.2 *Einfache* vs. *potentiell-kooperative* Operationen

Das Erstellen und Löschen von Dateien gehört zu den aufwändigsten Operationen beim Einsatz von Metadaten-Surrogaten (siehe auch Abschnitt 5.4.4.2). Diese Operationen erfordern den Zugriff auf zwei unterschiedliche Inode-Objekte: Einerseits das Verzeichnis und andererseits die betroffene Datei. Weil die Zuteilung der Inodes zu den Surrogaten zufällig erfolgt, sind bis zu zwei Surrogate an der Bearbeitung beteiligt. Metadaten-Operationen, an denen mehrere Surrogate beteiligt sind, werden *potentiell-kooperativ* genannt. Operationen können andererseits auch *einfache* Operationen sein, wenn sie von einem einzelnen Surrogat bearbeitet werden können. Dazu gehört beispielsweise das Öffnen von Dateien oder das Suchen einer Datei in einem Verzeichnis.

In diesem Abschnitt wird die Leistung von *potentiell-kooperativen* und *einfachen* Operationen bei Einsatz von zwei Surrogaten (Konfiguration: Abb. 6.5)

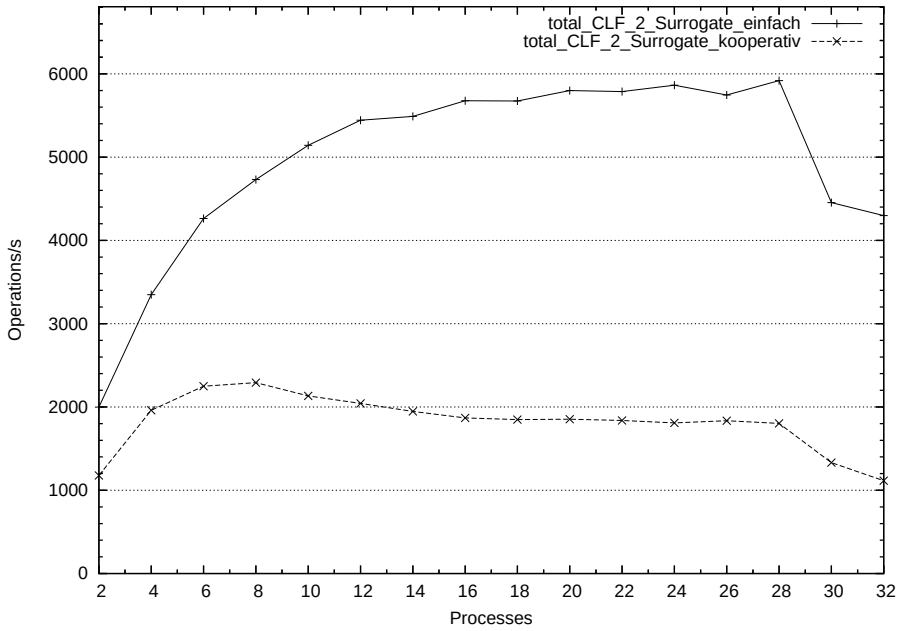


Abbildung 6.7: *CLF* mit 2 Surrogaten bei *potentiell-kooperativen* und *einfachen* Operationen

Konfiguration: *CLF 2 Surrogate*: Abb. 6.5

verglichen. Es ist dabei zu beachten, dass es in der Praxis nicht möglich ist, durch einen Klienten ausschließlich *potentiell-kooperative* Anfragen generieren zu lassen, weil diese immer auf *einfachen* Anfragen basieren. Es ist beispielsweise beim Erstellen einer Datei zunächst notwendig, eine Pfadsuche durchzuführen, um so die Inode-Nummer des Verzeichnisses zu bestimmen. Dazu stellt der Klient *einfache* Anfragen an die Surrogate. Jede *potentiell-kooperative* Operation eines Klienten enthält also auch immer einige *einfache* Anfragen an die Surrogate.

Um die Auswirkungen von *potentiell-kooperativen* Anfragen zu untersuchen, wurde der *Medabench*-Benchmark so manipuliert, dass er einerseits ausschließlich *einfache* (`CLF_2_Surrogate_einfach`) und andererseits ausschließlich *potentiell-kooperative* (`CLF_2_Surrogate_kooperativ`) Operationen durchführt. Dabei wird das aufgrund der Trace-Dateien festgelegte Verhältnis zwischen den verwendeten Operationen beibehalten. Wie bereits im vorangegangen Abschnitt beschrieben wurde, ist zu beachten, dass rein *potentiell-kooperative* Anfragen von dem Benchmark nicht generiert werden können, weil Erstellen und Löschen von Dateien immer auch eine Pfadsuche beinhaltet, die sich aus *einfachen* Anfragen an die Metadaten-Surrogate zusammen setzt.

Das Ergebnis der Messungen ist in Abbildung 6.7 zu sehen. Es ist zu erkennen, dass das Dateisystem bei Verwendung von ausschließlich *einfachen* Operationen beinahe dreimal soviel Durchsatz zeigt, wie bei den *kooperativen Operationen*. Während `CLF_2_Surrogate_einfach` insgesamt ein asymptotisches Verhalten zeigt, das bei 28 Klientenprozessen sein Maximum zeigt, erreicht `CLF_2_-Surrogate_kooperativ` einen Maximaldurchsatz von 2291,5 Operationen pro

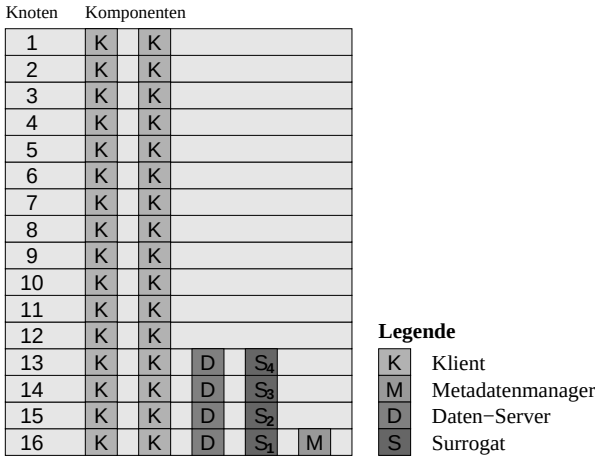


Abbildung 6.8: Die Konfiguration mit bis zu 4 Metadaten-Surrogaten

Sekunde bei 8 Klientenprozessen und sinkt dann asymptotisch auf 1803,1 Operationen pro Sekunde ab. Beide Kurven zeigen den typischen Einbruch des Durchsatzes, der eintritt, wenn die Knoten auf denen Surrogate laufen auch für Klientenprozesse verwendet werden.

Der geringere Durchsatz der *potentiell-kooperativen* Operationen ist einerseits darauf zurück zu führen, dass eine Operation von mehreren Surrogaten bearbeitet werden muss und andererseits darauf, dass das Einfügen und Löschen von Dateien eine höhere Bearbeitungszeit benötigt als die *einfachen* Operationen. Aufgrund dessen sind diese Operationen nicht nur um 50% langsamer, wie man erwarten könnte, wenn man annimmt, dass die Bearbeitung einer Anfrage auf einem Surrogat eine konstante Zeit benötigt und 50% aller Anfragen von zwei Surrogaten bearbeitet werden müssen.

6.7 Mehr als zwei Surrogate

Nun soll das Verhalten von *CLF* beim Einsatz von mehr als zwei Surrogaten untersucht werden. Dazu wurde das Dateisystem zunächst mit drei und vier Surrogaten (Abschnitt 6.7.1) und anschließend mit 8 und 16 Surrogaten konfiguriert (Abschnitt 6.7.2). Die Messung der Metadatenleistung wurde wiederum mit *Medabench* und den in Abschnitt 6.3 beschriebenen Einstellungen durchgeführt.

6.7.1 Bis zu 4 Surrogate

Abbildung 6.8 zeigt das Layout des Dateisystems für die Messung mit bis zu 4 Surrogaten. Für die Messreihen mit weniger als vier Surrogaten wurden die Surrogate mit den höheren Ordnungsnummern deaktiviert.

Der Verlauf der vier Messreihen ist in Abbildung 6.9 zu sehen. Die Konfigurationen `CLF_Manager` und `CLF_2_Surrogate` wurden bereits im vorangegangenen Abschnitt kommentiert. Die dort beobachteten Unterschiede setzen sich bei

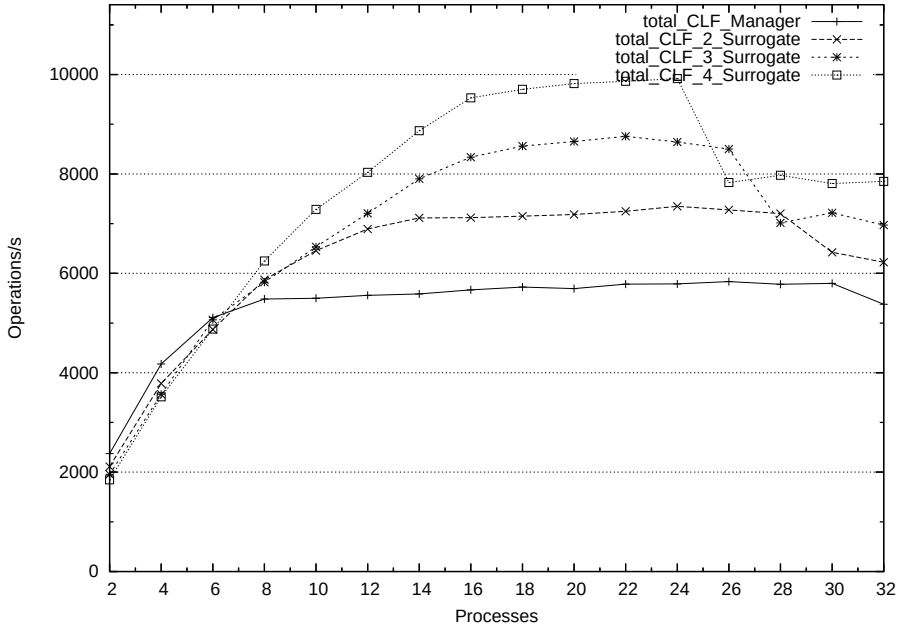


Abbildung 6.9: *CLF* mit Metadatenmanager vs. 1, 2, 3 Surrogate
Konfigurationen: *CLF Manager*: Abb. 6.1, *CLF 2/3/4 Surrogate*: Abb. 6.8

Konfiguration	Minimum	Durchschnitt	Maximum
	in Operationen / Sekunde		
`CLF_Manager`	2371,91	5326,83	5834,19
`CLF_2_Surrogate`	2111,07	6267,62	7348,48
`CLF_3_Surrogate`	1955,55	6918,83	8758,52
`CLF_4_Surrogate`	1844,96	7561,14	9919,34

Tabelle 6.6: Statistische Daten zu den Messreihen in Abbildung 6.9

`CLF_3_Surrogate` und `CLF_4_Surrogate` fort: Mit steigender Surrogat-Zahl steigt der maximale Durchsatz um jeweils zirka 1300 Operationen pro Sekunde. Auch zeigt die Leistung der Konfigurationen einen Einbruch, wenn auf den Metadaten-Surrogaten auch Klientenprozesse ablaufen. Dies ist bei drei Surrogaten ab 28 Klienten und bei vier Surrogaten ab 26 Klienten der Fall.

Das Ausmaß des Leistungseinbruchs, der eintritt, sobald die Dienstgeber (Surrogate und Metadatenamanager) auch für Klientenprozesse verwendet werden, steigt mit größerer Zahl an Surrogaten. Betrachtet man allerdings nur die Metadatenleistung bei 32 Klientenprozessen, steigt diese trotzdem bei steigender Surrogatzahl. Die größten Leistungssteigerungen lassen sich allerdings erreichen, wenn dedizierte Cluster-Knoten für die Metadatenverarbeitung verwendet werden.

Tabelle 6.6 illustriert diese Beobachtung: Der maximal erreichte Durchsatz steigt mit jedem zusätzlichen Surrogaten um 1500–1200 Operationen pro Sekunde. Minimalen Durchsatz erreichen alle Konfigurationen bei zwei Klienten,

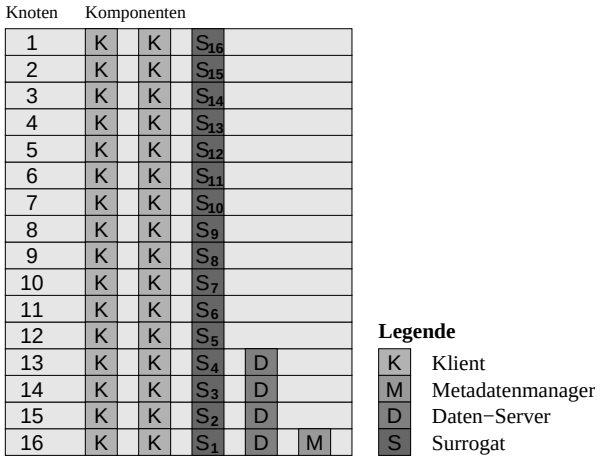


Abbildung 6.10: Die Konfiguration mit bis zu 16 Metadaten-Surrogaten

wobei dieser Wert mit steigender Surrogat-Zahl sinkt. Aufgrund dieses Verhaltens schneiden sich die Kurven zwischen 4 und 8 Klienten; anschließend besitzen sie einen asymptotischen Verlauf.

Insgesamt zeigen die Konfigurationen mit bis zu 4 Surrogaten das erwartete und erwünschte Verhalten. Allerdings bleiben die Leistungssteigerungen, wie auch schon bei zwei Surrogaten, hinter den Erwartungen zurück: Eine maximale Leistungssteigerung von 70% (im Vergleich mit der Standard-Konfiguration) wird mit vier Surrogaten auf dedizierten Dienstgeber-Knoten erreicht.

6.7.2 Bis zu 16 Surrogate

Im vorangegangen Abschnitt 6.7.1 wurde beobachtet, dass mit steigender Surrogatenzahl auch die Metadatenleistung des Dateisystems steigt, für eine maximale Leistungssteigerung aber dedizierte Surrogaten-Knoten notwendig sind. Aufgrund der geringen Größe des Kia-Clusters, ist bei mehr dedizierten Surrogaten-Knoten immer weniger „Platz“ für die Klienten Prozesse.

Die bisherigen Messungen zeigen, dass die maximale Metadatenleistung bei einer größeren Zahl von Surrogaten auch erst bei einer größeren Zahl von Klienten erreicht wird. Je mehr Surrogate eingesetzt werden, desto mehr Klienten-Prozesse benötigt man, um diese auszulasten. Bei der Verwendung von vier Surrogaten wird mit 16 Klientenprozessen (8 Knoten) ein Durchsatz von 9531,84 Operationen pro Sekunde erreicht. Dies entspricht bereits 96,1% der mit 24 Klienten erzielten Maximalleistung von 9919,34 Operationen pro Sekunde. Verwendet man also 8 Surrogate wird es nicht möglich sein, diese bereits mit den 8 für Klientenprozesse verbliebenen Knoten auszulasten.

An den Messungen mit 8 Surrogaten und insbesondere mit 16 Surrogaten wird daher abzulesen sein, inwiefern die Koexistenz von Klienten und Surrogaten auf einem Knoten bei einer starken Verteilung der Metadaten möglich ist. Abbildung 6.10 zeigt die Platzierung der Surrogate auf den Knoten. Wie in den vorangegangenen Abschnitten, werden beim Einsatz von weniger Surrogaten zunächst diejenigen mit niedriger Ordnungsnummer verwendet. Die Ergebnisse der

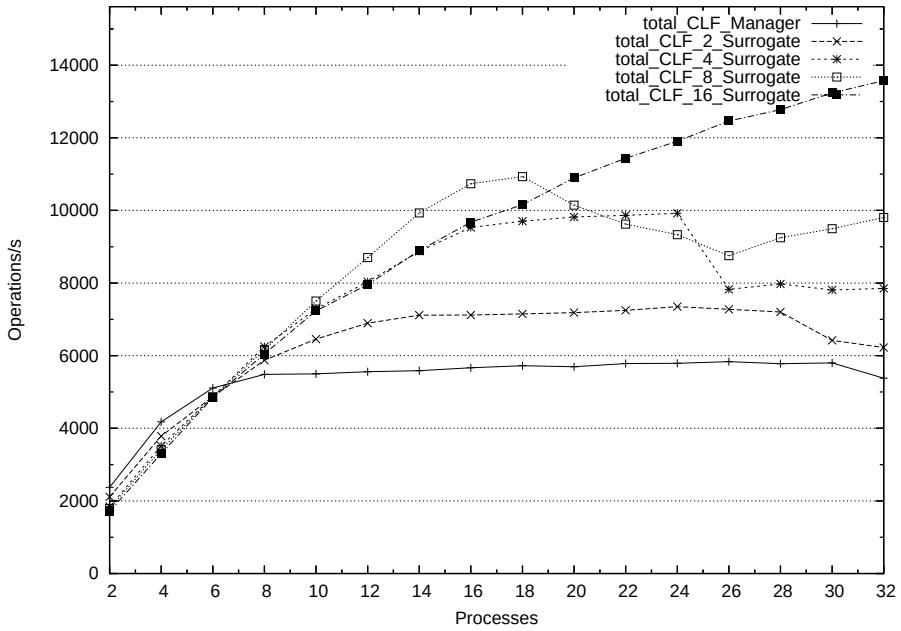


Abbildung 6.11: *CLF* mit Metadatenmanager vs. 2, 4, 8, 16 Surrogate
Konfigurationen: *CLF Manager*: Abb. 6.1, *CLF 2/4/8/16 Surrogate*: Abb. 6.10

Messungen sind, zusammen mit den Messungen mit der Standard-Konfiguration und denen mit zwei und vier Surrogaten in Abbildung 6.11 zu sehen.

Die Konfiguration `CLF_8_Surrogate` liefert zwischen 8 und 18 Kientenprozessen die höchste Leistung aller bisher betrachteten Konfigurationen, ab 20 Prozessen sinkt sie allerdings ab und hat bei 22 und 24 Klienten zeitweise sogar geringeren Durchsatz als die Konfiguration mit 4 dedizierten Surrogaten. Das Absinken der Leistung ist jedoch nicht so rapide, wie bei den Konfigurationen mit deutlich weniger Surrogaten. Das ist darauf zurück zu führen, dass die Zusätzliche Belastung eines Surrogat-Knoten mit Klientprozessen aufgrund der großen Anzahl an Surrogaten keinen so großen Einfluss auf die Gesamtleistung des Dateisystems hat, wie bei den zuvor betrachteten Konfigurationen. Das erklärt auch warum mit 18 Klienten eine höhere Leistung erreicht wird als mit 16 Klienten, obwohl bereits auf einem der Knoten beide Arten von Prozessen ablaufen.

Weiterhin auffällig ist, dass im weiteren Verlauf, zwischen 26 und 32 Klienten eine geringfügige Steigerung der Metadatenleistung zu beobachten ist. Offenbar ist eine gleichmäßige Belastung der Surrogat-Knoten der Gesamtleistung des Dateisystems nicht so abträglich, wie zunächst bei wenigen Surrogaten beobachtet.

Das wird auch von der Messung mit Konfiguration `CLF_16_Surrogate` bestätigt. Bei dieser Konfiguration wird jeder Knoten des Clusters an der Metadatenverwaltung beteiligt. Die Messung bestätigt die zuvor beobachteten Trends: Der minimale Durchsatz sinkt im Vergleich zu allen anderen Konfigurationen weiter und der maximale Durchsatz steigt (vergleiche Tabelle 6.7). Im Gegen-

Konfiguration	Minimum	Durchschnitt	Maximum
	in Operationen / Sekunde		
CLF_Manager	2371,91	5326,83	5834,19
CLF_2_Surrogate	2111,07	6267,62	7348,48
CLF_4_Surrogate	1844,96	7561,14	9919,34
CLF_8_Surrogate	1790,21	8151,95	10931,27
CLF_16_Surrogate	1710,26	9131,46	13575,72

Tabelle 6.7: Statistische Daten zu den Messreihen in Abbildung 6.11

satz zu allen anderen Konfigurationen zeigt diese allerdings keinen Einbruch, weil ohnehin auf allen Knoten Surrogate ablaufen. Aufgrund der gleichmäßigen Verteilung der Metadaten-Verwaltung skaliert die Leistung gut mit der Belastung durch die Klientenprozesse und zeigt im Messbereich auch keine Auslastung, die bei den übrigen Messreihen an einer starke Abflachung der Kurve bei steigenden Klientenzahlen ablesbar ist. Der maximale Metadatendurchsatz von 13575,72 Operationen pro Sekunde wird bei 32 Klientenprozessen erreicht; dies entspricht einer Steigerung von 132,7% im Vergleich zu der maximalen Leistung des zentralen Metadatenmanagers.

6.7.3 Skalierbarkeit

An dem Vergleich der Messreihen lässt sich ablesen, dass auch mit zunehmender Surrogat-Zahl die Durchsatzsteigerung fortsetzt. Obwohl die Steigerung nicht in dem erwünschten Maße erfolgt, belegen die Ergebnisse des Benchmarks dennoch, dass sich mit Erweiterung des *CLF*-Dateisystem um Metadaten-Surrogate die Leistungsfähigkeit der Metadatenverwaltung steigern lässt. Die Messungen mit bis zu 16 Cluster-Knoten illustrieren, dass eine möglichst gleichmäßige Verteilung der Metadatenverwaltung zu der besten Skalierbarkeit führt.

6.8 Surrogate und Daten-Operationen

Die bisherigen Messungen untersuchten nur die Metadaten-Leistung des Dateisystems, ohne Beachtung der Daten-Operationen. Der Grund für diese Vorgehensweise war einerseits die zugrundeliegende Annahme, dass die Metadaten-Operationen die Skalierbarkeit stärker bedrohen als die Daten-Operationen. Andererseits ist es durch die geringe Größe des Clusters notwendig, einzelne Knoten gleichzeitig als Daten-Server, Surrogat und Klient zu verwenden. Ob trotz diese Mehrfachbelastung die Skalierbarkeit des Dateisystems erhalten bleibt, wird in diesem Abschnitt untersucht.

6.8.1 Konfiguration des Dateisystems

Um die Belastung der Knoten, die durch die Daten-Operationen entsteht, möglichst gleichmäßig auf die Knoten des Clusters zu verteilen, wird die im letzten Abschnitt verwendete Konfiguration (Abbildung 6.10) für die Messungen so geändert, dass auf jedem Knoten ein Daten-Server läuft, wie in Abbildung 6.12 zu sehen ist. Durch diese Änderung werden Unregelmäßigkeiten verhindert, die

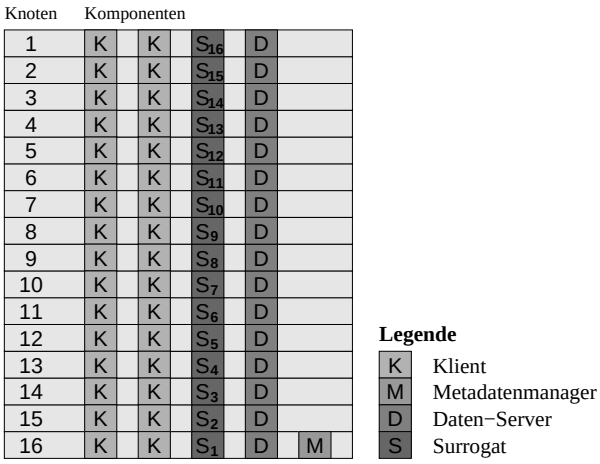


Abbildung 6.12: Die Konfiguration mit bis zu 16 Metadaten-Surrogaten und 16 Daten-Servern

durch die ungleichmäßige Belastung der Knoten entstehen, auf denen mehrere Dienstgeber und Klienten ablaufen. Für die Messungen ohne Datenoperationen wird die ursprüngliche Konfiguration verwendet, weil in diesem Fall die Daten-Server nur unerhebliche Last erzeugen; die Messreihen entsprechen denen, die in Abschnitt 6.7 vorgestellt wurden.

6.8.2 Messungen mit 2, 4 und 16 Surrogaten

In Abbildung 6.13 sind die Ergebnisse der Messung bei Verwendung von zwei Surrogaten zu sehen. `CLF_2_Surrogate` zeigt die Leistung des Dateisystems ohne und `CLF_2_Surrogate_Daten` mit Daten-Operationen. Der Verlauf der Kurven ähnelt einander, auch wenn die Leistung des Dateisystems bei Durchführung von Daten-Operationen generell niedriger ist als ohne. Dennoch ist der Unterschied mit ca. 200–1000 Operationen pro Sekunde bei relativ gering. Die maximale Leistung von `CLF_2_Surrogate` beträgt 7348,48 Operationen pro Sekunde, bei `CLF_2_Surrogate_Daten` werden 6912,69 Operationen pro Sekunde erreicht.

Bei Einsatz von vier Surrogaten (Abbildung 6.14) zeigt sich ein größerer Leistungsunterschied: Während ohne Daten-Operationen maximal 9919,34 Operationen pro Sekunde erreicht werden, kann das Dateisystem mit Daten-Operationen maximal 8873,90 Operationen pro Sekunde leisten. Beide Konfigurationen erreichen die maximale Leistung bei 24 Klienten, bei 26 Klienten ist der bereits bekannte Leistungsabfall zu beobachten, der darauf zurück zu führen ist, dass die Cluster-Knoten 13–16 als Metadaten-Surrogate verwendet werden. Bei weniger als 26 Klienten lässt sich beobachten, dass der Abstand zwischen den beiden Kurven mit steigendeer Klientenzahl zunimmt. Ursächlich dafür ist die steigende Belastung der Daten-Server und damit aller Cluster-Knoten, welche die Leistungsfähigkeit der übrigen Komponenten (Klienten und Surrogate) verringert.

Verwendet man 16 Surrogate, lässt sich diese Tendenz noch deutlicher ab-

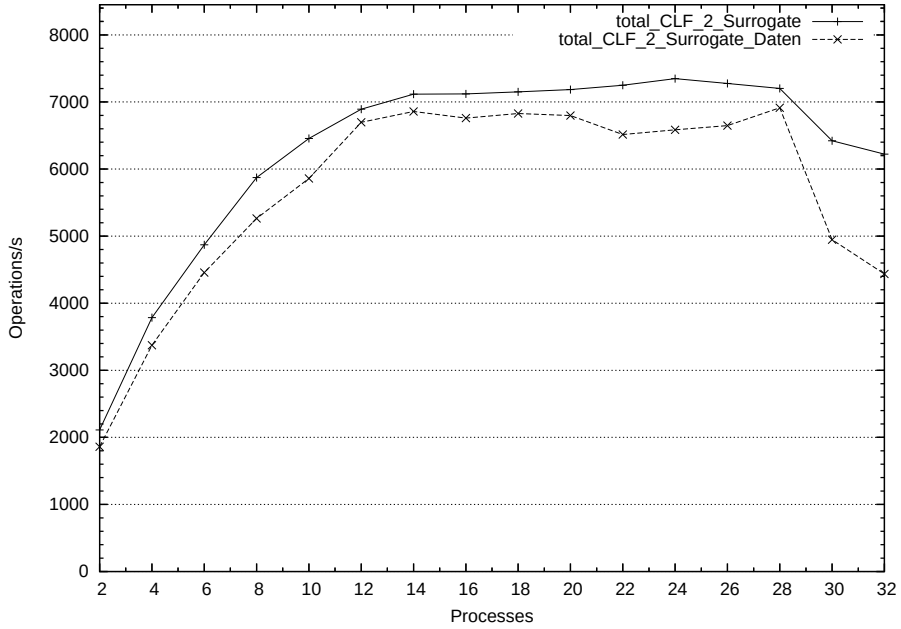


Abbildung 6.13: *CLF* mit 2 Surrogaten, mit und ohne Daten-Operationen
Konfiguration: Abb. 6.12

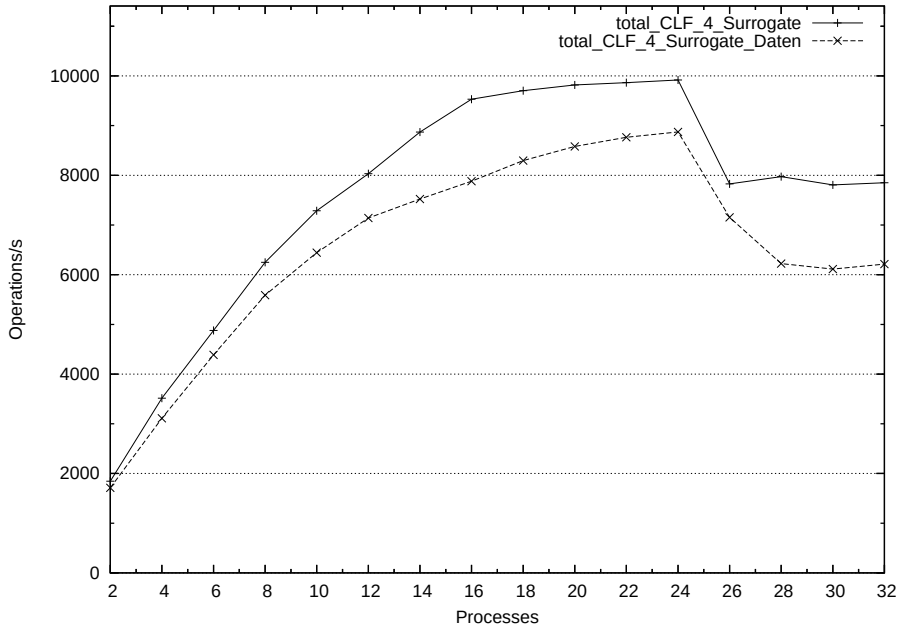


Abbildung 6.14: *CLF* mit 4 Surrogaten, mit und ohne Daten-Operationen
Konfiguration: Abb. 6.12

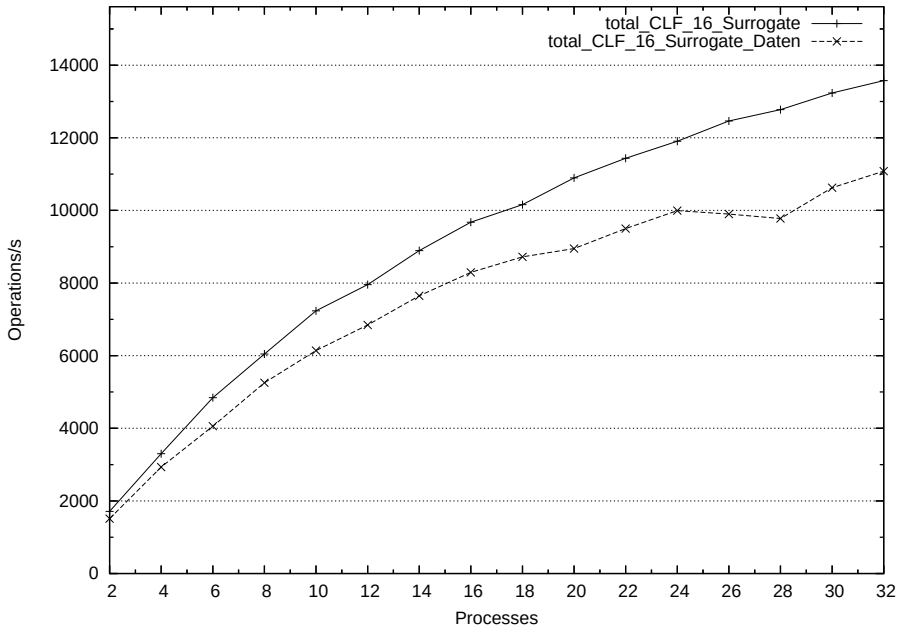


Abbildung 6.15: *CLF* mit 16 Surrogaten, mit und ohne Daten-Operationen
Konfiguration: Abb. 6.12

lesen (Abbildung 6.15). Je mehr Operationen das Dateisystems ausführt, desto stärker wird der Einfluss der Daten-Operationen auf die Leistung. Bei 16 Klienten und einer Leistung von 9675,88 Operationen pro Sekunde ohne und 8293,65 Operationen pro Sekunde mit Daten-Operationen, beträgt der Unterschied noch 1382,23 Operationen pro Sekunde. Bei 32 Klienten ist ein Unterschied von 2593,83 Operationen pro Sekunde (13575,72 mit und 11081,89 ohne) zu beobachten. Diese Ergebnisse sind nicht überraschend, da bereits ohne Daten-Operationen der Cluster vor ausgelastet war. Durch die zusätzliche Belastung der Daten-Server wird die Leistungsfähigkeit der Metadaten-Surrogate und der Klienten, die, wie die Daten-Server, auf allen Knoten des Clusters ablaufen, eingeschränkt. Dieser Effekt war bereits in der Voruntersuchung (Abschnitt 6.3.1, Abbildung 6.2) zu sehen und verstärkt sich, je mehr Operationen pro Sekunde ausgeführt werden.

6.8.3 Ergebnis

Insgesamt zeigen die Messungen, dass die Leistungsfähigkeit von *CLF* in allen Konfigurationen von der Metadaten-Leistung dominiert wird. Die Belastung der Daten-Server steigt wenn mehr Operationen -und damit auch Daten-Operationen- aufgeführt werden. Damit steigt auch der Einfluss der Daten-Server auf die Gesamtleitung des Dateisystems, insbesondere auch deshalb, weil sich, aufgrund der geringen Größe des Clusters, mehrere Komponenten des Dateisystems einen Cluster-Knoten teilen müssen und sich dadurch gegenseitig beeinflussen.

6.9 Zusammenfassung

Anhand des Metadatenbenchmarks *Medabench* wurden in diesem Kapitel die Dateisysteme NFS, PVFS und *CLF* auf einen 16-Knoten Itanium 2 Cluster vermessen. Dabei konnte nachgewiesen werden, dass die *CLF*-Metadatenverwaltung höheren Durchsatz als die beiden Vergleichssysteme erreicht. Weiterhin wurde der Nutzen der Metadatenverteilung mittels Metadaten-Surrogaten durch den Vergleich unterschiedlicher Konfigurationen von *CLF* evaluiert.

Es konnte gezeigt werden, dass *CLF* die Surrogate erfolgreich zur Verbesserung der Metadatenverwaltung einsetzen kann. Beim Einsatz von zwei Surrogaten lässt sich der vom *Medabench*-Benchmark gemessene Durchsatz im Vergleich zu einer Konfiguration mit einem zentralen Metadatenmanager auf dem Cluster um maximal 26% steigern. Beim Einsatz von vier Surrogaten steigert sich der Durchsatz um maximal 70% im Vergleich zur Standard-Konfiguration.

Die größte Steigerung des Metadatendurchsatzes ließ sich mit einer Verteilung der Metadaten-Verwaltung auf alle 16 Knoten des Clusters erreichen. Bei geringer Belastung zeigt diese Konfiguration zwar eine geringere Leistung als die übrigen Konfigurationen des Dateisystems, aber im Gegensatz zu diesen skaliert sie besonders gut mit steigender Belastung und erreicht so einen um 132,7% höheren Durchsatz als die Vergleichskonfiguration mit einen zentralen Metadatenmanager.

Schließlich konnte nachgewiesen werden, dass die Belastung durch Daten-Operationen geringer ist als die durch Metadaten-Operationen. Die Belastung der Daten-Server steigt zwar auch bei höherer Belastung, jedoch wird diese Belastung durch die maximale Leistungsfähigkeit der Metadaten-Verwaltung dominiert und beschränkt.

Kapitel 7

Fazit und Ausblick

Mit dieser Arbeit wurde das Ziel verfolgt, die Leistungsfähigkeit der Metadatenverarbeitung von Cluster-Dateisystemen durch die Verteilung der Metadaten auf mehrere Knoten des Clusters zu verbessern.

Das entwickelte Dateisystem *CLF* folgt der klassischen Architektur für Cluster-Dateisysteme. Nach dieser Architektur kann jeder Knoten im Cluster bezüglich des Dateisystems eine oder mehrere Rollen einnehmen: *Metadatenmanager*, *Daten-Server* und *Klient*. Zusätzlich zu diesen Rollen wird in *CLF* die neue Rolle der *Metadaten-Surrogate* eingeführt. Es ist Aufgabe der Surrogate, anstelle des Metadatenmanagers die Anfragen der Klienten zu beantworten. Die Surrogate ersetzen den Metadatenmanager jedoch nicht vollständig, sondern übernehmen die Verwaltung der Dateisystemobjekte, während der Manager wenige zentrale Aufgaben, wie beispielsweise das Beantworten der initialen Anfragen von Klienten zum Montieren des Dateisystems behält.

Die Metadaten jedes Dateisystem-Objektes (im Fall eines Verzeichnisses auch der dazu gehörige Inhalt) werden auf genau einem der *Metadaten-Surrogate* verwaltet. Die Zuweisung der Metadaten zu einem Surrogat erfolgt mittels einer Hash-Funktion, die eine gleichmäßige Verteilung der Metadaten erreicht und zugleich ein leichtes Auffinden ermöglicht.

Im Gegensatz zu den Daten-Servern können Surrogate jedoch aufgrund der Abhängigkeiten zwischen den verwalteten Dateisystemobjekten nicht unabhängig voneinander operieren. Betrifft eine Anfrage voneinander abhängige Dateisystemobjekte, die von unterschiedlichen Surrogaten verwaltet werden, müssen die Surrogate kooperieren, um die Anfrage bearbeiten zu können. Diese Art der Anfragen wird *potentiell-kooperativ* genannt.

7.1 Ergebnisse

Im Rahmen dieser Arbeit wurde gezeigt, dass es möglich ist, die Verwaltung der Metadaten auf mehrere Knoten zu verteilen und dabei die Funktionalität des Dateisystems zu erhalten. Dazu wurde *CLF* vollständig unter Linux implementiert und getestet. Für die Bearbeitung von *potentiell-kooperativen* Anfragen wurden Kommunikationsmuster entwickelt, die für bestimmte Anfragen eine Kooperation von bis zu vier Surrogaten vorsehen.

Mit Hilfe von Benchmarks konnte gezeigt werden, dass die Standard-Kon-

figuration (ohne Surrogate) von *CLF* einen höheren Durchsatz bietet als vergleichbare Konfiguration von *NFS* und *PVFS2*. Dieser Unterschied kann auf den Aufbau der verwendeten Kern-Module und die systemnahe Programmierung des Metadatenmanagers von *CLF* zurückgeführt werden.

Schließlich wurde untersucht, ob der Einsatz von Surrogaten die Leistungsfähigkeit weiter steigern kann. Dies ist nicht selbstverständlich, da bei Surrogaten die *potentiell-kooperativen* Anfragen eine längere Bearbeitungszeit benötigen, als die gleichen Anfragen an den Metadatenmanager, der diese immer lokal bearbeiten kann.

Auf einem Itanium2-Cluster mit 16 Doppelprozessor-Knoten konnte gezeigt werden, dass der Anfragen-Durchsatz bei Einsatz von Surrogaten im Vergleich zu der Standard-Konfiguration steigt. Bei der Messung wurde das Dateisystem jeweils mit bis zu 32 gleichzeitig zugreifenden Benchmark-Prozessen belastet. Die Zahl der beantworteten Anfragen pro Sekunde mit zwei Surrogaten steigt im Vergleich zu der mit zentralem Metadatenmanager um 26%. Bei einer steigenden Zahl an Benchmark-Prozessen tritt ab einem bestimmten Punkt eine Sättigung der Dienstgeber ein. Mit der Standard-Konfiguration wird dieser Punkt bei 8 Benchmarkprozessen, unter Einsatz von zwei Surrogaten bei 14 Benchmarkprozessen erreicht, wobei beide Konfigurationen auch im Überlastbereich einen stabilen Durchsatz liefern.

Die Leistungssteigerung der Metadatenverwaltung erreicht bei Verwendung von vier Surrogaten 70% im Vergleich zur Standard-Konfiguration, wobei eine Auslastung der vier Surrogaten erst bei $\sim$18 Benchmark-Prozessen erreicht wird.

Die beste Leistung zeigt das Dateisystem, wenn die Surrogate auf dedizierten Knoten des Cluster ablaufen. „Teilen“ sich Klienten und Surrogate einen Knoten, behindern sich die Prozesse gegenseitig, so dass das Dateisystem insgesamt eine schlechtere Metadatenleistung zeigt als mit dedizierten Knoten. Dieser Effekt schwächt sich allerdings bei einer großen Zahl an Surrogaten ab, so dass mit 16 Surrogaten (das bedeutet, dass auf jedem Cluster-Knoten ein Surrogat läuft) auf der Testumgebung die besten Ergebnisse erzielt werden konnten. Mit 16 Surrogaten konnte im Vergleich zur Standardkonfiguration eine Steigerung von 132,7% erreicht werden.

Insgesamt konnte damit gezeigt werden, dass die Verteilung der Metadaten in Cluster-Dateisystemen möglich ist und der Durchsatz der Metadatenverwaltung durch den Einsatz der Metadaten-Surrogaten steigt.

7.2 Ausblick

Die Ergebnisse dieser Arbeit werfen weiterführende Fragen auf, die nicht im Rahmen der Arbeit selber beantwortet werden konnten. Dazu gehören einige Detailfragen, aber auch grundsätzliche Fragen, die in den folgenden Abschnitten angesprochen werden.

7.2.1 Detail: Verbesserungen an *Paradis-Net*

Paradis-Net war sehr hilfreich bei der Implementierung der Metadatenverarbeitung. Einerseits hat die Ereignis-gesteuerte Verarbeitung der Anfragen auf dem *Metadatenmanager* und dem *Metadaten-Surrogat* deren Programmierung

vereinfacht und der Weiterleitungsmechanismus hat insbesondere die Implementierung der *potentiell-kooperativen* Anfragen vereinfacht. Im Kernmodul des *Klienten* haben sich die Stärken des Kooperations-Mechanismus gezeigt, der gleichzeitige Anfragen aus unterschiedlichen Verarbeitungsfäden und mehrere gleichzeitige Anfragen an unterschiedliche *Daten-Server* ermöglicht. Außerdem erlauben es Kooperationen, deklarativ den Speicherbereich vorzugeben, in dem die erwartete Antwort abgespeichert werden soll und verhindern damit das sonst eventuell notwendige Zwischenspeichern der Nachricht in einem zusätzlichen Speicherbereich.

Auf den *Daten-Servern* hat sich jedoch eine Schwäche des Ereignis-gesteuerten Empfangsmechanismus gezeigt: Schreibt ein *Klient* Daten in eine Datei, schickt er diese Daten in einer entsprechenden Anfrage an den *Daten-Server*. Bei den Schreib-Anfragen an die *Daten-Server* handelt es sich um „unerwartete" Daten, die zunächst in einem von *Paradis-Net* angelegten Speicherbereich gespeichert werden. Mit Hilfe des Kooperationsmechanismus wäre es zwar möglich, Daten in einem vorbestimmten Speicherbereich zu empfangen, aber Kooperationen lassen sich nur auf Dienstnehmer-Seite verwenden, wenn die Daten als Antwort auf eine Anfrage eintreffen.

Bei dem Daten-Server muss die Anfrage daher von einer Behandlungsfunktion bearbeitet werden. Die Behandlungsfunktion muss die Daten einmal kopieren, um sie an der richtigen Stelle in der lokalen Datei abzuspeichern. Diese Stelle wird aus den mitgesendeten Meta-Informationen (Datei-Nummer, Offset, Länge) ausgelesen.

Um diese zusätzliche Kopie und die Verwendung zusätzlichen Speichers zu vermeiden, sind zwei Erweiterungen an *Paradis-Net* notwendig: Einerseits muss eine stärkere Parametrisierung der Nachrichten, über den Nachrichtentyp hinaus, möglich sein und andererseits muss es dem Benutzerprogramm erlaubt werden, den Empfangspuffer für eine Nachricht zur Verfügung zu stellen. Mit diesen Erweiterungen könnte die Schreib-Anforderung des *Klienten* zusätzlich mit den Meta-Information der Schreib-Anfrage parametrisiert werden. Diese Parameter würden an eine Benutzer-Routine weiter gegeben, die den richtigen Speicherbereich aus der lokalen Datei in den Hauptspeicher einblenden und einen Verweis darauf an *Paradis-Net* zurückgeben könnte. Dadurch wäre *Paradis-Net* in der Lage, die zu schreibenden Daten über die Speicherabbildung direkt an die richtige Stelle der lokalen Datei zu kopieren.

Die Benutzer-definierten Speicheranforderungsroutinen wurden bereits im Rahmen des *CHIL* Projektes [79] implementiert und werden dort zur Vermeidung von Kopieroperationen eingesetzt. Bei der Anwendung war allerdings die Parametrisierung über einen Nachrichtentyp ausreichend, so dass eine zusätzliche Erweiterung des Nachrichten-Typ-Mechanismus nicht notwendig war. Weil die Daten-Server nicht im Mittelpunkt dieser Arbeit standen, wurde auf die Weiterentwicklung von *Paradis-Net* verzichtet.

7.2.2 Detail: Verbesserung des Aktualisierungsprotokolls

In *CLF* übernehmen Surrogate die Arbeit des Metadatenmanagers und verwenden dazu eine lokale Kopie der Metadaten. In regelmäßigen Abständen schickt jedes Surrogat die Änderungen an den lokalen Metadaten zu dem Metadatenmanager. Dieses Verfahren (siehe Abschnitt 5.4.3.2) ist an zwei Stellen verbesserungswürdig.

Einerseits werden veränderte Verzeichnisse immer als Ganzes an den Manager gesendet, wenn sie verändert wurden. Das erzeugt besonders bei großen Verzeichnissen großen Overhead. Günstiger wäre es, denn die Surrogate auch ein Änderungsliste an den Verzeichnisinhalten führen würden, in der das Hinzufügen und Löschen von Einträgen protokolliert wird. Mit dieser Liste könnten Änderungen viel effizienter zum Metadatenmanager übertragen werden, als mit dem aktuellen Verfahren.

Andererseits führt das regelmäßige Aktualisieren unter Umständen zu unnötigen Aktualisierungen, wenn die selben Metadaten mehrfach über einen längeren Zeitraum geändert werden. Dadurch wird zusätzlicher Netzwerkverkehr erzeugt, weil die selben Inodes mehrfach aktualisiert werden. Günstiger könnte eine bedarfsgesteuerte Aktualisierung sein, bei der Inodes erst dann auf dem Metadatenmanager aktualisiert werden, wenn sie eine gewisse Zeit lang nicht mehr geändert wurden. Allerdings muss gleichzeitig eine Liste der geänderten Inodes und Verzeichnisse geführt werden, die eine gewisse Größe nicht überschreiten sollte, weil mit steigender Größe auch der Platzbedarf und der Verwaltungsaufwand steigen. Es erscheint also sinnvoll, den Zeitpunkt der Aktualisierung einerseits von der seit der letzten Aktualisierung vergangen Zeit und andererseits von der Anzahl der geänderten Metadaten abhängig zu machen.

7.2.3 Grundsätzlich: Streuverteilung vs. gezielte Verteilung

Bei *CLF* wird für die Zuordnung der Metadaten zu den Surrogaten eine Hash-Funktion verwendet, die eine Zuordnung anhand der Inode-Nummer vornimmt. Der Vorteil dieses Vorgehens ist, dass jede Komponente des Dateisystems auf einfach Weise berechnen kann, welches Surrogat eine bestimmte Inode verwaltet, ohne dies zuvor von einem zentralen Dienstgeber erfragen zu müssen.

Obwohl die Streuverteilung bei den Messungen in Kapitel 6 zu guten Ergebnissen geführt hat, könnte dennoch eine gezielte Verteilung der Metadaten wünschenswert sein. Beispielsweise könnte es sich als sinnvoll erweisen, dass die Metadaten von Dateien auf dem gleichen Surrogat verwaltet werden wie des Verzeichnisses in dem sie enthalten sind. Insbesondere dann, wenn die Dateien nur eine geringe Lebensspanne besitzen, wird so die Kooperation von Surrogaten beim Anlegen und Löschen von Dateien vermieden. Auf der anderen Seite kann eine solche Verteilung auch schädlich sein, wenn auf die Dateien eines Verzeichnisses sehr häufig zugegriffen wird, weil dann alle Anfragen an das gleiche Surrogat gestellt werden. In diesem Fall wäre es besser, wenn die Metadaten der Dateien möglichst gleichmäßig über die Surrogate verteilt würden.

Eine gezielte Verteilung der Metadaten könnte also in bestimmten Fällen zu einer besseren Leistung führen. Obwohl dies mit *CLF* aufgrund der festen Abbildung von Inode-Nummer und Surrogat nicht möglich ist, wäre es dennoch denkbar, beim Anlegen von Verzeichnissen und Dateien freie Inode-Nummern gezielt so zuzuweisen, dass ein bestimmtes Surrogat für die Verwaltung der Metadaten zuständig ist. Das setzt aber voraus, dass die optimale Zuordnung schon bei Erzeugung der Datei fest steht, weil eine spätere Umverteilung nicht möglich ist. Inwiefern es möglich ist die Zuordnung im Vorraus zu bestimmen und ob eine gezielte Verteile Vorteile gegenüber der Streuverteilung hat, konnte im Rahmen dieser Arbeit nicht untersucht werden und bedarf weiterer Untersuchungen.

7.2.4 Grundsätzlich: Zusätzliches Cachen

Durch die Surrogate können im Vergleich zum Einsatz eines zentralen Metadatenmanagers mehr Klientenanfragen in der gleichen Zeit beantwortet werden. Ein Großteil dieser Anfragen führt jedoch keine Änderung an den Metadaten durch und könnte durch Cachen der Metadaten auf dem Klienten vermieden und damit besonders schnell „beantwortet“ werden.

Werden die Ergebnisse einer Metadatenanfrage jedoch auf einem Klienten wiederverwendet, kann dies zu einem Konsistenzproblem führen, wenn gleichzeitig ein anderer Klient Änderungen an den Metadaten vornimmt. In diesem Fall müssen die im Cache gespeicherten Metadaten entweder invalidiert oder aktualisiert werden, indem das verantwortliche Surrogat eine entsprechende Nachricht an den Klienten sendet. Das kann auf zwei Arten gemacht werden: Entweder schicken die Surrogate allen Klienten eine Nachricht oder speichern Informationen über den Cacheinhalt der Klienten.

In *CLF* ist ein Mechanismus vorgesehen, dass das Cachen von Metadaten auf den Klienten ermöglicht. Die Grundidee ist es, den Auskünften der Surrogate eine *Gültigkeitsdauer* anzuhängen, die es dem Klienten erlaubt, die Information für die angegebene Zeitspanne wieder zu verwenden. Auf diese Weise erhält das Surrogat Einfluss auf den Cacheinhalt des Klienten und ist in der Lage, bei Änderungen an bestimmten Metadaten, genau die Klienten zu kontaktieren, welche die Informationen mit großer Sicherheit im Cache speichern. Das VII-Modul in *CLF*, welches flüchtige Informationen über Inodes speichert (siehe Abschnitt 5.3.6), besitzt bereits entsprechende Felder, um *Gültigkeitsversprechen*, die Klienten gegeben werden, zu speichern.

Wird die *Gültigkeitsdauer* von Metadaten überschritten und erfolgt ein weiterer Zugriff auf diese, muss der Klient die Metadaten erneut von dem Surrogat anfordern. Auf der anderen Seite kann das Surrogat die Informationen darüber, welche *Gültigkeitsversprechen* welchem Klienten gegeben wurden nach Ablauf der Gültigkeit verwerfen.

Die *Gültigkeitsdauer* von häufig geänderten Metadaten sollte bei „0“ liegen, während die *Gültigkeitsdauer* von selten geänderten Metadaten (wie beispielsweise die von ausführbaren Programmen oder Verzeichnissen, die nahe an der Wurzel des Dateisystems liegen) im Bereich von Sekunden und Minuten liegen sollte.

Der beschriebene Mechanismus wurde in *CLF* bereits teilweise implementiert, konnte aber aus Zeitgründen im Rahmen dieser Arbeit nicht zu Ende entwickelt werden. Es ergeben sich interessante Fragestellungen: Bezüglich der Wahl der *Gültigkeitsdauer* ist offen, auf Basis welcher Heuristik diese gewählt werden sollte. Hier könnte beispielsweise die seit der letzten Änderung an den Metadaten vergangene Zeit eine gute Heuristik bilden. Bezüglich der Buchführung auf den Surrogaten ist offen, welche zusätzlichen Kosten (Speicherplatz und Rechenzeit) die Verwaltung der *Gültigkeitsversprechen* verursachen würde. Die zweite Fragestellung ist mit der ersten in so fern verknüpft, dass die Anzahl der zu verwaltenden *Versprechen* von der verwendeten Heuristik abhängt.

Diese Erweiterung von *CLF* erscheint besonders vielversprechend, ihr wirklicher Nutzen kann jedoch nur mit einer vollständigen Implementierung evaluiert werden.

7.2.5 Grundsätzlich: Ausfallsicherheit

In *CLF* wurde Ausfallsicherheit beinahe vollständig außer Acht gelassen. Fällt eine der Infrastruktur-Komponenten aus, ist das Dateisystem nicht mehr funktionsfähig. Das gilt sowohl für die Daten-Server, wie auch für den Metadatenmanager und die Surrogate. Fehlt ein Daten-Server, sind Metadaten-Operationen zwar noch möglich, aber das Lesen und Schreiben von Dateien wird mit hoher Wahrscheinlichkeit fehlschlagen. Bei Ausfall eines Surrogats funktionieren alle Operationen, die Metadaten betreffen für die das Surrogat verantwortlich ist, nicht mehr.

Der Ausfall des Metadatenmanagers kann unterschiedliche Auswirkungen haben: Verwendet die Konfiguration von *CLF* keine Surrogate, wird das Dateisystem augenblicklich unbrauchbar, bei der Verwendung von Surrogaten, können diese den Ausfall zeitweise kompensieren, weil sie alle Anfragen der Klienten bearbeiten. Trotzdem wäre es nach einer gewissen Zeit nicht mehr möglich, Dateien und Verzeichnisse anzulegen, weil der Manager für die Verwaltung der Inode-Nummern zuständig ist.

Es ist eine offene Frage, ob mögliche Ausfälle durch Replizierung der Komponenten aufgefangen werden könnte. In diesem Zusammenhang muss auch untersucht werden, ob die Metadaten bei Ausfall eines Surrogats weiterhin konsistent sind. Unter Umständen muss nach Ausfall eines Surrogats eine Untersuchung des gesamten Dateisystems durchgeführt werden um die Konsistenz des Dateisystems zu sichern.

Anhang A

Daten-Server

Die Daten-Server in *CLF* speichern die in dem Dateisystem abgelegten Daten. Sie sind als Benutzerebenenprozesse implementiert und benutzen das lokale Dateisystem des Knotens zur persistenten Speicherung. Sie können als Blockgeräte verstanden werden, denn sie besitzen nur wenige Informationen über die auf ihnen gespeicherten Datenblöcke.

Wie die anderen Komponenten des Dateisystems verwenden die Daten-Server zur Kommunikation die Bibliothek *Paradis-Net*, wobei sie als einzige Komponenten von *CLF* eine reine Dienstgeber-Rolle einnehmen (siehe Abbildung 5.1). Die Daten-Server werden hauptsächlich von den Klienten des Dateisystems kontaktiert, wenn diese aus Dateien lesen oder in diese schreiben. Weiterhin werden sie auch von dem Metadatenmanager des Dateisystems kontaktiert, wenn Dateien gelöscht werden.

Diese Arbeit konzentriert sich auf die Verwaltung der Metadaten, daher wird die Beschreibung der Daten-Server sich auf deren nach außen sichtbare Funktionalität beschränken. Die Daten-Server sind bis auf wenige nachträgliche Protokolländerungen mit denen identisch, die im Rahmen der früheren Arbeiten an *Clusterfile* entwickelt wurden [44].

A.1 Konfigurationsmodus

Die Konfigurierung der Daten-Server erfolgt bei Initialisierung des Dateisystems. Der Daten-Server ist nach seinem Start zunächst im Konfigurationsmodus und wartet auf eine Nachricht des Metadatenmanagers mit den Konfigurationsinformationen. Diese bestehen aus folgenden Daten:

nr_ios, ios_idx Anzahl der Daten-Server und eigener Index des Daten-Servers. Diese Informationen benötigt der Server um seine eigenen Zuständigkeiten zu berechnen. So wird in einer Leseanfrage beispielsweise der gesamte zu lesende Bereich angegeben und der Daten-Server berechnet daraus die Datenblöcke, die der Klient von ihm erwartet.

block_size Die innerhalb des Dateisystems benutzte Blockgröße. (Bei *CLF* üblicherweise 64 KB.)

ios_path Verzeichnispfad. Der Daten-Server speichert seine Daten in dem lokalen Dateisystem des Knotens. Die Option verweist auf das dafür verwendete Verzeichnis.

ios_logpath Pfad der Log-Datei. Auch der Name der Log-Datei wird von dem Metadatenmanager bestimmt.

Nach Erhalt dieser Nachricht wechselt der Daten-Server in den Dienstmodus und akzeptiert jetzt auch die Anfragen der anderen Dateisystemkomponenten.

A.2 Dienstmodus

Im Dienstmodus akzeptiert der Daten-Server die fünf in Tabelle A.1 aufgezählten Anfragetypen. Neben diesen Anfragetypen können die Daten-Server auch noch weitere Anfragen verarbeiten, die aber nur in Zusammenhang mit der Benutzerebenenbibliothek von *Clusterfile* verwendet werden. Sie dienen der Unterstützung von unregelmäßigen Datenverteilungs-Anforderungen von parallelen Programmen, die mit Hilfe von so genannten *PITFALLS* (siehe [47]) formuliert werden. Diese erweiterten Möglichkeiten zur physikalischen Verteilung von Dateien auf Daten-Server werden von Cluster-Dateisystemen nicht benötigt, da sie von den Anwendungen Kenntnis über die Struktur des Dateisystems voraussetzen. Im Gegensatz dazu ist es aber das Ziel von *CLF*, dass die Beschaffenheit des Dateisystems transparent und nicht von einem lokalen Dateisystem unterscheidbar ist.

Die Nachrichten vom Typ `CLF_RQ_IOS_FSYNC`, `CLF_RQ_IOS_WRITE` und `CLF_RQ_IOS_READ` werden von Klienten an die Daten-Server geschickt. Die Daten-Server bearbeiten die Anfragen sequentiell in der Reihenfolge in der sie eintreffen. Schreiben dabei zwei unterschiedliche Klienten in die selbe Datei, so überschreibt die zweite Schreibanforderung unter Umständen die von der ersten Anforderung in die Datei geschrieben Daten. Dieses Problem existiert allerdings nicht nur bei Netzwerkdateisystemen, sondern auch bei lokalen Dateisystemen, wenn zwei Prozesse auf die selbe Datei zugreifen.

Die Daten-Server besitzen im Gegensatz zum Metadatenmanager keine Informationen über die auf den Klienten geöffneten Dateien. Sie benutzen allerdings zum Speichern und Lesen der Daten im lokalen Dateisystem die üblichen Dateioperationen der Glibc-Bibliothek [29] und müssen daher zuvor die lokalen Dateien, in denen die Daten abgelegt werden, öffnen und nach Gebrauch wieder schließen. Dazu verwenden sei einen LRU-Puffer, der immer eine gewisse Menge an Dateien offen hält, um wiederholtes Öffnen und Schließen von einzelnen Dateien zu verhindern. Dies geschieht unter der Annahme der zeitlichen Lokalität von Zugriffen auf eine bestimmte Datei.

Anfragen vom Typ `CLF_RQ_IOS_FTRUNCATE` werden sowohl von Klienten, wie auch von dem Metadatenmanager an die Daten-Server verschickt. Stammt die Nachricht von einem Klienten, ist sie das Resultat eines expliziten `ftrunc()`-Dateisystemaufrufs eines Anwendungsprogramms. Stammt diese Nachricht allerdings von dem Metadatenmanager, so handelt es sich um eine Lösch-Operation, die zwar die lokale Datei nicht entfernt, aber durch das Zurücksetzen der Länge auf 0 alle in der Daten gespeicherten Daten löscht. Die Ursache dieser Anfrage ist üblicherweise eine Lösch-Anfrage von einem der Klienten (Nachricht vom Typ `CLF_RQ_UNLINK`).

Konfigurationsmodus	
CLF_RQ_IOS_INIT	Der Metadatenmanager schickt eine Anfrage dieses Typs an den Daten-Server und versetzt damit in den *Dienstmodus*. Parameter: Konfiguration des Daten-Servers (siehe A.1)
Dienstmodus	
CLF_RQ_IOS_FSYNC	Synchronisiert den Inhalt einer Datei mit der Festplatte, bzw. gibt dem lokalen Dateisystem diesen Befehl. Parameter: Dateiinformation
CLF_RQ_IOS_FTRUNCATE	Verkürzt die Länge einer Datei auf einen bestimmten Wert. Mit dieser Anfrage ist es auch möglich Dateien zu löschen. Parameter: Dateiinformationen, neue Dateilänge
CLF_RQ_IOS_WRITE	Mit dieser Anfrage werden Daten in eine Datei geschrieben. Parameter: Dateiinformation, Versatz innerhalb der Datei, Länge der Daten, Daten
CLF_RQ_IOS_READ	Mit dieser Anfrage werden Daten aus einer Datei angefordert. Parameter: Dateiinformation, Versatz innerhalb der Datei, Länge der Daten

Tabelle A.1: Anfragen an den Daten-Server

Der Parameter „Dateiinformation“ wird in Abschnitt A.3 beschrieben.

Es gibt zwei Gründe für diesen Unterschied: Der erste Grund ist, dass eine Lösch-Operation nicht nur den Inhalt der Datei betrifft, sondern auch ihre Metadaten und ihren Verzeichniseintrag. Der zweite Grund ist, dass der Dateiinhalt erst dann gelöscht werden darf, wenn kein Klient diese Datei geöffnet hält. Der Metadatenmanager tritt daher als Mittler zwischen Klient und Daten-Server ein und schickt den Befehl zum Löschen des Dateiinhaltes erst dann an die Daten-Server, wenn die Bedingung erfüllt ist. Der Verzeichniseintrag der Datei wird allerdings sofort entfernt.

A.3 Dateiinformationen

Die Daten-Server besitzen, abgesehen von der Inode-Nummer der Dateien, keine Informationen über die Daten, die auf ihnen gespeichert werden. Sie kennen aber die Gesamtzahl der Daten-Server, sowie ihren eigenen Index innerhalb dieser Gruppe. Aus diesen Informationen wird zusammen mit dem Index des Daten-Servers, auf dem der erste Block der Datei gespeichert wird, und der verwendeten Blockgröße der Teil der Datei berechnet, für die der Daten-Server verantwortlich ist.

Daher müssen die anderen Komponenten jeder Anfrage an einen Daten-Server (siehe Tabelle A.1) die Informationen über die Verteilung der Datei anhängen. Der Vorteil dieser Vorgehensweise liegt in größerer Flexibilität. Der Metadatenmanager kann bei der Erstellung einer Datei festlegen, wie diese auf den Daten-Servern verteilt wird ohne einem bestimmten, abgesprochenen Schema folgen zu müssen. Diese Information wird auf dem Metadatenmanager gespeichert und an die Klienten weitergegeben. Die Klienten benötigen die Informationen über die Verteilung der Daten um daraus errechnen zu können, welche Daten-Server für einen bestimmten Ausschnitt der Datei zuständig sind. In den Anfragen werden die Dateiinformationen an den Daten-Server weitergereicht, der damit die Daten in seiner unstrukturierten Datei auffinden kann.

A.4 Zusammenfassung

In diesem Abschnitt wurde die Grundfunktionalität der Daten-Server und die zur Kommunikation mit Metadatenmanager und Klient verwendeten Anfragetypen vorgestellt.

Anhang B

Metabench

Im Rahmen der Recherchen konnte nur ein Benchmark gefunden werden, der die Metadaten-Leistung eines Dateisystems misst: *Metabench* [80]. Leider besitzt *Metabench* eine Reihe von Mängeln, die zu der Entwicklung von *Medabench* geführt haben, welcher in Kapitel 6 für die Vermessung von *CLF* und dessen Vergleich mit anderen Dateisystemen verwendet wird.

In diesem Anhang wird zunächst *Metabench* beschrieben und im Anschluss werden einige Messungen mit *CLF* und *PVFS* vorgestellt und diskutiert.

B.1 Beschreibung

Metabench wurde im Rahmen einer Studienarbeit an der Technischen Universität München entwickelt. Das Ziel der Entwicklung war die Vermessung verschiedener Dateisystem (darunter PVFS [17], AFS [41] und NFS [70]) auf dem Linux-Cluster des Leibniz-Rechenzentrums in München. Eine vereinfachte Version ist Teil einer Benchmark-Sammlung, die bei der Evaluierung eines neuen Supercomputers („Höchstleistungsrechner in Bayern II“, Installierung: Ende 2005) eingesetzt wurde.

Der Benchmark zählt zu den *Szenariobasierten Benchmarks*, welche die Leistung eines Systems anhand eines bestimmten Einsatzszenarios messen. *Metabench* ist eine Erweiterung des bekannten *Postmark*-Benchmarks [48], die es durch den Einsatz von MPI [33] erlaubt, den Benchmark auf beliebig vielen Knoten eines Clusters auszuführen.

Der ursprüngliche *Postmark*-Benchmark wurde 1997 entwickelt und bildet die Dateisystemzugriffscharakteristik eines *News- oder Email-Servers* nach. Charakteristisch für diese Server ist die Verwaltung einer großen Zahl relativ kleiner Dateien auf die lesend und schreibend zugegriffen wird. Auch das Erstellen und Löschen von Dateien ist Teil der Messung, wobei der Anteil der vier Operationen an der gesamten Anfragemenge konfiguriert werden kann.

In *Metabench* wird eine feste Menge von Dateien verwendet, auf der die Operationen ausgeführt werden, allerdings wurde die Leistung der Metadatenverwaltung dadurch in den Vordergrund gerückt, dass nur sehr kleine Datenmengen gelesen und geschrieben werden. Die Dateien werden, je nach Konfiguration, in einem gemeinsamen Verzeichnis für alle Prozesse oder in Prozess-individuellen Verzeichnissen abgelegt. Die Messung verläuft, wie bei *Postmark*, in drei Phasen:

Phase 1: In der ersten Phase wird gemessen, wieviele Dateien pro Sekunde erzeugt werden können. Diese Phase dient gleichzeitig dem Aufbau der Dateimenge. Die Anzahl der Dateien die erzeugt werden (sowie die Zahl der durchgeführten Operationen in Phase 2) werden durch einen kurzen Vor-Test ermittelt. Messung: Erzeugte Dateien pro Sekunde.

Phase 2: In der zweiten Phase werden auf den in Phase 1 erstellten Dateien zufällige Operationen ausgeführt. Die Operationen setzen sich zu gleichen Teilen aus Lese-, Schreibe-, Datei-Erzeugungs- und Datei-Löschungs-Operationen zusammen. Messung: Operationen pro Sekunde.

Phase 3: In der dritten Phase wird gemessen, wie lange das Entfernen aller Testdaten dauert, woraus berechnet wird, wie viele Lösch-Operationen das Dateisystem pro Sekunde durchführen kann. Messung: Gelöschte Dateien pro Sekunde.

Metabench führt die drei Phasen wiederholt mit einer steigenden Zahl von Prozessen aus, bis die zur Verfügung stehende MPI-Obergrenze erreicht ist. Das Benchmarkprogramm ist in der Programmiersprache C implementiert. Die von diesem Programm generierten Textdateien werden in einem zweistufigen Prozess mit Hilfe von Perl-Skripten ausgewertet und in generierten HTML-Seiten zusammengefasst. Die Seiten enthalten Graphen, die das Ergebnis der einzelnen Messungen mit unterschiedlicher Knotenzahl, sowie drei zusammenfassende Graphiken der drei Phasen, welche die Leistung des Dateisystems jeweils in Abhängigkeit von der Knotenzahl zeigen.

B.1.0.1 *CLF*

Metabench führt in den Phasen 1 und 3 ausschließlich Operationen aus, die zu *potentiell-kooperativen* Anfragen bei der Verwendung von Metadaten-Surrogaten führen. Auch während der Phase 2 sind 50% der Operationen von diesem Typ. Diese Zusammensetzung der Operationen wird, beinahe zwangsläufig, zu einem schlechten Abschneiden von *CLF* bei Verwendung von Surrogaten führen, weil *CLF* so optimiert wurde, dass *einfache* Operationen schneller und *potentiell-kooperative* Operationen *potentiell* langsamer ausgeführt werden. Dieser Optimierung liegt die Annahme zugrunde (siehe [27]), dass der Anteil der *einfachen* Operationen bei über 95% liegt.

B.1.1 Kritik

Leider ist die vorliegende Version 0.4.3 von *Metabench* in einem schlechten Zustand. Zum einen werden einige in der Dokumentation beschriebene Auswertungen nicht durchgeführt und zum anderen muss zur Veränderung einiger Parameter der Quelltext verändert werden.

Vor dem eigentlichen Benchmark wird ein Vortest durchgeführt, der die Leistung des Dateisystems abschätzen und die Anfragemenge entsprechend anpassen soll. Dies funktioniert jedoch nur unzureichend, so dass die einzelnen Testläufe teilweise sehr kurz sind. Die folge sind große Schwankungen in den Ergebnissen.

Ein grundsätzlicher Kritikpunkt ist das verwendete Szenario: Es ist unklar, ob das Szenario eines *News- und EMail-Server* auf andere Anwendungsbereiche übertragbar ist. Auch ist die Verzeichnisstruktur, wenn überhaupt, dann nur für

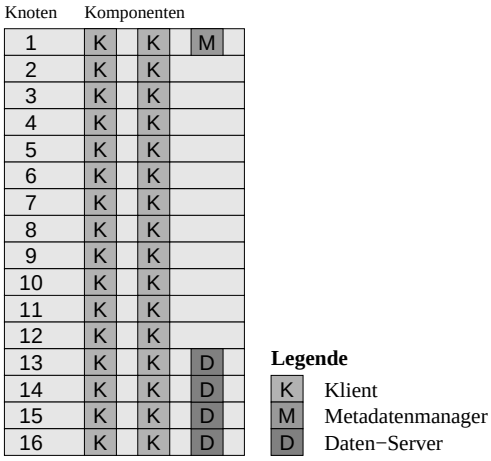


Abbildung B.1: Die im Anhang verwendete Standard-Konfiguration mit einem zentralen Metadatenmanager

diese spezielle Anwendung typisch. Weiterhin ist die Auswahl der untersuchten Operationen (Lesen, Schreiben, Erzeugen und Löschen von Dateien), sowie deren Verhältnis zueinander eine beliebige Festlegung des Programmierers. Die Tatsache, dass alle beteiligten Prozesse auf Dateien zugreifen, die entweder in einem gemeinsamen Verzeichnis oder in individuellen Unterverzeichnissen abgelegt werden, scheint den pragmatischen Überlegungen des Programmierers zu entstammen, aber nicht durch reale Anwendungen untermauert zu sein.

Der Benchmark misst also nicht die Metadatenleistung eines Cluster-Dateisystems, sondern vor allem die Leistung der verwendeten vier Operationen, wodurch vor allem die Auswertung der Phasen 1 und 3, die nur Erzeugungs- und Lösch-Operationen vermessenen, eine gewisse Aussagekraft besitzen.

B.2 Vergleich von *PVFS2* und *CLF*

In diesem Abschnitt wird *CLF* anhand von *Metabench* mit PVFS2 [17] (Version 1.5.1) verglichen. Bei *CLF*wird eine Konfiguration mit einem zentralen Metadatenmanager verwendet, die dem Layout einer PVFS2-Installation entspricht.

Abbildung B.1 (Seite 165) zeigt die Verteilung der Komponenten auf die Knoten. Der Metadatenmanager kommt auf Knoten 1 zur Ausführung, die Daten-Server werden auf den Knoten 13, 14, 15 und 16 platziert. Alle Knoten, einschließlich der Knoten auf denen Dienstgeber laufen, werden für jeweils zwei Klienten verwendet. Bei Messungen mit weniger als 32 Klienten werden zuerst die Klienten auf Knoten mit geringerer Ordnungsnummer verwendet, während die übrigen Klienten nicht verwendet werden.

Abbildung B.2 zeigt die Leistung der beiden Dateisysteme in Phase eins. Es ist deutlich zu sehen, dass die Leistung von *CLF* deutlich höher als die von PVFS ist. *CLF* (`CLF_Manager`) erreicht mit 1032.51 Operationen pro Sekunde seine maximalen Durchsatz bei 6 Klienten, während PVFS seine maximale Leistung

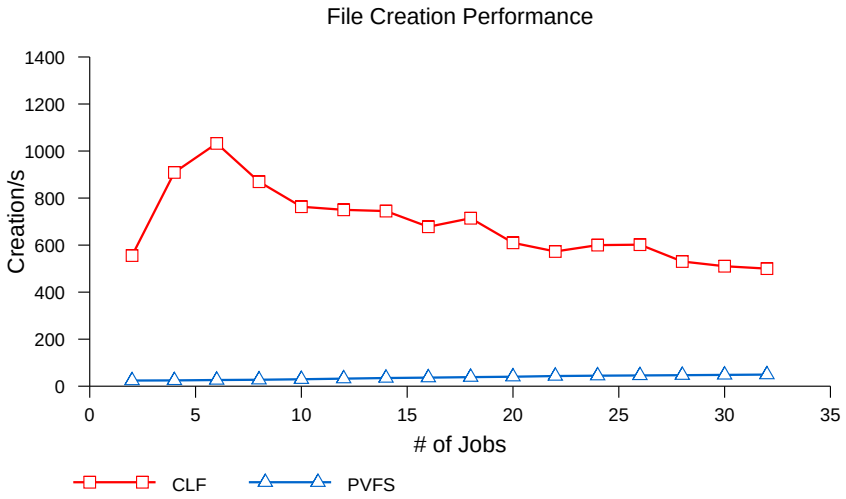


Abbildung B.2: Metabench, Phase 1: Erstellen von Dateien
Konfiguration: Abb. B.1

mit 49,36 Operationen pro Sekunde bei 32 Klienten erreicht. Die maximale Datei-Erzeugungs-Rate von *CLF* liegt damit ca. 20 mal höher als die von PVFS.

In Phase zwei (Abbildung B.3) zeigt sich ein ähnliches Bild: Auch hier ist der Durchsatz von *CLF* um ein Vielfaches höher als der von PVFS. *CLF* erreicht seine maximale Leistung bei 6 Knoten mit 1150,12 Transaktionen pro Sekunde, während bei 32 Knoten ein Maximum von 68,91 Transaktionen pro Sekunde erreicht (Faktor 16 Unterschied). Beide Dateisysteme können in Phase zwei mehr Operationen pro Sekunde bearbeiten als in Phase eins. Der Grund dafür liegt in der Zusammensetzung der Operationen: Die Manipulation von Verzeichnissen ist offenbar viel aufwändiger als die Manipulation von Metadaten, die beim Lesen und Schreiben von Daten vorgenommen wird.

Auch in Phase drei erreicht *CLF* bei 6 Klienten seine maximale Auslastung (3766,50 Lösch-Operationen pro Sekunde), während PVFS bei 32 Klienten sein Maximum (36,83 Lösch-Operationen pro Sekunde) erreicht. In dieser Phase liegt sogar ein Faktor von 102 zwischen den Leistungen der Dateisysteme.

Auffallend an Phase drei ist die im Vergleich zu Phase eins deutlich höhere Verarbeitungsgeschwindigkeit bei *CLF*. Dieser Unterschied ist vermutlich auf das in Abschnitt 5.3.5.7 beobachtete Verhalten des Modules zur Verzeichnisverwaltung zurück zu führen. Dort wurde gezeigt, dass das Entfernen eines Verzeichniseintrags aufgrund der Implementierung um Faktor 3 schneller ist als das Hinzufügen von Einträgen.

Insgesamt ist die Leistung von PVFS überraschend schlecht und das nicht nur im Vergleich mit *CLF*. Den höchsten Durchsatz erreicht PVFS in Phase zwei mit nur 68,91 Transaktionen pro Sekunde; in allen drei Phasen strebt die Leistung von *CLF* bei steigender Klientenzahl auf einen Wert von ca. 600 Operationen pro Sekunde zu. Es ist leider nicht nachzuvollziehen, warum der Durchsatz bei *CLF* nur bei niedrigen Klientenzahlen so groß ist und bei mehr Klienten

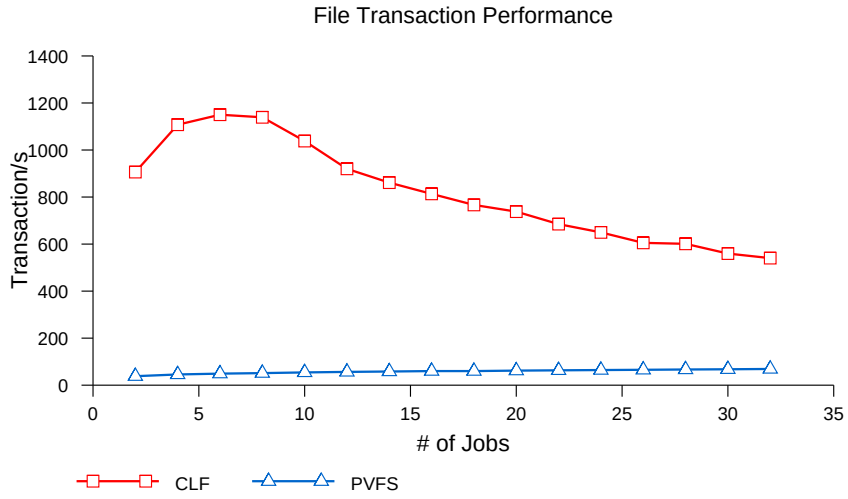


Abbildung B.3: Metabench, Phase 2: Transaktionen (Erstellen, Löschen, Lesen und Schreiben von Dateien)
Konfiguration: Abb. B.1

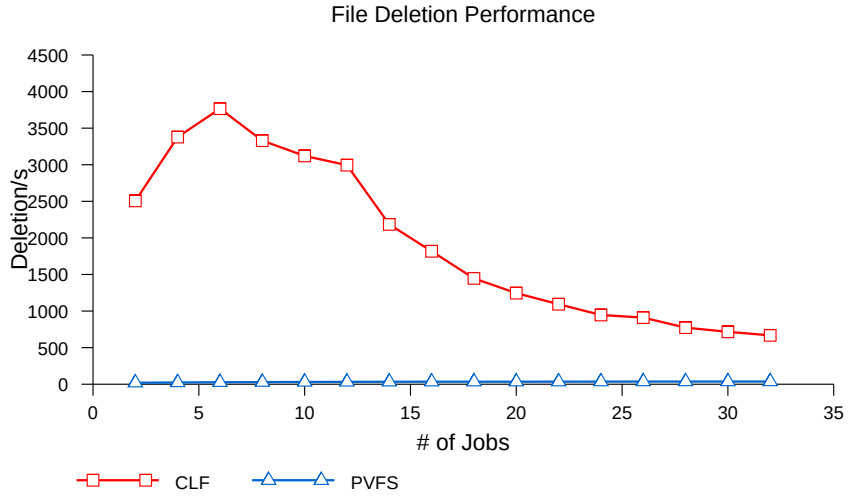


Abbildung B.4: Metabench, Phase 3: Löschen von Dateien
Konfiguration: Abb. B.1

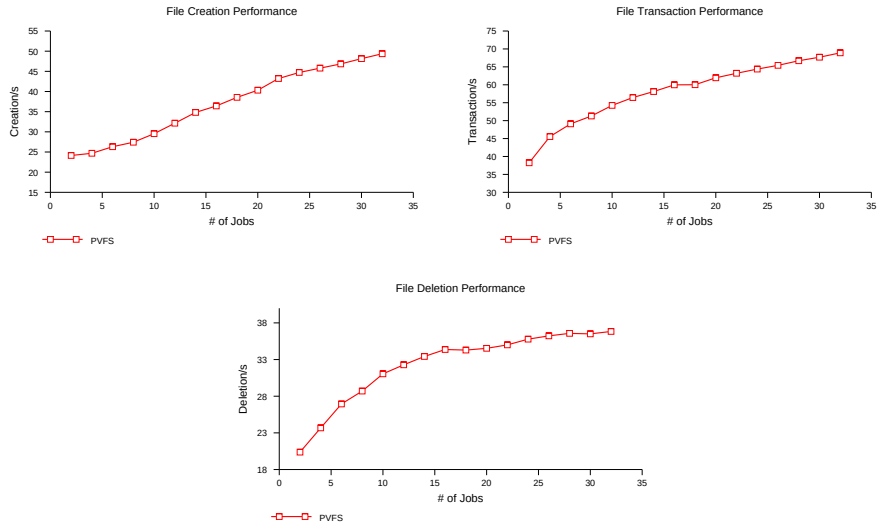


Abbildung B.5: Metabench, Phasen 1–3: PVFS
Konfiguration: Abb. B.1

nachlässt, weil der Benchmark keine detaillierte Auswertung der einzelnen Klienten zulässt.

B.2.1 Deutung

Nach den Vergleichsmessungen mit beiden Benchmarks stellt sich die Frage, warum PVFS2 eine um mindestens eine Größenordnung schlechtere Leistung als *CLF* zeigt? Es handelt sich dabei vermutlich um eine Kombination aus zwei Ursachen, die bei zwei der involvierten Komponenten zu finden sind: Metadatenmanager und Klient.

Bei Betrachtung der Messergebnisse von *Metabench* fällt auf, dass die Leistung von PVFS in allen drei Phasen, besonders aber in Phase 1, bei der Erzeugung von Dateien, stetig und beinahe linear ansteigt. Abbildung B.5 zeigt nochmals die Ergebnisse der Messungen bei voller Ausnutzung der y-Achse. Dieses stetige Wachstum weist darauf hin, dass der Metadatenmanager von PVFS zu keinem Zeitpunkt ausgelastet ist und dass statt dessen ein Problem auf den Klienten vorliegt, weil jeder zusätzliche Klient die Leistung um (näherungsweise) den gleichen Betrag erhöht.

Die abschließende Deutung des Verhaltens von *PVFS2* ist in Abschnitt 6.5.3 zu finden.

B.3 Zwei *CLF* Surrogate

In diesem Abschnitt wird das Verhalten einer *CLF*-Konfiguration, die Surrogate verwendet mit der Standardkonfiguration (Abbildung B.1) verglichen. Die Surrogate werden auf den Knoten 11 und 12 gestartet und senden bei Bedarf

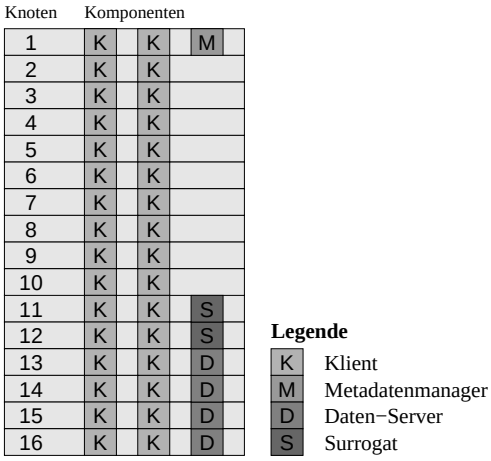


Abbildung B.6: Die Konfiguration bei Einsatz von zwei Surrogaten

im Abstand von einer Sekunde Aktualisierungsnachrichten an den Metadatenmanager. Die Konfiguration ist in Abbildung B.6 dargestellt.

Abbildung B.7 zeigt den Vergleich der beiden Konfigurationen in der ersten Phase. Der Verlauf der Kurven ist vergleichbar, wobei die Leistung der Konfiguration mit zwei Surrogaten (`CLF_2_Surrogate`) unter der Leistung der Konfiguration mit dem zentralen Metadatenmanager `CLF_Manager` bleibt. Eine Ausnahme bildet der Bereich zwischen 8 und 16 Klienten, in dem Konfiguration A eine ähnliche Leistung zeigt.

Von den drei Phasen der Messung, schneidet `CLF_2_Surrogate` nur in dieser Phase deutlich schlechter ab. Die Gründe dafür liegen vermutlich in dem größeren Aufwand der Operation zum Hinzufügen von Verzeichniseinträgen, die auf beiden Surrogaten durchgeführt wird. Weil aufgrund der Zahl an Surrogaten, durchschnittlich 50% dieser Operationen beide Surrogate betreffen, muss eine Anfrage in diesem Fall von beiden bearbeitet werden. Dadurch werden eventuell auch Einfüge-Operationen verzögert, die von einem Surrogat alleine bearbeitet werden.

Abbildung B.8 zeigt das Ergebnis der Messungen in Phase zwei. Der Verlauf der Kurven läuft in dieser Phase weitgehend parallel, wobei die Leistung von `CLF_2_Surrogate` nur knapp unter der von Konfiguration `CLF_Manager` bleibt. Dies ist vermutlich auf den geringeren Anteil der potentiell-kooperativen Anfragen zurück zu führen. Die Lese- und schreibe-Anfragen werden von beiden Surrogaten parallel abgearbeitet, wodurch der größere Aufwand für das Erstellen und Löschen von Dateien teilweise wieder ausgeglichen wird.

Auch in Phase drei (Abbildung B.9) ist die Konfiguration `CLF_2_Surrogate` der Konfiguration `CLF_Manager` unterlegen. Trotz der ausschließlich aus Lösch-Anfragen bestehenden Anfragen ist der Unterschied jedoch nicht so groß wie in der ersten Phase. Der Grund dafür liegt in der kürzeren Bearbeitungszeit von Lösch-Anfragen (siehe auch Abschnitt 5.3.5.7).

Wie bereits in Abschnitt B.1.0.1 beschrieben, ist es nicht überraschend, dass die Verwendung von Surrogaten bei den Messungen mit *Metabench* keine Vorteile gegenüber der klassischen Konfiguration zeigt. Aufgrund der von *Metabench*

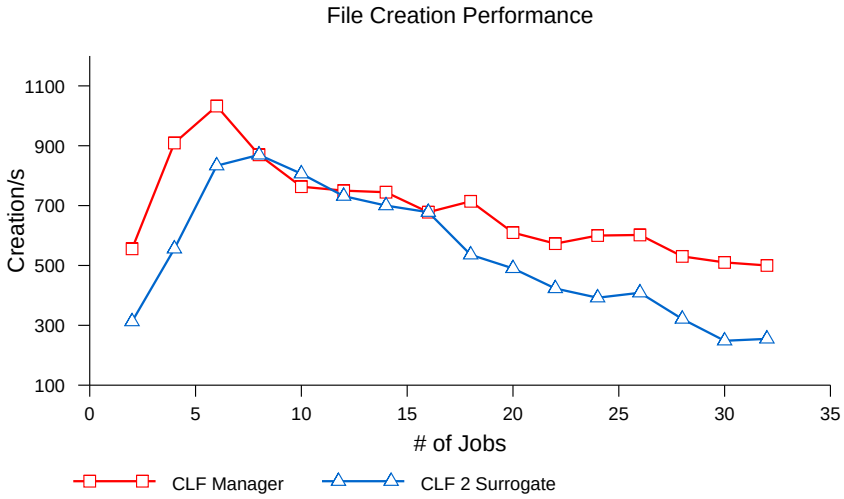


Abbildung B.7: Metabench, Phase 1: Erstellen von Dateien
Konfigurationen: *CLF Manager*: Abb. B.1, *CLF 2 Surrogate*: Abb. B.6

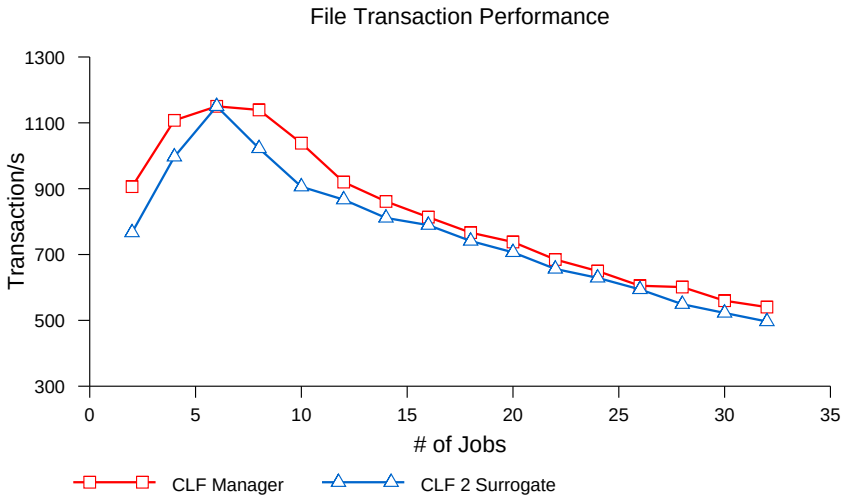


Abbildung B.8: Metabench, Phase 2: Transaktionen (Erstellen, Löschen, Lesen und Schreiben von Dateien)
Konfigurationen: *CLF Manager*: Abb. B.1, *CLF 2 Surrogate*: Abb. B.6

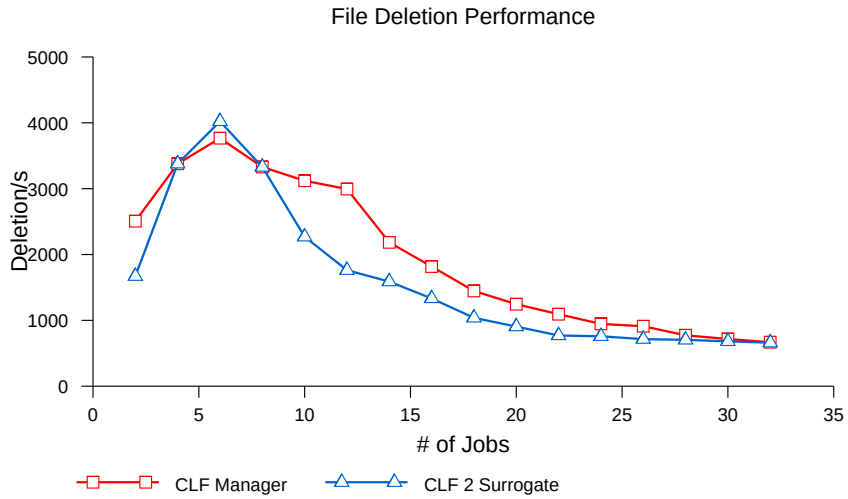


Abbildung B.9: Metabench, Phase 3: Löschen von Dateien
Konfigurationen: *CLF Manager*: Abb. B.1, *CLF 2 Surrogate*: Abb. B.6

durchgeführten Operationen ist die schlechtere Leistung sogar erwartungsgemäß, weil das Erstellen und das Löschen von Dateien einen erhöhten Kommunikationsaufwand zur Folge hat und die Antwortzeiten durch das Weiterleiten der Anforderung von einem Surrogat auf den anderen steigen.

Trotzdem ist das Ergebnis des Benchmarks als positiv zu bewerten, weil sich die Verwendung der Surrogate selbst bei dieser höchst ungünstigen Zusammensetzung von Operationen nicht sehr negativ auswirkt. In der Transaktions-Phase, in der neben Erstellung und Löschung von Dateien auch Zugriffe auf diese erfolgen, ist der Unterschied kaum noch bemerkbar.

Literaturverzeichnis

[1] Berkeley-Traces. `http://tracehost.cs.berkeley.edu/traces.html`, 1996.

[2] Thomas E. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb 1996.

[3] Maurice J. Bach. *The design of the UNIX operating system.* Prentice-Hall, Inc., 1986.

[4] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *SIGOPS Oper. Syst. Rev.*, 25(5):198–212, 1991.

[5] Steve Best. JFS Log: How the Journaled File System Performs Logging. In *Proceedings of the USENIX ALS 2000*, October 2000.

[6] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, February 1995.

[7] Daniel P. Bovet and Marco Cesati. *Unterstanding the Linux Kernel.* O'Reilly Media Inc., 3rd edition edition, November 2005.

[8] Peter J. Braam. Lustre: A Scalable, High-Performance File System. Whitepaper, Cluster File Systems, `www.clusterfs.org`, November 2002.

[9] Peter J. Braam and Rumi Zahir. Lustre Technical Project Summary. Technical report, Cluster File Systems, `www.clusterfs.org`, July 2001.

[10] Scott A. Brandt, Ethan L. Miller, Darell D. E. Long, and Lan Xue. Efficient Metadata Management in Large Distributed Storage Systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 290–298, April 2003.

[11] Scott A. Brandt, Ethan L. Miller, Sage A. Weil, and Kristal T. Pollack. Dynamic Metadata Management for Petabyte-scale File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage (SC2004)*, November 2004.

[12] David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley professional computing series. Addison-Wesley, 2004.

[13] Brent Callaghan, Brian Pawlowski, and Peter Staubach. RFC 1813: NFS Version 3 Protocol Specification, June 1995.

[14] Richard Campbell. *Managing AFS: The Andrew File System.* Prentice Hall, ISBN 0-13-802729-3, 1998.

[15] Pei Cao, Edward W. Felten, and Kai Li. Implementation and Performance of Application-Controlled File Caching. *Operating Systems Design and Implementation*, pages 165–177, 1994.

[16] R. Card, T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. *Proceedings of the First Dutch International Symposium on Linux*, pages 90–367, 1994.

[17] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta*, pages 317–327, October 2000.

[18] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 1–14, Berkeley, CA, USA, 1999. USENIX Association.

[19] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[20] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[21] IEEE Portable Applications Standards Comittee. IEEE std 1003.1-2001 (posix), December 2001.

[22] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, 1996.

[23] Microsoft Corporation. FAT: General Overview of On-Disk Format. Technical report, December 2000.

[24] SGI Corporation. Xfs: A high-performance journaling file system. `http://oss.sgi.com/projects/xfs/`.

[25] Toni Cortes, Sergi Girona, and Jesús Labarta. PACA: A Cooperative File System Cache for Parallel Machines. *2nd International Euro-Par Conference*, pages 477–486, 1996.

[26] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *First Symposium on Operating Systems Design and Implementation*, November 1994.

[27] Oliver Denninger. Entwurf und Entwicklung eines Benchmarks für Clusterdateisysteme. Master's thesis, Fakultät für Informatik, Universität Karlsruhe, September 2006.

[28] Jon Postel (ed.). RFC793: Transmission Control Protocol, 1981.

[29] Sandra Loosemore et al. *GNU C Library: Application Fundamentals.* The Free Software Foundation (FSF), March 2004.

[30] Sandra Loosemore et al. *GNU C Library: System & Network Applications.* The Free Software Foundation (FSF), March 2004.

[31] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing – a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, September 1979.

[32] Markus Fischer. Sockets-GM: Mapping Distributed Applications to Myrinet. Talk at MUG-2002, 2002.

[33] The MPI Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994.

[34] The MPI Forum. *MPI-2: Extensions to the Message-Passing Interface.* University of Tennessee, November 1997.

[35] Ian Foster. What is the Grid? A Three Point Checklist. *Grid Today*, 1(6), 2002.

[36] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *19th ACM Symposium on Operating Systems Principles*, Lake George, NY, October 2003.

[37] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-112, School of Computer Science, Carnegie Mellon University, March 1997.

[38] Richard Gooch. Overview of the virtual file system. Teil der Linux Kernel documentation, July 1999.

[39] Object Management Group. Unified modeling language (uml), version 2.0. Technical report, Object Management Group, `http://www.omg.org/`, 2006.

[40] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

[41] John H. Howard. On Overview of the Andrew File System. In *USENIX Winter*, pages 23–26, 1988.

[42] InfiniBand Trade Association. *InfiniBand Architecture Specification Release 1.1.* 2002.

[43] Florin Isailă. *Algorithms for Memory Hierarchies: Advanced Lectures*, chapter An Overview of File System Architectures, pages 273–289. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, July 2003.

[44] Florin Isailă. *Clusterfile: A Parallel File System for Clusters.* PhD thesis, Universität Karlsruhe, Juli 2004.

[45] Florin Isailă, Guido Malpohl, Vlad Olaru, Gábor Szeder, and Walter F. Tichy. Integrating collective I/O and cooperative caching into the Clusterfile parallel file system. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, June 2004.

[46] Florin Isailă and Walter F. Tichy. Clusterfile: A flexible physical layout parallel file system. In *Proceedings of IEEE Cluster Computing Conference, Newport Beach*, October 2001.

[47] Florin Isailă and Walter F. Tichy. Mapping functions and data distribution for parallel files. In *Proceedings of the IPDPS Workshop*, page 237, April 2002.

[48] Jeffrey Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR-3022, Network Appliance, 1997.

[49] Steve R. Kleiman. Vnodes: An architecture for multiple file system types in sun UNIX. In *USENIX Summer*, pages 238–247, 1986.

[50] D. Kotz and C.S. Ellis. Practical prefetching techniques for multiprocessor file systems. *Distributed and Parallel Databases*, 1(1):33–51, 1993.

[51] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, October 1996.

[52] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159. ACM Press, May 1990.

[53] Alan Mainwaring and David Culler. Active message applications programming interface and communication subsystem organization. Technical report, Computer Science Division, University of California at Berkeley, October 1996.

[54] Guido Malpohl and Florin Isailă. Paradis-Net: A Network Interface for Parallel and Distributed Applications. In *Proceedings of the 4th International Conference on Networking, Part 2*, pages 762–771. Springer, April 2005.

[55] Chris Mason. Journaling with ReisersFS. *Linux Journal*, 2001(82es):3, 2001.

[56] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst D. Simon. Top500 Supercomputer Sites. `http://www.top500.org/`, November 2006.

[57] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation*, pages 3–17. USENIX Association, 1996.

[58] Bruce Jay Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, 1981.

[59] Nils Nieuwejaar, David Kotz, Apratim Purakayastha, Carla S. Ellis, and Michael L. Best. File access characteristics of parallel scientific workloads. In *IEEE Transactions on Parallel and Distributed Systems, 7(10)*, October 1996.

[60] Matthew T. O'Keefe. Shared File Systems and Fibre Channel. In *Proceedings of the Sixth NASA Goddard Conference on Mass Storage Systems*, pages 1–16. IEEE Computer Society Press, 1998.

[61] Vlad Olaru. *Single System Image Servers on top of Clusters of PCs.* PhD thesis, Universität Karlsruhe, December 2004.

[62] Vlad Olaru and Walter F. Tichy. CARDs: Cluster-Aware Remote Disks. In *3rd IEEE International Symposium on Cluster Computing and the Grid*, pages 112–119, May 2003.

[63] Ron Oldfield and David Kotz. Armada: A parallel file system for computational grids. *Proc. of IEEE/ACM International Symposium on Cluster Computing and the Grid*, 2001.

[64] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the USENIX Annual Technical Conference*. The USENIX Association, June 1999.

[65] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th Symposium on Operating Systems Principles (SOSP)*, pages 15–24, December 1985.

[66] James Pinkerton, Ellen Deleganes, and Michael Krause. Sockets Direct Protocol (SDP) for iWARP oder TCP. Technical report, The RDMA Consortium, `http://www.rdmaconsortium.org`, October 2003.

[67] S. Ramasulamy and P. Banerjee. Automatic generation of efficient array redistribution routines fordistributed memory multicomputers. *Fifth Symposium on the the Frontiers of Massively Parallel Computation*, pages 342–349, 1995.

[68] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, February 1992.

[69] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 188–201, New York, NY, USA, 2001. ACM Press.

[70] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyone. Design and implementation of the Sun network file system. In *Proceedings of Usenix 1985 Summer Conference*, pages 119–130, June 1985.

[71] Mahadev Satyanarayanan, James J. Kistler, Punet Kumar, Maria E. Okasai, Ellen H. Siegel, and David C. Steere. Coda: A hignly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.

[72] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proc. of the First Conference on File and Storage Technologies (FAST)*, pages 231–244, January 2002.

[73] Robert Sedgewick. *Algorithmen in C*, chapter Erweiterbares Hashing, pages 313–319. Addison-Wesley, 1992.

[74] Robert Sedgewick. *Algorithmen in C*, chapter Binäre Suche, pages 236–240. Addison-Wesley, 1992.

[75] X. Shen and A. Choudhary. DPFS: A Distributed Parallel File System. *IEEE 30th International Conference on Parallel Processing (ICPP)*, 2001.

[76] Evgenia Smirni and Daniel A. Reed. Workload characterization of i/o intensive parallel applications. In *Proceedings of the Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, June 1997.

[77] AJ Smith. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, 7(4):403–417, 1981.

[78] Hal Stern. *Managing NFS and NIS*. O'Reilly, 2nd edition, 2001.

[79] Rainer Stiefelhagen, Hartwig Steusloff, and Alex Waibel. CHIL- Computers in der Human Interaction Loop. In *Proceedings of NIST ICASSP Meeting Recognition Workshop*, Montreal, Canada, May 2004.

[80] Oliver Strutynski and Christoph Biardzki. Design and Implementation of a benchmark for parallel file systems. Technical report, Technische Universität München, 2004.

[81] Gábor Szeder. MPI-IO Datenschnittstelle für Clusterfile. Master's thesis, Universität Karlsruhe, Karlsruhe, Deutschland, 2002.

[82] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, 3rd edition, 2006.

[83] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 878–883, 1993.

[84] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[85] `http://www.viarch.org`. *VIA: The Virtual Interface Architecture*, 1998.

[86] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[87] Randolph Y. Wang and Thomas E. Anderson. Experience with a distributed file system implementation. Technical Report CSD-98-986, University of California at Berkeley, 1998.

[88] Ting Zheng. A High Performance Implementation of Paradis-Net using RDMA. Master's thesis, Universität Karlsruhe, September 2004.