DIPLOMARBEIT

DRBD

Festplattenspiegelung übers Netzwerk
für
die Realisierung hochverfügbarer Server unter Linux

ausgeführt am Institut für Computersprachen
Abteilung für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter Anleitung von
o. Prof. Dipl.-Ing. Dr. Manfred Brockhaus
und
Univ.-Ass. Dipl.-Ing. Dr. Anton Ertl
als verantwortlich mitwirkenden Universitätsassistenten

durch

Philipp Reisner
Diehlgasse 33/21
1050 Wien

Wien, am 18. Mai 2000

Kurzfassung

Diese Arbeit zeigt, daß Hochverfügbarkeits-Cluster auch ohne teure Shared Devices implementiert werden können. Es werden das Design und die Implementierung eines Gerätetreibers (=DRBD) für Linux gezeigt, der das Spiegeln von Festplatten über das Netzwerk erlaubt. Um sowohl gute Leistung als auch Unterstützung für Journaling-Filesysteme bieten zu können, wurde ein Algorithmus entwickelt, der dem Disk-Scheduler beim Schreiben die größtmögliche Freiheit einräumt, Blöcke umzuordnen, dabei aber die Reihenfolge, die das Filesystem vorgibt, nicht verletzt.

Das Gerät erreicht zwischen 50 % und 98 % der theoretisch möglichen Leistung. Weiters gebe ich einen Überblick darüber, wie sich DRBD in die anderen Clustering-Komponenten unter Linux eingliedert.

Abstract

This work shows that it is possible to implement high-availability clusters without expensive shared devices. A description of the design and the implementation of a device driver for Linux is provided, which allows harddisk mirroring via the network. In order to be able to offer good performance and support for journaling filesystems an algorithm was developed which gives the disk scheduler maximum freedom during the write process to reorder blocks without compromising the order imposed by the filesystem.

The device reaches between 50 % and 98 % of the maximum theoretical performance. Apart from that, this work gives an overview of how DRBD is integrated into the other clustering compnents under Linux.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Unsere Gesellschaft ist auf dem Weg in das Informationszeitalter. Eines der vielen Anzeichen dafür ist das enorme Wachstum des Internets, das über 100 % im Jahr liegt. Dies stellt sogar die enorm hohen Wachstumszahlen der gesamten EDV-Branche in den Schatten.

In den vergangenen Jahrzehnten diente das Internet hauptsächlich dem (akademischen) Informationsaustausch. Durch die rasante Entwicklung hat sich das Netz auch in Richtung kommerzieller Anwendung geöffnet, wahrscheinlich ist aber auch die Kommerzialisierung ein Grund für das hohe Wachstum. Zu den populärsten kommerziellen Anwendungen gehören heute Online-Shops und Telebanking.

Doch wir sind noch nicht abhängig von kommerziellen Anwendungen, die über das Internet angeboten werden, wie wir von anderen Errungenschaften des Fortschritts abhängig sind. Nach ein paar Tagen ohne Autobahnen würde z.B. die öffentliche Versorgung zusammenbrechen, während ein Totalausfall des Internets von einem großen Teil der Bevölkerung gar nicht wahrgenommen werden würde.

Doch wie die Geschichte zeigt, werden die neuen technischen Errungenschaften allmählich in das tägliche Leben integriert, und wir werden zunehmend von ihnen abhängig. Es ist sicher nur eine Frage der Zeit, bis sich die Abhängigkeit von funktionierenden Informationssystemen, wie sie schon jetzt in kleinen Teilbereichen existiert, auch auf einen Großteil der Bevölkerung ausdehnt.

Vor allem das Finanz- und Versicherungswesen sowie viele isolierte Echtzeitsteuerungssysteme sind für uns von so großer Wichtigkeit, daß wir ihre fortlaufende Funktion unter allen Umständen sicherstellen müssen. Für diese Bereiche gibt es bereits entsprechende Lösungen.

Erhöht sich jedoch die Abhängigkeit der Gesellschaft von Online-Anwendungen weiter, so muß hier noch an deutlichen Verbesserungen gearbeitet werden, denn heute ist es nichts Unübliches, wenn Internetanwendungen wie das Elektronische Telefonbuch (www.etb.at), Telebanking oder der Wiener Stadtplan für mehrere Stunden oder Tage einfach nicht funktionieren.

Auch in Klein- und Mittelbetrieben, wo vor allem auf PC basierte EDV-Lösungen gesetzt wird, kann der Ausfall des EDV-Systems gravierende Folgen für das Unternehmen haben. Sowohl der Ausfall der Produktivität im Zeitraum des EDV-Ausfalls als auch Regreßsansprüche der Kunden, wenn wegen eines EDV-Ausfalls vereinbarte Leistungen nicht erbracht werden können, können für das Unternehmen existenzbedrohend werden.

Es werden Systeme benötigt, die garantiert zur Verfügung stehen. Im Unterschied zu Echtzeitanwendungen dürfen kurze Unterbrechungen (im Bereich von Minuten) auftreten, wenn dadurch die Kosten des Systems in den Bereich herkömmlicher Systeme gebracht werden können. Es müssen also Systeme gebaut werden, die den Ausfall einzelner Komponenten tolerieren können. Diese Systeme müssen aus herkömmlichen Computern bestehen, da nur so die Kosten gering gehalten werden können.

Diese Arbeit gibt zuerst einen Überblick über existierende Lösungen, die vor allem mit Hilfe spezieller Hardware arbeiten; anschließend wird eine neue Lösung vorgestellt, die ohne spezielle Hardware auskommt.

In **Kapitel 2** wird der Begriff **Clustering** erläutert; der Schwerpunkt liegt bei existierenden Clusterlösungen, die hohe Verfügbarkeit bieten.

In **Kapitel 3** wird das **Design von DRBD** — die zentrale Komponente des Hochverfügbarkeits-Clusters, der ohne spezielle Hardware auskommt — vorgestellt.

In **Kapitel 4** werden für die Implementierung von DRBD relevante **Teile des Linux Kernels** vorgestellt.

Kapitel 5 behandelt die **Implementierung und erzielte Leistung von DRBD**.

In **Kapitel 6** wird gezeigt, wie mit dem DRBD-Gerät und anderer frei erhältlicher Software ein **Hochverfügbarkeits-Cluster implementiert** werden kann.

Kapitel 2

Clustering

Wie bereits erwähnt können, preiswerte, hochverfügbare Systeme gebaut werden, indem mehrere herkömmliche Computer zu einem Cluster zusammengeschlossen werden. Da der Begriff Clustering aber mehr beinhaltet als Systeme für hohe Verfügbarkeit, gebe ich in diesem Kapitel zuerst einen Überblick über verschiedene Cluster und gehe danach speziell auf Cluster für hohe Verfügbarkeit ein.

Unter einem Cluster versteht man den Zusammenschluß mehrerer unabhängiger Computer, um eine Aufgabe zu lösen. Ein Cluster unterscheidet sich dadurch von einem Netzwerk von Computern, das eine verteilte Applikation ausführt, daß er in einem bestimmten Aspekt den Eindruck erweckt, daß es sich um einen einzigen, herkömmlichen Computer handelt.

Der „perfekte Cluster" wäre ein System, das sich in allen Aspekten, wie z.B. Administration, Programmiermodell, Repräsentation im Netzwerk, wie ein einzelner Computer verhält, aber mit dem Hinzufügen zusätzlicher Knoten seine Leistung und Ausfallsicherheit steigern kann. Die Sichtbarkeit des Clusters als eine Einheit wird als SSI (single system image) bezeichnet. [Pfi98]

Die meisten der heutigen Clustering-Ansätze setzen sich ein wesentlich bescheideneres Ziel. Sie versuchen, das SSI für einen bestimmten Zweck aufrechtzuerhalten. Verläßt man bei einer Betrachtung solch eines Clusters die Grenzen des SSI, so wird die Tatsache, daß es sich um mehrere Computer handelt, wieder sichtbar.

Ein Faktor, der viele kommerzielle Hersteller von EDV-Infrastruktur davon abgehalten hat, fortschrittliche Clustering-Technologien zu entwickeln, ist ihre eigene Linzenzpolitik. Wird Software für mehrere Computer, auch wenn es sich um Knoten eines Clusters handelt, verkauft, so ist für jeden Computer eine Lizenz zu erwerben. Das ist ein Grund, weshalb viele Kunden SMP-Computersysteme vorziehen, da hier nur Lizenzen für einen Computer erworben werden müssen, aber trotzdem die Leistung mehrerer CPUs zur

Verfügung steht. Die Nachfrage der Kunden wiederum beeinflußt das Angebot der Hersteller [Pfi98, Seite 511]. (Eine Ausnahme bilden jene Lizenzen, deren Preis von der Anzahl der maximal erlaubten Benutzer abhängt.)

Linux ist für den Einsatz in großen Clustern viel besser geeignet, da bei Linux keine Lizenzgebühren pro Installation anfallen. Im folgenden eine kleine Übersicht über die verschiedenen Aufgabenstellungen, die mit Clustern unter Linux gelöst werden können. [Sha00]

2.1 Wissenschaftliche Cluster

Für aufwendige Simulationen wird oft eine derart große Anzahl an Rechenoperationen benötigt, daß ein einzelner Computer viele Jahre benötigen würde, um die gesamte Simulation durchzuführen. Es gibt in diesem Bereich den Ansatz, eine Applikation speziell für den Einsatz auf einem Cluster zu entwickeln. In diesem Fall ist der Entwickler selbst für die Aufteilung der Last auf die einzelnen Knoten des Clusters zuständig. Das SSI wird hier mit Hilfe einer Bibliothek den Applikationen zur Verfügung gestellt. Die bekanntesten Bibliotheken sind PVM [Sun90] und MPI [MPI97].

2.2 Lastverteilung

2.2.1 hohe CPU-Last

Ein anderer Ansatz beruht darauf, die Tatsache, daß es sich um einen Cluster handelt, vor der Applikation zu verbergen. In diesem Fall muß die Cluster-Software für die Aufteilung der Last auf die einzelnen Knoten sorgen. Dies wird in MOSIX, einer Cluster-Software, durch die Migration von Anwendungsprozessen ermöglicht. Hier findet sich das SSI in der Systemrufschnittstelle des Linux-MOSIX Kernels wieder.

Auf die Grenzen des SSI stößt der MOSIX-Cluster bei Netzwerkverbindungen. Wenn eine Applikation einen IP-Socket verwendet, kann sie nicht mehr migriert werden.

2.2.2 hohe Netzwerk-Last

Vor allem für Webserver mit vielen Zugriffen baut man Cluster, bei denen die hereinkommenden Anforderungen der Clients auf mehrere Webserver aufgeteilt werden. Auf diesem Gebiet gibt es speziell in letzter Zeit zahlreiche Entwicklungen.

RDNS Die ersten Lösungen basierten auf RRDNS (round robin dns), wobei der DNS (domain name system) Server mit gleicher Verteilung die IP-Adressen der einzelnen Cluster-Knoten an die Clients weitergibt. Der größte Nachteil dieser Lösung liegt darin, daß die Last unter den Cluster-Knoten meist schlecht verteilt wird. Die Gründe dafür sind vielseitig, einer davon ist, daß der DNS-Server nicht kontrollieren kann, wie lange die IP-Adresse client-seitig gespeichert wird.

Application level Proxies Bei dieser Lösung wird ein Proxy-Server, wie er von der Client-Seite bekannt ist, auf der Server-Seite eingesetzt. Dieser Proxy kennt die Lastverteilung im Cluster und leitet die Anfrage des Clients an den Knoten mit der geringsten Last weiter. Diese Architektur hat jedoch den Nachteil, daß die maximale Größe des Clusters durch die Leistungsfähigkeit des Proxys begrenzt ist.

IP level Scheduling Es gibt — ebenso wie bei bei der Proxy-Lösung — einen Computer, der die Anforderungen auf die Knoten aufteilt; allerdings geschieht dies durch NAT (network address translations). Ein Vorteil gegenüber der Lösung mit dem Proxy ist, daß nicht für jedes Protokoll ein spezieller Proxy entwickelt werden muß. Die maximale Größe des Clusters wird zwar weiterhin durch die Leistungsfähigkeit des Rechners begrenzt, jedoch ist NAT im Kernel implementiert, was diesen Nachteil etwas entschärft.
Eine andere Variante wird mit Hilfe von IP-Tunnelling implementiert. Hier werden die IP-Pakete innerhalb anderer IP-Pakete zu den Knoten des Clusters befördert. Für den angesprochenen Knoten entsteht der Eindruck, daß diese Anforderung direkt vom Client kommt, daher schickt er die Antwort direkt zum Client zurück. Im Unterschied zu den Lösungen mit Proxy und NAT ist hier die Netzlast für den Knoten, der die Aufteilung der Anforderungen übernimmt, viel kleiner.

Im Rahmen des Linux Virtual Server Projektes [ZJW99] wurde eine linuxbasierte Implementierung des IP-Level-Scheduling-Ansatzes geschaffen.

Sowohl bei diesem Cluster, wie auch bei Clustern für hohe Verfügbarkeit, wird das SSI ausschließlich auf Netzwerkebene erfüllt.

2.3 Hohe Verfügbarkeit

Für viele Anwendungen ist hohe Verfügbarkeit (high availability) die wichtigste Eigenschaft. Beispiele dafür sind File-, Print-, Datenbank-, Gateway- und Applikations-Server in Unternehmen, aber auch öffentlich zugängliche Web-Server.

2.3.1 Verfügbarkeit

Verfügbarkeit (availability) wird wie folgt definiert [Kop97]:

$$A = \frac{MTTF}{MTTF + MTTR}$$

MTTF Durchschnittliche Zeit der fehlerfreien Funktion (mean time to failure).

MTTR Durchschnittliche Dauer einer Reparatur (mean time to repair).

MTBF Die durchschnittliche Zeit zwischen zwei Ausfällen ist definiert als $MTBF = MTTF + MTTR$ (mean time between failures).

Dieser Ausdruck kann erst am Ende der Lebensdauer eines Systems berechnet werden. Betrachtet man allerdings die Verfügbarkeit eines Systems in einem Jahr, so ergibt sich folgende Tabelle für die maximal erlaubte Ausfallszeit eines Systems.

Verfügbarkeit	erlaubte Ausfallszeit	Klasse
99 %	88 h	2
99,9 %	9 h	3
99,99 %	52 min	4
99,999 %	5 min	5
99,9999 %	31 sec	6

Vergleicht man dies mit der Tatsache, daß die meisten Linux-Distributoren mindestens 4 mal im Jahr neue Distributionen herausbringen, so ist es ersichtlich, daß eine Verfügbarkeit von mehr als 99 % nur mit zwei Systemen zu erreichen ist, von denen ein System in Produktion bleiben kann, während auf dem anderen Wartungsarbeiten durchgeführt werden. Zwar ist die Installation einer Linux-Distribution heute wesentlich einfacher als früher, doch nach dem Einspielen neuer Programmversionen sind oft Änderungen an Konfigurationsfiles notwendig. (Diese Problematik betrifft nicht alle Cluster, siehe auch Abschnitt 2.3.1.1.)

Verzichtet man auf die laufende Aktualisierung der Linux-Distribution, so ist es bei öffentlich zugänglichen Systemen eine Mindestanforderung, die neuesten sicherheitsrelevanten Aktualisierungen einzuspielen.

Betrachtet man nur die Hardware, dann ist es nur mehr ein kleiner Schritt, aus zwei unabhängigen Systemen einen Cluster zu bauen, bei dem ein Knoten aktiv das Service anbietet und der zweite Knoten den aktiven überwacht.

Pfister [Pfi98] führt auch den Begriff der Verfügbarkeitsklasse ein. Er gibt folgende Beispiele:

1-2	LANs
2	herkömmliche Computer
3(-4)	Hochverfügbarkeits-Cluster aus COTS-Komponenten
(3-)4	Mainframes
5	Computer des Telefonsystems
6	Bordcomputer eines Flugzeuges

COTS steht für commercial-off-the-shelf, d.h. Komponenten, die weitläufig verfügbar sind.

Bei Systemen der höchsten Klasse wird zumeist von fehlertolerierenden (fault tolerant) Systemen gesprochen. Bisher wurden ausschließlich Systeme, bei denen diese Eigenschaft in Hardware implementiert wurde und somit der Ausfall einer Komponente der Hardware für die Umgebung nicht sichtbar war, so bezeichnet.

Da Kunden den Begriff „fehlertolerierend" gerne hören, wird dieser Begriff oder Abwandlungen wie „software fault tolerant" auch gerne von findigen Anbietern von Hochverfügbarkeits-Clustern verwendet. Hochverfügbarkeits-Cluster (high availability clusters) werden im folgendem als HA-Cluster bezeichnet.

2.3.1.1 Betriebszustände

Eine Unterbrechung des Betriebes wegen eines Ereignisses, das bereits im Vorhinein bekannt ist, wird als geplante Unterbrechung (planned outage) bezeichnet. In diese Kategorie fällt auch das oben genannte Beispiel der Aktualisierung der Systemsoftware.

Das Gegenstück dazu sind (nicht geplante) Ausfälle (unplanned outages), die — wie der Name bereits vermuten läßt — auf nicht vorherzusehende Ereignisse zurückgehen. In diese Kategorie fallen Ausfälle von Hardwarekomponenten, bisher nicht bekannte Softwarefehler und Ereignisse der Umgebung (z.B. Stromausfall).

Nicht alle Computersysteme müssen jedoch rund um die Uhr verfügbar sein. Ein System für den Aktienhandel muß z.B. nur während der Börsenöffnungszeiten verfügbar sein. Bei diesen Systemen kann der Betrieb zu jenen Zeiten unterbrochen werden, in denen die Verfügbarkeit des Systems ohnehin nicht erforderlich ist; die Verfügbarkeit wird nur über den Zeitraum, der von Interesse ist, berechnet.

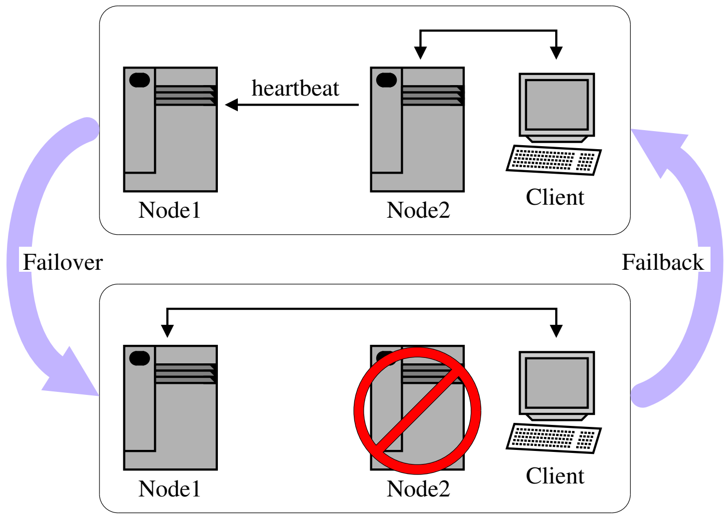


Abbildung 2.1: prinzipielle Funktion eines HA-Clusters

2.3.2 HA-Cluster

Das Prinzip, nach dem die hier besprochenen HA-Cluster arbeiten, ist der sogenannte Failover. Dabei übernimmt ein Knoten im Cluster die gesamte Arbeit des Clusters. Die einzige Aufgabe des verbleibenden Knotens ist es, ständig zu überprüfen, ob der aktive Knoten noch funktioniert. Wird der Ausfall des aktiven Knotens festgestellt, wird die Arbeit des Clusters von nun an durch den verbleibenden Knoten übernommen. In Abbildung 2.1 sind diese Vorgänge graphisch dargestellt.

HA-Cluster werden eingesetzt, um geplante Unterbrechungen und (nicht geplante) Ausfälle abzufangen. Ein Punkt, der zur Beliebtheit dieser Cluster sicher beigetragen hat, ist, daß nicht nur Ausfälle von Hardwarekomponenten, sondern auch viele Ausfälle auf Grund von Fehlern in der Software für den Anwender transparent werden.

Die Wirkung von Software-Fehler[1] die nur unter ganz bestimmten Umständen eintreten, sodaß es fast nicht möglich ist, diese Fehler zu reproduzieren und zu beheben, können auf Grund genau dieser Eigenschaft von einem HA-Cluster, der die Software ja einfach noch einmal laufen läßt, ausgeglichen werden. Bei der erneuten Ausführung der Software ist es sehr wahrscheinlich, daß Pakete vom Netzwerk in einer anderen Reihenfolge empfangen werden, und daß daher der Fehler nicht auftritt. [Pfi98, Seite 391]

[1]Pfister bezeichnet diese Fehler als Heisenbugs. Streng genommen sind nur solche Fehler, die beim „Betrachten", also dem Ausführen mit einem Debugger verschwinden oder erscheinen, Heisenbugs.

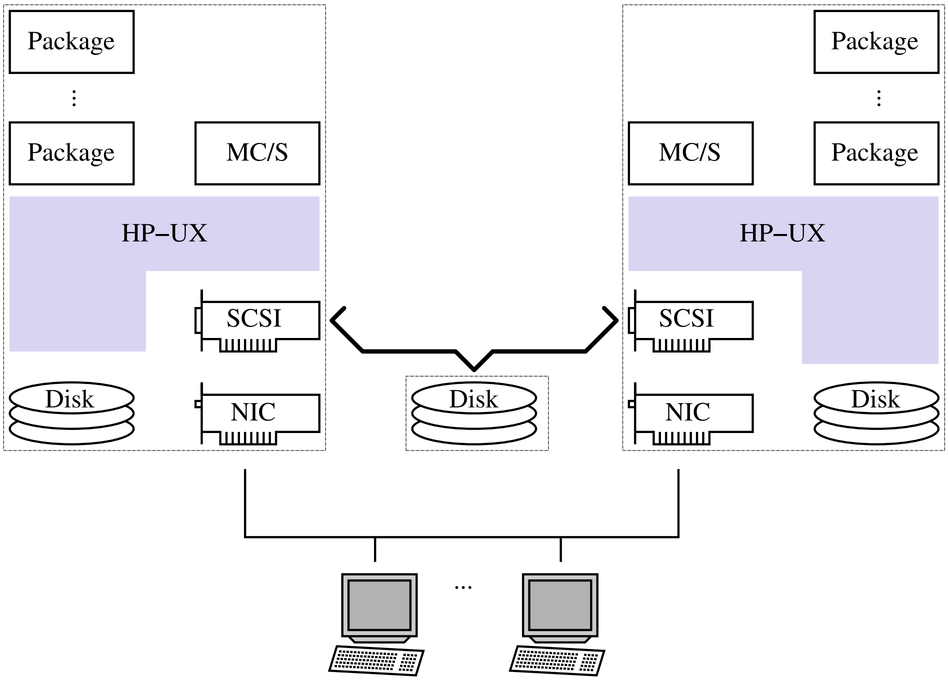


Abbildung 2.2: HP Beispielkonfiguration

Das Prinzip des Failovers ist auch das Grundprinzip der Clusterlösungen für Linux die in Kapitel 6 vorgestellt werden.

2.3.3 Beispielkonfiguration

In diesem Abschnitt wird eine Beispielkonfiguration von HP [Wey96] betrachtet, die ebenfalls nach dem Prinzip des Failovers arbeitet. Sie ist in Abbildung 2.2 schematisch dargestellt. HP wurde zufällig als Beispiel gewählt, ganz ähnliche Cluster sind auch von anderen Herstellern erhältlich.

Die lokalen Festplatten der Knoten werden für das Betriebssystem, die HA Software und die Applikationssoftware verwendet. Die Daten der Applikationen hingegen werden auf dem von beiden Knoten aus zugänglichen Festplattensystem (shared disk) gespeichert.

2.3.3.1 Shared Disk

HP realisiert die Shared Disk mit Hilfe eines SCSI-Busses. Es werden zwei SCSI-Kontroller an einen SCSI-Bus angeschlossen; natürlich müssen die beiden Kontroller auf verschiedene SCSI-IDs konfiguriert werden. Die heute üblichen SCSI-Kontroller haben in der Regel bereits aktive SCSI-Terminatoren integriert, die allerdings in diesem Anwendungsfall nicht verwendet werden können. Wenn ein Knoten des Clusters nicht in Betrieb ist, ist somit der aktive Terminator ohne Versorgung, und der gesamte SCSI-Bus ist wegen fehlender Termination unbrauchbar. Daher müssen die internen Terminatoren deaktiviert, der SCSI-Bus verlängert und an den Enden des Busses externe Terminatoren angeschlossen werden. Viele SCSI-Kontroller haben einen externen und einen internen Anschluß. Würde man den externen Terminator am internen Anschluß des SCSI-Kontrollers anbringen, könnte man den Knoten nicht vom SCSI-Bus trennen. Daher ist es unbedingt notwendig, diesen Terminator am externen Bus anzubringen. Dafür werden SCSI-Y-Kabel benötigt.

Diese Kabel sind — außer bei den HA-System-Anbietern — fast nicht erhältlich. Eine Umfrage auf der Linux-HA Mailingliste hat ergeben, daß es nur einen einzigen Händler in Europa gibt, der diese Kabel in seinem Sortiment hat.

Tritt ein Problem auf dem SCSI-Bus auf, wird z.B. der Terminator versehentlich entfernt, so sind die Zugriffswege beider Knoten unterbrochen. Ein Shared SCSI-Bus ist ein SPOF (single point of failure) für den ganzen Cluster. Daher wird bei den Clustern von HP der Shared SCSI-Bus doppelt ausgeführt. Falls eine normale SCSI-Festplatte am Shared SCSI-Bus angeschlossen ist, wird diese ebenfalls doppelt ausgeführt, und zwischen den beiden Festplatten sorgt MirrorDisk/UX für die Datenspiegelung.

Neben HP verwenden auch andere Anbieter von Unix-basierten HA-Cluster-Lösungen einen Shared SCSI-Bus, sogar im Linux-HA-HOWTO [Mil98] wird die Verwendung dieses Busses vorgeschlagen.

Shared Disks können neben SCSI auch mit SSA, Fiber Chanel (Switch und Arbitrated Loop), Digitals Star Coupler und IBMs S/390 ESCON implementiert werden. [Pfi98]

2.3.3.2 Software

In der Clusterlösung von HP übernimmt MC/ServiceGuard das Starten der hochverfügbaren Applikationen. Für jede hochverfügbare Applikation muß ein sogenanntes Package erstellt werden. Dieses Package enthält Angaben zu den benötigten Ressourcen der Applikation sowie Shell-Skripts zum Starten und Stoppen der Applikation. Zu den erwähnten Ressourcen zählen z.B. IP-Adressen, Logic Volumes und Filesysteme.

MC/ServiceGuard läuft auf allen Knoten des Clusters. Die einzelnen Instanzen stehen mit Hilfe von Heartbeat-Paketen, die über ein gemeinsames Ethernet-Lan verschickt werden, miteinander in Verbindung. Reagiert ein System auf die Heartbeat-Pakete nicht mehr, wird es vom anderen System als ausgefallen betrachtet.

Falls auf dem gerade ausgefallenen System Instanzen von Packages gelaufen sind, wird nun MC/ServiceGuard die notwendigen Ressourcen einrichten, z.B. der Netzwerkkarte die entsprechende IP-Adresse zuweisen, das SCSI-Gerät auf dem Shared SCSI-Bus belegen, die notwendigen Filesysteme anmelden und schließlich die Applikation erneut starten. Laut HP dauert so ein Wiederanlauf eines Softwaresystems 45 Sekunden zuzüglich der für das Anmelden der Filesysteme benötigten Zeit.

Daraus lassen sich zwei Anforderungen an die Software eines Clusters ableiten:

- Das Filesystem muß in der Lage sein, nach nicht ordnungsgemäßer Abmeldung binnen sehr kurzer Zeit wieder verfügbar zu sein. Diese Anforderung kann von herkömmlichen Filesystemen nicht erfüllt werden, da diese in solch einem Fall die Konsistenz der Datenstrukturen auf der Festplatte überprüfen müssen.
- Die Applikationen müssen in der Lage sein, nach nicht ordnungsgemäßer Beendigung wieder starten zu können. Sie müssen z.B. mit Inkonsistenzen in ihren dynamischen Datenfiles zurechtkommen.

2.3.3.3 Filesystem

HP setzt sein hauseigenes Online-Journaling-Filesystem ein, das Änderungen der Filesystemgröße, während es im System angemeldet ist, unterstützt. Der Journaling-Mechanismus stellt sicher, daß die Datenstrukturen auf der Festplatte auch nach einem Ausfall eines Knotens vom anderen Knoten wieder rasch in einen konsistenten Zustand gebracht werden können. Es darf aber immer nur ein Knoten das Filesystem auf der Shared Disk angemeldet haben.

Da bei simultanen Versuchen, das Filesystem anzumelden, der Inhalt der Festplatte unwiderruflich zerstört werden würde, verwendet HP ein spezielles SCSI-Lock-Kommando, um sicherzustellen, daß immer nur ein Knoten auf die Shared Disk zugreift. Stellt der HP/UX Kernel fest, daß der andere Knoten das SCSI-Lock-Kommando ausführt, beendet er sofort all seine Aktivitäten und legt das System mit einer Kernel-Panik still.

2.3.3.4 Netzwerk

In der Minimalkonfiguration (wie in Abbildung 2.2 auf Seite 12 dargestellt) gibt es ein Netzwerksegment, das sowohl für die Heartbeat-Pakete als auch den Netzwerkverkehr zwischen Clients und Server-Applikation verwendet wird.

MC/ServiceGuard unterstützt aber auch redundante Konfigurationen: Jeder Knoten verfügt über zwei Netzwerkkarten in einem oder mehreren Netzwerksegmenten, die redundant durch Bridges, die das Spanning-Tree Protokoll unterstützen, vernetzt sind.

Kapitel 3

DRBD Design

Ein Shared SCSI-Bus hat eine Reihe von Nachteilen, die ihn als Standardlösung für HA-Cluster auf der Basis von Linux nicht geeignet erscheinen lassen:

- Sehr Teuer: Zwei SCSI-Kontroller pro Knoten, zwei SCSI-Festplatten. Die bei heutigen PC Motherboards zur Standardausstattung gehörenden IDE-Kontroller würden zum Teil unbelegt bleiben.
- Geringe lokale Ausdehnung: Die geringen Maximallängen des SCSI-Busses verhindern die Aufteilung der Knoten auf mehrere Gebäude; genau das würde jedoch wesentlich bessere Chancen bieten, Katastrophen wie Feuer oder Wassereinbruch ohne Totalausfall zu überstehen. Eine Übersicht über die SCSI-Standards und deren Maximallängen:

SCSI Bezeichnung	MB/sec	Datenleitungen	Länge [m]	Geräte
SCSI 1	5	8	6	7
SCSI 2	5	8	6	7
Fast SCSI	10	8	3	7
Wide SCSI	20	16	6	15
Ultra SCSI	20	8	1,5	7
Ultra Wide SCSI	40	16	1,5	15
Ultra2 Wide SCSI	40	16	12	15
Ultra 160 SCSI	160	16	12	15
Ultra3 SCSI	160	16	12	15

- SCSI-Y-Kabel sind nur schwer erhältlich.
- Im Gegensatz zu einem Netzwerkkabel ist ein SCSI-Kabel unhandlich in der Handhabung.

Die Idee von DRBD[1] ist es, die Daten redundant in jedem Knoten des Clusters abzulegen und alle Veränderungen der Daten an alle Mitglieder des Clusters über ein normales IP-basiertes Netzwerk zu übertragen. Folgende Punkte sprechen für den Einsatz von DRBD:

- Wesentlich kostengünstiger: Da die Daten von DRBD redundant gespeichert werden, sind in den Knoten keine SCSI- oder RAID-Kontroller notwendig, und es können sowohl die ohnehin vorhandenen IDE-Kontroller als auch günstige IDE-Festplatten zum Einsatz kommen. (Eine IDE Festplatte kostet in der Regel nur die Hälfte einer SCSI-Festplatte.)
- Beliebige Ausdehnung: IP-basierende Netzwerke können die ganze Welt umspannen. Die Knoten des Clusters könnten sogar auf verschiedenen Kontinenten stehen. (Bei großer Ausdehnung schränkt allerdings die große Verzögerung des Netzwerkes den Durchsatz des DRBD-Gerätes stark ein.)
- Es sind nur Standard-Netzwerkkomponenten erforderlich.
- Netzwerkkabel sind wesentlich einfacher in der Handhabung als SCSI-Kabel.

3.1 DRBD im Kernel

In Abbildung 3.1 sind die Komponenten des Linux-Kernels so dargestellt, wie man sie auf einem herkömmlichen Server vorfindet. Der graue Bereich stellt den Kernel dar. Mit Ausnahme des Festplattentreibers sind alle dargestellten Komponenten fix im Kernel vorhanden. Da es möglich ist, Festplattentreiber in einen laufenden Linux-Kernel nachzuladen, meldet sich DRBD als Festplattentreiber im Linux-Kernel an. (Siehe Abbildung 3.2)

DRBD erhält vom Buffer-Cache Subsystem des Kernels Schreib- und Leseanforderungen. Leseanforderungen werden durch eine Leseanforderung an den lokalen Festplattentreiber erfüllt, während die Datenblöcke, die von einer Schreibanforderung stammen, auch an das zweite System über das Netzwerk weitergegeben werden. Dadurch wird der Inhalt der virtuellen Shared Disk auf beiden Festplatten des Clusters redundant abgelegt.

Grundsätzlich ist für den Betrieb von DRBD nur eine Netzwerkkarte pro System erforderlich. Es kommt aber nicht nur DRBD zugute, wenn für die Verbindung zwischen den Systemen ein eigenes Netz zur Verfügung steht. Dies hilft vor allem der Cluster-Software, zwischen dem Ausfall eines ganzen Knotens und dem Ausfall einer Netzwerkkarte zu unterscheiden.

[1] Distributed replicated block device

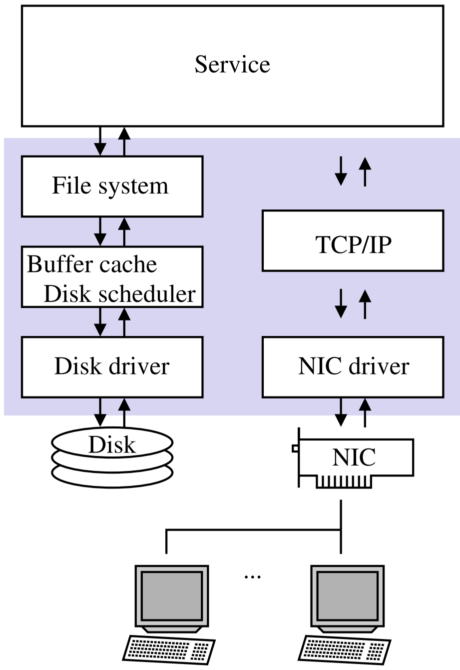


Abbildung 3.1: Datenfluß in einem einfachen Serversystem

3.1.1 Semantiken

DRBD wurde vor allem für den Einsatz in einem HA-Cluster entwickelt, Lastverteilung wurde dabei nicht berücksichtigt. Daher wurde bisher auf die exakte Nachbildung der Semantik einer Shared Disk verzichtet. Für eine exakte Nachbildung der Semantik müßten die Daten, wenn sie von DRBD über das Netzwerk empfangen werden, ohne daß diese im Buffer-Cache abgelegt werden, direkt an den darunterliegenden Festplattentreiber weitergegeben werden. Diese Semantik wäre für den Einsatz eines Shared-Disk Filesystems wie z.B. GFS[2] [PBB+99] erforderlich.

3.1.1.1 GFS

Im Unterschied zu anderen Filesystemen kann GFS ein Dateisystem, das auf einer Shared Disk liegt, von mehreren Knoten aus gleichzeitig anmelden. Um die Zugriffe der Knoten auf die Datenstrukturen auf der Festplatte zu koordinieren, ist ein verteilter Koordinationsmechanismus erforderlich. Die GFS Entwickler haben dies mit Hilfe eines speziellen SCSI-Kommandos implementiert, das auch im SCSI-Gerät implementiert sein muß.

[2] Global Filesystem

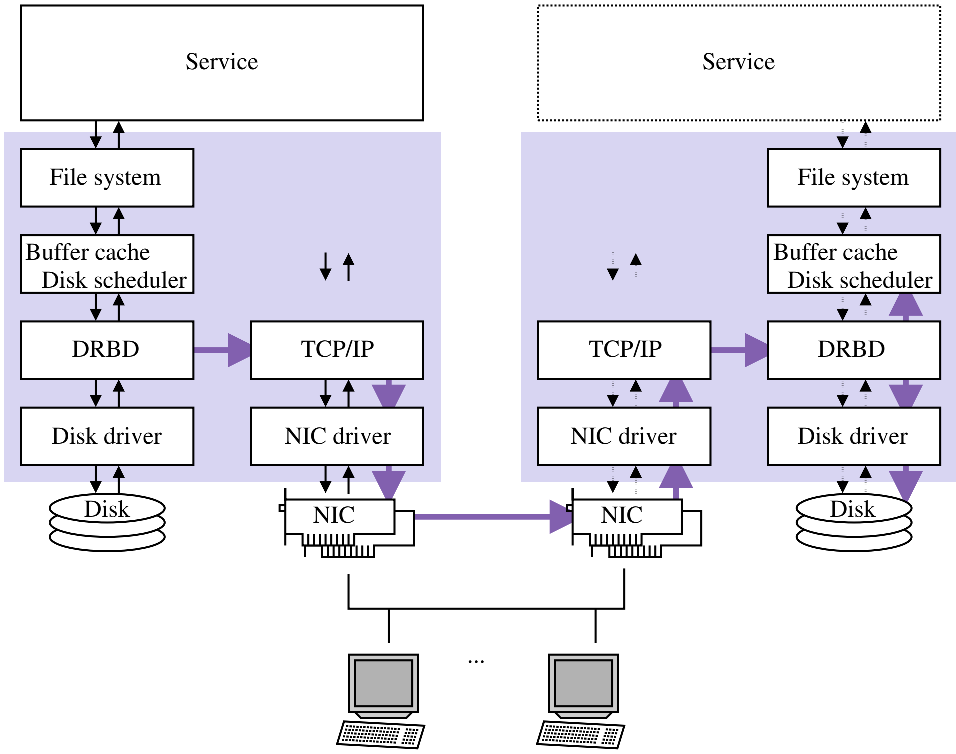


Abbildung 3.2: DRBD im Kernel

Alternativ gibt es auch die Möglichkeit, eine auf IP-Vernetzung aufsetzende Implementierung zu verwenden. Dieser Lock-Manager könnte zusammen mit einem um die exakte Shared-Disk Semantik erweiterten DRBD den Einsatz von GFS auf DRBD ermöglichen.

3.1.1.2 DRBDs Semantik

Jedes DRBD-Gerät ist einem der folgenden Zustände, die jederzeit geändert werden können, zugeordnet:

Primary Das DRBD-Gerät kann zum Schreiben und Lesen geöffnet werden. Es kann mit DRBD-Geräten auf anderen Knoten, die im sekundären Zustand sind, verbunden sein.

Secondary Das DRBD-Gerät kann nur zum Lesen geöffnet werden. Es kann mit anderen DRBD-Geräten auf anderen Knoten verbunden sein.

Die Einführung dieser Zustände entspricht der Verwendung des SCSI-Lock-Kommandos von HP auf dem Shared SCSI-Bus (siehe 2.3.3.3).

3.1.2 Single Points of Failure

Ein Single Point of Failure (=SPOF) ist eine Komponente eines Systems, deren Ausfall den Ausfall des ganzen Systems bedeutet. Ein HA-Cluster sollte keinen SPOF aufweisen.

3.1.2.1 Festplatten und deren Kontroller

Legt man die Daten auf einem Massenspeichergerät ab, das von allen Knoten des Clusters aus zugänglich ist, so ist der Zugriffsweg ein prinzipieller SPOF.

Dieser SPOF existiert bei DRBD nicht. Fällt die Festplatte oder eine Komponente auf dem Zugriffsweg zur Festplatte aus (z.B. Kontroller, Kabel, PCI-Slot, etc.), wird einfach der gesamte Knoten als ausgefallen betrachtet, und der zweite Knoten, der ja ebenfalls mit einer Kopie der Daten ausgestattet ist, übernimmt.

3.1.2.2 Netzwerk

Falls das Netzwerk zwischen den Knoten des Clusters ausfällt, ist dies für Cluster beider Arten gleichermaßen ein Problem (split brain problem). Ist das Netzwerk zu den Clients nicht unterbrochen oder handelt es sich nur

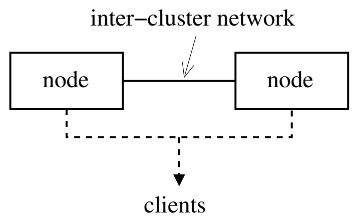


Abbildung 3.3: Modell des HA-Clusters

um eine vorübergehende Störung des Netzwerkes, kann es dazu kommen, daß beide Knoten annehmen, daß der jeweils andere ausgefallen ist. Im günstigsten Fall können die Clients keinen der Cluster-Knoten erreichen, im schlimmsten Fall zerstört der aktive Knoten den Datenstand.

Damit der Ausfall des Netzwerkes nicht zu diesem skizzierten Szenario führt, sollte es einen weiteren Kommunikationsweg zwischen den Knoten geben. Hierfür wird gerne ein serielles Kabel verwendet, da serielle Schnittstellen meistens vorhanden sind und die Kommunikation über die serielle Schnittstelle nicht die Funktion des IP-Subsystems im Kernel erfordert.

3.2 Die Protokolle

Der Nachteil, der möglicherweise bei der Verwendung eines IP-basierten Netzwerkes gegenüber eines SCSI-Busses entsteht, ist, daß die Zeit, bis ein einzelners Paket sein Ziel erreicht, auf einem IP-Netzwerk höher sein kann als auf einem SCSI-Bus bei gleicher Bandbreite.

Dieser Nachteil wird besonders deutlich, wenn in den Knoten des DRBD-Clusters auch SCSI-Festplatten verwendet werden. Die Verzögerung, die durch die Übertragung über das IP-Netzwerk entsteht, betrifft nur DRBD-basierte Cluster, während die Verzögerung des SCSI-Busses den DRBD-Cluster und Shared SCSI-Bus-Cluster gleichermaßen betrifft.

Aus diesem Grund ist DRBD mit drei Protokollen ausgestattet, die verschiedene Abstufungen des Durchsatz-Konsistenz-Kompromisses bieten. Für die Betrachtung der Protokolle im Falle des Ausfalls einer Komponente wird das in Abbildung 3.3 dargestellte Modell verwendet.

Dabei wird davon ausgegangen, daß jede der Komponenten, die beiden Knoten und das verbindende Netzwerk, entweder funktionieren oder nicht funktionieren.

3.2.1 Protokoll A

Bei Protokoll A wird eine Schreibanforderung als abgeschlossen betrachtet, sobald der Schreibvorgang auf die physische Festplatte abgeschlossen ist und die Daten gesendet wurden. Da TCP, das verwendete IP-Protokoll, ein peer-to-peer Protokoll ist, funktioniert der Kommunikationskanal nur solange beide Endpunkte der Kommunikation funktionieren.

Fällt nun der sendende Knoten unmittelbar nach dem Abschicken eines Datenpaketes aus, so kann TCP nicht sicherstellen, daß dieses Datenpaket auch beim empfangenden Knoten ankommt. Übernimmt nun der noch funktionierende Knoten das Service, kann es sein, daß er die letzten Datenpakete, die der gerade ausgefallene Knoten noch gesendet hat, nicht bekommen hat.

Dieses Verhalten ist für eine transaktionsbasierte Anwendung nicht akzeptabel. Eine Transaktion muß permanent oder gar nicht durchgeführt werden. Hat der sendende Knoten, bevor er ausgefallen ist, dem Client mitgeteilt, daß die Transaktion funktioniert hat, sind die Veränderungen durch die Transaktion nach der Übernahme durch den zweiten Knoten nicht mehr vorhanden. Eine der grundlegenden Eigenschaften einer Transaktion wurde verletzt.

Mit Protokoll A kann kein Ausfall einer Komponente (des Modells aus Abbildung 3.3) toleriert werden, ohne die Transaktionseigenschaften zu verletzen. Protokoll A findet jedoch seine Anwendung, wenn auf hohen Datendurchsatz, nicht aber auf die Transaktionseigenschaften wertgelegt wird.

3.2.2 Protokoll B

Bei der Verwendung von Protokoll B wird ein Schreibvorgang dann als abgeschlossen betrachtet, sobald der Schreibvorgang auf die lokale Platte abgeschlossen ist und eine Empfangsbestätigung vom zweiten Knoten eingetroffen ist.

Im Gegensatz zu Protokoll A bleiben beim Ausfall einer Komponente die Transaktionseigenschaften erhalten. Fallen jedoch zwei Komponenten auf einmal aus, so kann es auch bei diesem Protokoll zu einer Verletzung der Transaktionseigenschaften kommen. Ein Beispiel:

> Beide Knoten eines Clusters fallen in kurzem zeitlichen Abstand wegen eines Stromausfalls aus. Der primäre Knoten teilt einem Client mit, daß eine Transaktion abgeschlossen ist, danach fällt er aus. Der sekundäre Knoten hat zwar den Datenblock empfangen und die Empfangsbestätigung zurück an den primären Knoten gesendet, fällt jedoch aus, bevor der Datenblock auf die Festplatte geschrieben wird.
> Entscheidet der Cluster-Manager nach dem Stromausfall, daß der andere Knoten das Service anbieten soll, so sind die Auswirkungen der letzten Transaktion verloren.

Dieses Problem könnte man dadurch lösen, daß der Cluster-Mananger nach dem Ausfall beider Knoten, das Service wieder auf jenem Knoten startet, auf dem es zuletzt gelaufen ist.

3.2.3 Protokoll C

Bei Protokoll C wird ein Schreibvorgang abgeschlossen, sobald der Schreibvorgang auf die lokale Platte abgeschlossen ist und eine Bestätigung vom zweiten Knoten eingetroffen ist, die besagt, daß der Datenblock auch dort erfolgreich geschrieben wurde.

Mit Protokoll C bleibt die Transaktionssemantik auf jeden Fall erhalten.

Bei Protokoll B und C muß bei Ausfall des verbindenden Netzwerkes oder des zweiten Cluster-Knotens der Empfang der noch ausstehenden Bestätigungen simuliert werden. Das Block-IO-System unter Linux erlaubt nur eine begrenzte Anzahl nicht abgeschlossener Anforderungen, und es würde blockieren, wenn nicht alle Anforderungen abgeschlossen werden.

3.3 Schreiboperationen

Für viele Filesysteme ist die Reihenfolge, in der bestimmte Datenblöcke auf die Festplatte geschrieben werden, von entscheidender Bedeutung. Zum Beispiel muß ein Journaling-Filesystem vor jeder Metadatenaktualisierung eine Transaktion in den Journalbereich schreiben, wobei der Commit-Record als letzter geschrieben werden muß.

Unter Linux wird derzeit explizit auf den Abschluß jener Schreiboperationen gewartet, die vor den nächsten Schreiboperationen auf die Festplatte geschrieben werden müssen, bevor die folgende Schreiboperation in Auftrag gebeben wird.

Damit DRBD auf dem sekundären Knoten alle Schreiboperationen in der gleichen Reihenfolge ausführt, in der sie auf dem primären Knoten vom IO-System an den Treiber gesendet wurden, müßte der Algorithmus wie folgt aussehen:

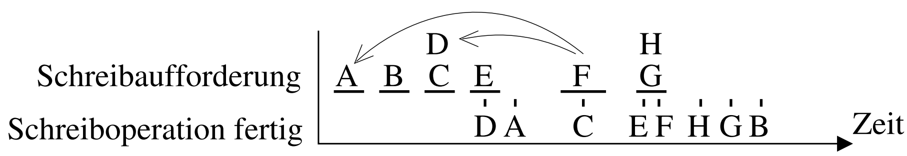
1. Empfange einen Block vom Netzwerk.
2. Schreibe diesen Block auf die Festplatte.
3. Warte, bis die Schreiboperation abgeschlossen ist.

Dieser restriktive Algorithmus würde wahrscheinlich sowohl den primären Knoten bremsen, als auch das IO-System des sekundären Knotens nicht auslasten können. Der Datendurchsatz kann wesentlich verbessert werden, wenn das Empfangen vom Netzwerk und das Schreiben auf die Festplatte parallel stattfinden.

3.3.1 Abhängigkeiten

Ob ein bestimmter Datenblock vor einem anderen Datenblock auf die Festplatte geschrieben werden muß, weiß leider nur das Filesystem. Besteht diese Notwendigkeit, so existiert zwischen dem Abschluß der Schreiboperation des ersten Blockes und dem Schreibbefehl für den zweiten Block eine Abhängigkeit.

Es kann keine Abhängigkeit zwischen zwei Blöcken geben, wenn der Schreibbefehl für den zweiten Block gegeben wird, bevor die Schreiboperation des ersten abgeschlossen ist. Ein Beispiel:



Es folgen nun ein paar Erläuterungen zum Diagramm:
Die zeitliche Ausdehnung, mit der die Übergabe von Datenblöcken an das IO-System dargestellt ist, entspricht einem Aufruf von ll_rw_block(). Aus der Abbildung ist ersichtlich, daß auch mehrere Blöcke auf einmal übergeben werden können und daß Aufrufe von ll_rw_block() auch verschieden lang dauern können, denn ll_rw_block() kann blockiert werden. Der Abschluß einer Schreiboperation ist als Zeitpunkt dargestellt, der dem Zeitpunkt entspricht, zu dem das uptodate Bit im korrespondierenden Buffer-Head gesetzt wird.

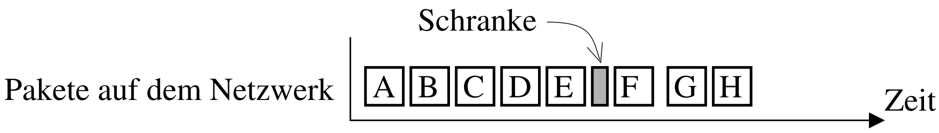
Zwischen den Datenblöcken A, B, C, D und E kann es keine Abhängigkeit geben, da vor deren Schreibbefehlen keine Schreiboperation abgeschlossen wurde. F kann aber von A und D abhängen, da der Schreibbefehl für F erst erteilt wurde, nachdem im System bekannt wurde, daß A und D vollständig auf die Festplatte geschrieben wurden. Ebenso kann es eine Abhängigkeit von G und H zu A, C und D geben. Es kann aber keine Abhängigkeit von H und G zu E und F geben. Ob diese Abhängigkeit wirklich existiert, weiß nur der Auftraggeber der Schreiboperationen (das Filesystem). Auf der Ebene eines Festplattentreibers darf man keine mögliche Abhängigkeit verletzen.

3.3.1.1 Write Barriers

Diese Erkenntnis ermöglicht es, einen Algorithmus anzugeben, der Schranken (write barriers), über die Schreibanforderungen nicht verschoben werden dürfen, erzeugt.

1. Jeder Block, der geschrieben werden muß, wird zu einer Menge hinzugefügt.
2. Ist das Schreiben einer der Blöcke aus der Menge abgeschlossen, wird die Menge geleert, und eine Schranke muß ausgegeben werden.

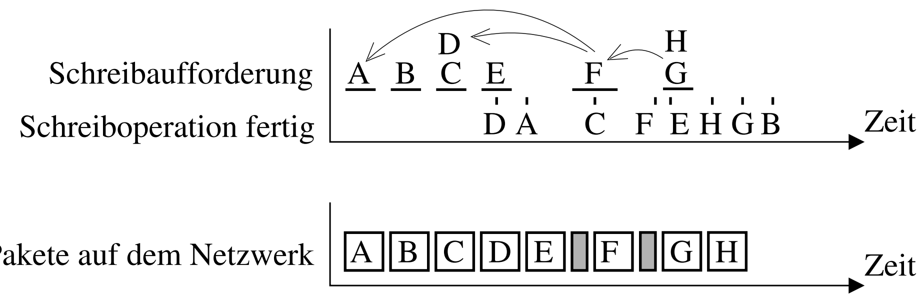
Wendet man den Algorithmus auf das Beispiel an, so werden folgende Pakete über das Netzwerk versendet:



Nun kann der Algorithmus für den Empfang der Pakete folgendermaßen verbessert werden:

1. Empfange ein Paket.
   (a) Falls das Paket einen Datenblock enthält, gib den Befehl, den Datenblock auf die Festplatte zu schreiben, und merke dir den Datenblock in einer Menge.
   (b) Falls das Paket eine Schranke ist, warte auf den Abschluß der Schreiboperationen aller Datenblöcke in der Menge, und leere die Menge anschließend.

Hier noch ein Beispiel:



3.4 Synchronisation

Neben der Replikation der Datenblöcke, die Bestandteil der normalen Schreiboperationen ist, muß es auch die Möglichkeit geben, den Inhalt der gespiegelten Festplatten wieder in den aktuellen Zustand zu bringen, wenn wegen des Ausfalls einer Komponente die Festplatten nicht mehr die gleichen Daten enthalten.

Die offensichtliche Lösung für das Problem ist es, alle Blöcke einer der beiden Festplatten auf die andere zu kopieren. Diese Vorgangsweise ist allerdings für viele Anwendungsfälle nicht geeignet. Bei großen Partitionen z.B. (Terrabyte) würde der Kopiervorang sehr lange dauern, bei geographisch verteilten Knoten kann die Notwendigkeit, derart viele Daten zu transportieren, mit hohen Kosten verbunden sein.

3.4.1 Schnelle Synchronisation

Daher ist in DRBD noch zusätzlich ein verbessertes Synchronisationsverfahren vorhanden. Dabei werden nur jene Blöcke kopiert, die verändert wurden, während die Verbindung zum Knoten mit der gespiegelten Festplatte unterbrochen war.

Dafür ist es notwendig, genau zu wissen, welche Datenblöcke vom empfangenden Knoten ordnungsgemäß auf die Festplatte geschrieben wurden. In den Protokollen A und B ist aber eine solche Benachrichtigung nicht vorgesehen. Daher wird ein weiteres Paket in das Protokoll aufgenommen. Dieses Paket wird vom sekundären Knoten an den primären gesendet, wenn alle Datenblöcke einer der in Abschnitt 3.3.1.1 eingeführten Menge geschrieben wurden. Dieses Bestätigungspaket ist in allen drei Protokollen vorhanden.

Bricht nun die Verbindung zwischen zwei Knoten zusammen, wird am sendenden Knoten angenommen, daß alle Pakete, die in noch nicht bestätigten Mengen enthalten sind, noch nicht geschrieben wurden. Diese Blöcke und alle Blöcke, die, während die Verbindung unterbrochen ist, geschrieben werden, werden in einem Bitfeld markiert.

In diesem Bitfeld entspricht jedes Bit einer bestimmten Datenmenge, die in keiner fixen Beziehung zur Blockgröße auf dem Gerät steht.

Wenn die Kommunikation wieder hergestellt ist, läuft die Synchronisation parallel zum normalen Betrieb. Die Bandbreite, die der Synchronisationsprozeß maximal belegen darf, muß vom Anwender konfiguriert werden.

Die Datenblöcke, die im Zuge der Synchronisation versendet werden, stehen außerhalb des in Abschnitt 3.3.1.1 beschriebenen Write-Barrier-Mechanismus. Für sie wird immer eine Schreibbestätigung nach Protokoll C zurückgeschickt. Das entsprechende Bit im Bitfeld wird erst dann wieder gelöscht, wenn für alle Blöcke, die vom Bit abgedeckt sind, eine Schreibbestätigung vorliegt. Das ermöglicht die Fortsetzung eines schnellen Synchronisationsvorganges nach einer erneuten Unterbrechung des Kommunikationskanals, ohne daß eine Unvollständigkeit der Synchronisation befürchtet werden muß.

3.4.2 Datensicherheit

Vor der Synchronisation enthält die gespiegelte Festplatte einen veralteten aber konsistenten Datenstand, wärend der Synchronisation ist ihr Filesystem in keinem konsistenten Zustand. Fällt nun während der Synchronisation die Festplatte des Knotens mit dem aktuellen Datenstand aus, kann nur mehr ein Backup helfen, denn der Zustand auf der gespiegelten Festplatte, eine Mischung aus dem veralteten Zustand und dem aktuellen Zustand, ist nicht brauchbar.

Während der Synchronisation bietet ein Cluster aus zwei Knoten keine Sicherheit.

Dieses Problem könnte durch den Einsatz eines Log-Structured-Filesystems gelöst werden. Es müßten der Synchronisationsmechanismus und das Filesystem eng zusammenarbeiten und z.B. die Synchronisation als eine einzige Transaktion auf dem empfangenden Knoten durchführen.

Kapitel 4

Linux

Dieses Kapitel gibt einen kleinen Überblick über Gerätetreiber unter Linux und bildet die Grundlage für die weiterführenden Kapitel. Die Informationen in diesem Kapitel wurden hauptsächlich dem Quellcode des Kernels entnommen. Die meisten Bücher [Rus98][BBDK97][Rub98] und Artikel [Rub97] über den Linux-Kernel geben zwar einen guten Überblick über den gesamten Kernel, jedoch keine Details über den Buffer-Cache.

4.1 GPL

Die auffallendste Eigenschaft des jetzt so populären Betriebssystems Linux ist seine Lizenz, die Gnu General Public License. Diese Lizenz ermöglicht einerseits die rasend schnelle Verbreitung des Systems, sichert aber auch dessen Weiterentwicklung.

Die Lizenz räumt dem Anwender das Recht ein, die betroffene Software zu modifizieren. Wird Software, die unter den Bestimmungen der GPL steht, als Teil eines größeren Produktes oder als eigenständiges Produkt in binärer Form weitergegeben, muß dem Empfänger der Software der Quellcode zugänglich gemacht werden. Es muß auch der Quellcode aller Veränderungen, die eventuell an dem Produkt vorgenommen wurden, an den Benutzer weitergegeben werden.

4.2 Gerätetreiber

Zu den grundlegenden Aufgaben eines Betriebssystems gehört es, Zugriffe auf die verfügbare Hardware in geregelten Bahnen zu erlauben. Linux bietet hier grundsätzlich die Möglichkeit für zeichenorientierte Geräte (character devices), blockorientierte Geräte (block devices) und Netzwerkkartentreiber (network drivers).

Im Filesystem sind die Geräte durch spezielle Einträge für Anwenderprogramme sichtbar. Diese Einträge (special files) tragen zwei Nummern, die für den Kernel Bedeutung haben. Die erste Nummer identifiziert den angesprochenen Treiber, die zweite das angesprochene Gerät des Treibers. So ist das Gerät /dev/hda4, die vierte Partition des ersten IDE-Gerätes, für den Kernel durch das Tupel (3,4) identifiziert.

Die erste der beiden Nummern (major number) wird innerhalb des Kernels verwendet, um die Einsprungspunkte des Treibers in einem statischen Feld zu finden.

Obwohl der Kernel nicht in einer objektorientierten Programmiersprache implementiert ist, kann man das Design der Treiber objektorientiert beschreiben:

> Das VFS[1]-Interface ist die abstrakte Basisklasse aller Treiber. Davon abgeleitet sind die Sprungtabellen der einzelnen Treiber, da sie die abstrakten Funktionen des Interfaces mit reellen Prozeduren füllen. Bei der Initialisierung eines Treibers sucht dieser nach der vorhandenen Hardware und erzeugt für jede gefundene Instanz der Hardware ein Objekt des Treibers.

4.2.1 Zeichenorientierte Treiber

Das VFS-Interface, ist in include/linux/fs.h folgendermaßen definiert:

```
struct file_operations {
     loff_t (*llseek) (struct file *, loff_t, int);
     ssize_t (*read) (struct file *, char *, size_t, loff_t *);
     ssize_t (*write) (struct file *, const char *, size_t,
                       loff_t *);
     int (*readdir) (struct file *, void *, filldir_t);
     unsigned int (*poll) (struct file *,
                           struct poll_table_struct *);
     int (*ioctl) (struct inode *, struct file *,
                   unsigned int, unsigned long);
     int (*mmap) (struct file *, struct vm_area_struct *);
     int (*open) (struct inode *, struct file *);
     int (*flush) (struct file *);
     int (*release) (struct inode *, struct file *);
     int (*fsync) (struct file *, struct dentry *);
```

[1] Virtual Filesystem Switch

```
        int (*fasync) (int, struct file *, int);
        int (*check_media_change) (kdev_t dev);
        int (*revalidate) (kdev_t dev);
        int (*lock) (struct file *, int, struct file_lock *);
};
```

Zeichenorientierte Treiber können open(), release(), read(), write(), ioctl(), poll(), llseek(), mmap() und fasync() implementieren.

open() Wird aufgerufen, wenn ein Programm open(2) mit dem entsprechenden Geräteeintrag im Filesystem aufruft. Vom Kernel wird automatisch eine Instanz vom Typ file angelegt und mit open() dem Treiber übergeben. Wenn der Treiber ein Gerät, das von mehreren Prozessen gleichzeitig geöffnet werden kann, verwaltet, so hat er die Möglichkeit, in file->private_data einen Zeiger auf den eigenen Kontext abzulegen, der genau dem momentanen Prozeß und dem angesprochenen Gerät zugehörig ist. Handelt es sich um ein Gerät, das von nur einem Prozeß angesteuert werden kann, so kann open() mit einem entsprechenden errno-Code beendet werden.

release() Entspricht dem close(2) im Kontext eines Anwenderprogrammes.

read() Kopiert Daten vom Gerät in den Adreßraum des Prozesses.

write() Sendet Daten vom Adreßraum des Prozesses zum Gerät.

mmap() Mit dieser Funktion kann der Prozeß eine Repräsentation des Gerätes in seinen Adreßraum einblenden. Dies wird z.B. bei den neuen Graphiktreibern von Linux 2.2.x genutzt, um dem Prozeß den Framebuffer direkt zugänglich zu machen. Dabei kann allerdings über das physische Format der graphischen Daten nicht abstrahiert werden. Das VM-Subsystem des Kernels bietet für die Implementierung den remap_page_range() Aufruf.
Die neuen Soundtreiber hingegen bieten über mmap() die Möglichkeit, einen Puffer mit dem Prozeß zu teilen, sodaß das aufwendige Kopieren der Daten bei read() oder write() entfallen kann.
Um physischen Speicher in den Adreßraum des Prozesses einzublenden, gibt es leider keine maßgeschneiderte Funktion. Es ist entweder möglich, mit Hilfe der VM-Primitive die entsprechenden Page-Table-Einträge selbst zu erzeugen, oder ein eigenes VM Objekt mit einem eigenen vm_ops->nopage() Handler zu erzeugen.

poll() Muß implementiert werden, um Aufrufe von poll(2) und select(2) zu ermöglichen. Diese Prozedur muß folgende Semantik haben:

Daten vorhanden	Gibt die zutreffenden Ereignisse im Rückgabewert zurück.
keine Daten vorhanden	Läßt den Prozeß mit poll_wait() schlafen, bis Daten vorhanden sind, und gibt 0 zurück.

ioctl() Mit Hilfe dieser Funktion können Metadaten zwischen einem Prozeß und dem Gerätetreiber ausgetauscht werden. Während z.B. ein Treiber für die serielle Schnittstelle mit read(2) und write(2) den Datenstrom zugänglich macht, kann der Prozeß über den ioctl(2) Systemruf die Baudrate, die Anzahl der Datenbits, etc. einstellen.

Mit Hilfe dieses Interfaces können Treiber für zeichenorientiere Geräte komplett gekapselt werden. Es ist daher auch mit geringem Aufwand möglich, sie zu modularisieren. Diese Module können zur Laufzeit des Kernels nach Belieben nachgeladen und entfernt werden.

4.2.2 Blockorientierte Geräte

Das VFS-Interface wird nicht nur für zeichenorientierte Geräte verwendet, sondern auch für blockorientierte Geräte und alle anderen Objekte, die in einem Filesystem abgelegt werden können (FIFOs, Unix-Domain-Sockets, Files). Es wäre theoretisch möglich, ein blockorientiertes Gerät aufbauend auf das VFS-Interface zu implementieren und dabei die read() und write() Funktionen zu implementieren. Dieses Gerät wäre allerdings für Filesysteme unbrauchbar, da diese den Buffer-Cache ansprechen und nicht das VFS-Interface des Gerätes verwenden. Die korrekte Vorgangsweise besteht darin, bei read() und write() die generischen Funktionen block_read() und block_write() anzugeben. Diese Funktionen übertragen die Daten vom bzw. in den Buffer-Cache.

4.2.2.1 Buffer-Cache

Es gibt eine ganze Reihe von Caches unter Linux, doch der Buffer-Cache ist mit blockorientierten Geräten unweigerlich verbunden. Der Buffer-Cache verwaltet Datenblöcke blockorientierter Geräte während sie im Speicher liegen. Jedem dieser Datenblöcke ist eine Instanz vom Typ buffer_head zugeordnet, die alle Metadaten zu dem Datenblock enthält. Die wichtigsten Felder von buffer_head:

```
...
unsigned long b_blocknr; /* block number */
```

```
    unsigned long b_size;     /* block size */
    kdev_t b_dev;             /* device (B_FREE = free) */
    unsigned long b_state;    /* buffer state bitmap (see above) */
    unsigned int b_count;     /* users using this block */
    void (*b_end_io)(struct buffer_head *bh, int uptodate);
    void *b_dev_id;
    ...
```

b_blocknr Die Position des Blockes auf dem Gerät.

b_size Die Größe des Blockes in Bytes. Alle Blöcke eines Gerätes, die zu einem Zeitpunkt im Speicher sind, haben die gleiche Größe. Die Blockgröße eines Gerätes kann zur Laufzeit geändert werden. Dabei werden alle Blöcke mit der alten Größe aus dem Buffer-Cache entfernt.

b_dev Das Gerät, zu dem dieser Block gehört.

b_count Referenzzähler dieses Blockes.

b_state Die wichtigsten Zustände eines Blockes:

dirty	Der Block wurde verändert und muß auf das Gerät zurückgeschrieben werden.
uptodate	Der Block wurde gelesen oder geschrieben und entspricht somit dem Block des Gerätes.
locked	Die Datenübertragung vom oder zum Gerät findet derzeit statt.

b_end_io Diese Funktion wird nach dem Abschluß einer IO-Operation aufgerufen.

b_dev_id Dieser Zeiger wird der b_end_io Funktion übergeben.

Die Aufgabe des Buffer-Caches ist es, die Zugriffe auf das Gerät zu optimieren und den darauf aufsetzenden Dateisystemen eine einfach zu handhabende Verwaltung der Pufferspeicher zur Verfügung zu stellen. Der Buffer-Cache wächst, wenn genug freier Speicher vorhanden ist und IO-Operationen durchgeführt werden. Wird der physische Speicher knapp, dann wird der Buffer-Cache verkleinert, indem Blöcke, die weder verändert wurden (dirty) noch referenziert sind, verworfen werden. Dateisysteme verwenden unter anderem die folgenden Prozeduren des Buffer-Caches:

getblk() Liefert einen buffer_head. Wenn der Block schon im Buffer-Cache war, wird der Referenzzähler erhöht, wenn nicht, bekommt man einen neuen, der auf einen uninitialisierten Block zeigt.

brelse() Gibt einen Block frei. Falls der Block nicht mehr referenziert wird und er verändert wurde, wird der Zeitpunkt festgelegt, zu dem der Block auf das Gerät zurückgeschrieben wird.

ll_rw_block() Führt eine IO-Operation (Lesen oder Schreiben) auf den angegebenen Blöcken aus. Diese Funktion kehrt vor dem Abschluß der Operation zurück.

wait_on_buffer() Wartet auf den Abschluß einer IO-Operation.

4.2.2.2 Request-Queue

Die Request-Queue ist ein Feld statischer Größe, in das ll_rw_block() die IO-Aufträge einträgt. Beim Eintragen werden die Aufträge so geordnet, daß die Bewegungen des Schreib-/Lesekopfes der Festplatte optimiert werden. Zur Zeit wird der sogenannte Elevator-Algorithmus implementiert. Früher wurden Leseoperationen gegenüber Schreiboperationen bevorzugt, doch es hat sich herausgestellt, daß dadurch die Gesamtleistung des Systems herabgesetzt wurde. Jetzt werden Schreib- und Leseoperationen mit gleicher Priorität behandelt, die Schreiboperationen dürfen allerdings nur 66 % der Request-Queue ausfüllen. Das letzte Drittel kann nur von Leseoperationen belegt werden.

Die Aufträge, die einen Gerätetreiber betreffen, werden durch Zeiger zu einer Liste verkettet, und die Abarbeitung dieser Liste wird durch den Aufruf der do_request() Prozedur des Treibers gestartet. Die Zeiger auf die do_request() Prozeduren sind in einem globalen Feld abgelegt, die major number der Treiber gibt die Position im Feld an.

Doch leider wurden bei der Implementierung der Request-Queue die Konzepte nicht so deutlich getrennt wie in anderen Bereichen des Kernels. So gibt es eine Unzahl spezieller Eigenheiten in den einzelnen Treibern, die sich in der einen oder anderen Form im Quellcode der Request-Queue wiederfinden, die, würde man den Code der Treiber streng von den allgemeinen Teilen des Kernels trennen, dort nichts verloren hätten.

Mit Linux 2.3.38 wird diese Situation nun deutlich verbessert. Es ist nun möglich, die IO-Aufträge eines Gerätes in einer eigenen Request-Queue zu verwalten. Diese gerätespezifische Request-Queue ist in einer Datenstruktur abgelegt, die auch 6 Zeiger auf Prozeduren enthält, die die Verwaltung der IO-Aufträge in der Request-Queue steuern.

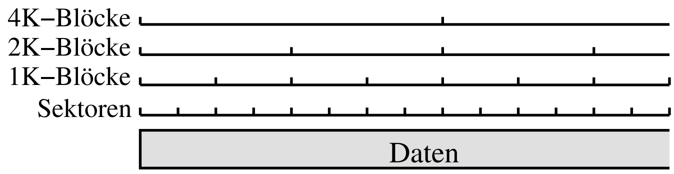
4.2.2.3 Blocknummern

Der Buffer-Cache identifiziert die von ihm verwalteten Blöcke durch Nummern. Welche Daten welcher Nummer zugeordnet werden, wird durch die momentane Blockgröße festgelegt. Diese Blockgröße kann jederzeit geändert werden. Bei Dateisystemen z.B. ist es beliebt, den Superblock mit einer Blockgröße von 1K zu schreiben, während auf alle anderen Blöcke mit einer Blockgröße von 4K zugegriffen wird.

Jedesmal, wenn die Blockgröße geändert wird, müssen alle Blöcke, die momentan im Buffer-Cache sind, aus dem Buffer-Cache entfernt werden, da sich die Bedeutung der Blocknummern ändert.

In den IO-Requests, die die do_request() Prozeduren der Treiber bekommen, werden die Daten als Sektoren adressiert, wobei jeder Sektor 512 Bytes hat.

Wenn jedoch die Größe eines Gerätes angegeben wird, werden 1K-Blöcke verwendet.



4.2.2.4 Die Rolle des VFS-Interfaces

Die Prozeduren, die die Brücke zwischen dem VFS-Interface und dem Buffer-Cache bilden, sind für alle blockorientierten Geräte gleich, nämlich block_read(), block_write() und block_fsync(). Diese Prozeduren kommen somit zum Einsatz, wenn ein Prozeß ein blockorientiertes Gerät direkt anspricht und nicht den Umweg über ein Dateisystem geht. Daher muß sich der Autor eines Gerätetreibers nur um die Implementierung von open() und release() der im VFS vorkommenden Prozeduren kümmern. Des weiteren beinhaltet das VFS-Interface zwei Prozeduren, die nur von blockorientierten Geräten genutzt werden können: check_media_change() und revalidate(). Die Rolle des VFS in Linux vor der Version 2.3.38 ist in Abbildung 4.1 dargestellt.

Da sich der von blockorientierten Geräten und von zeichenorientierten Geräten gemeinsam benutzte Teil des VFS-Interfaces auf open() und release() beschränkt, haben sich nun die Kernel-Entwickler dazu durchgerungen in Zukunft, das Interface für blockorientierte Geräte von dem der zeichenorientierten Geräte zu trennen. Diese Trennung wurde beginnend mit der Version 2.3.38 eingeführt.

4.3 Parallelausführung

Neben der Parallelausführung, die dadurch zustande kommt, daß ein Computer mit mehreren Prozessoren ausgerüstet ist, gibt es noch eine Vielzahl an

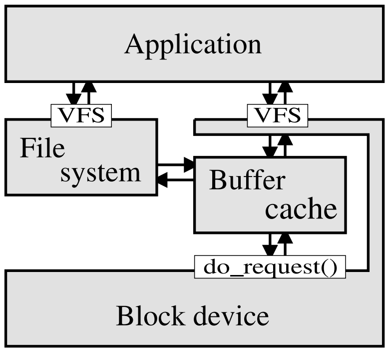


Abbildung 4.1: Rolle des VFS-Interfaces

verschiedenen Ausführungskontexten im Linux-Kernel. Da der Zugriff auf viele Ressourcen, die aus mehreren Ausführungskontexten verwendet werden, nicht atomar ist, benötigt man geeignete Mechanismen, um den Zugriff zu regeln. [Russ00]

4.3.1 Ausführungskontexte

Interrupts Interrupthandler sind Prozeduren des Betriebssystems, deren Ausführung durch zur CPU externe Ereignisse angestoßen wird. Die Ausführung eines Interrupthandlers kann wiederum von Interrupts unterbrochen werden; daher muß ein Interrupthandler auch reentrant programmiert werden (es sei denn, er unterbindet den Aufruf von Interrupthandlern).

Bottom Halves Aktionen, die aufgrund eines Interrupts ausgeführt werden, aber nicht so dringend sind, daß sie unmittelbar im Interrupthandler ausgeführt werden müssen, werden in Bottom Halves untergebracht. Die 32 Bottom Halves unter Linux sind statisch vergeben. Wenn man als Autor eines modularisierten Treibers eine Prozedur als Bottom Half ausführen lassen will, muß man sich der Task-Queue „tq_immediate" bedienen. In einem Interrupthandler können beliebige Bottom Halves markiert werden. Die Ausführung aller markierten Bottom Halves erfolgt bei jedem Scheduling-Vorgang und bei der Rückkehr von einem Systemruf. Die Markierung wird bei der Ausführung wieder gelöscht. Bottom Halves müssen nicht reentrant sein.
Mit der Kernelversion 2.3.43 werden die Bottom Halves durch SoftIRQs abgelöst. SoftIRQs unterscheiden sich dadurch von den Bottom Halves, daß einzelne SoftIRQs auf verschiedenen CPUs gleichzeitig laufen dürfen.

Task Queues Hier kann man dynamisch Prozeduren registrieren, die zu bestimmten Zeitpunkten ausgeführt werden sollen. Da die Ausführung der Task-Queues im Rahmen der Abarbeitung der Bottom Halves stattfindet, gilt das dort Angeführte. Task Queues, die im Kernel zur Verfügung stehen:

tq_immediate	Wird bei jeder Ausführung der Bottom Halves abgearbeitet.
tq_timer	Wird nach jedem Timerinterrupt abgearbeitet. (100 mal pro Sekunde auf i386, 1000 mal pro Sekunde auf Alpha)
tq_scheduler	Wird bei jedem Aufruf des Schedulers abgearbeitet.

Prozeß Ein Prozeß kann entweder seinen Programmcode, oder aber auch den Code des Betriebssystems ausführen. Im ersten Fall kann die Ausführung jederzeit unterbrochen und mit einem anderen Prozeß fortgesetzt werden. Dies ist eine ganz normale Eigenschaft eines Betriebssystems, das präemptives Multitasking hat.
Führt der Prozeß gerade einen Teil des Kernels aus, befindet er sich also im Kernelkontext, kann er zwar durch einen Interrupt kurzfristig unterbrochen werden; danach wird er allerdings weiter ausgeführt. D.h. im Kernelkontext ist Linux zum heutigen Zeitpunkt nicht präemtiv. Im Laufe der 2.3.x Entwicklung wurden die Möglichkeiten diskutiert, auch den Kernelkontext präemptiv zu gestalten, es ist aber unwahrscheinlich, daß das bereits bei Linux 2.4.x der Fall sein wird.

4.3.2 Synchronisationsprimitive

Die grundlegenden Mechanismen, aus denen die höheren Synchronisationsobjekte aufgebaut sind:

cti()/sti() Sperren und Freigeben aller Interrupts. Es gibt noch weitere Varianten dieser Kommandos, die aber keine grundsätzlich andere Funktion zur Verfügung stellen.

Spinlocks Ein Spinlock ist ein Synchronisationsobjekt, das nur einmal belegt werden kann. Ist es bereits belegt, wird so lange eine Warteschleife ausgeführt, bis der Spinlock wieder frei ist. Ein Spinlock kann nur für die Synchronisation zwischen zwei CPUs verwendet werden. Falls ein Kernel für einen Computer mit nur einem Prozessor erzeugt wird, werden Spinlocks bei der Übersetzung des Quellcodes eliminiert.

Wait Queues Eine Wait-Queue bietet zwei Funktionen: Ein Prozeß kann mit sleep_on() auf ein Ereignis warten. Dabei wird der Scheduler aufgerufen, und der Prozeß gibt die CPU ab. Tritt nun das Ereignis ein, so kann man alle Prozesse, die in einer Wait-Queue warten, wecken. Für den geweckten Prozeß bedeutet das, daß der sleep_on() Aufruf zurückkehrt.
Sleep_on() gibt es in zwei Varianten; eine läßt den Prozeß im Zustand TASK_UNINTERRUPTIBLE warten, während sleep_on_interruptible() den Prozeß im Zustand TASK_INTERRUPTIBLE warten läßt. Ein Prozeß im ersten der beiden Zustände reagiert auf keine Signale und kann daher auch nicht mit dem Kill-Kommando aus dem System entfernt werden.

Darauf aufbauend gibt es Semaphore und in neueren Versionen (2.3.x) auch eine Schreib- und Lesevariante von Semaphoren. Semaphore könne zur Synchronisation von Prozessen im Kernelkontext eingesetzt werden. Semaphore können natürlich nicht in Interrupthandlern und Bottom Halves eingesetzt werden, da diese außerhalb eines Prozeßkontextes ausgeführt werden und Semaphore einen Prozeßkontext brauchen.

Spinlocks können ebenfalls für die Synchronisation zwischen Prozessen im Kernelkontext eingesetzt werden. Spinlocks gibt es bereits in den 2.2.x Versionen des Kernels in Schreib- und Lesevarianten. Sie eignen sich gut für Synchronisationsaufgaben, bei denen die Ressource nur für kurze Zeit geschützt werden muß. Während man einen Spinlock besetzt hat, darf man auf keinen Fall den Scheduler aufrufen.

Für die Synchronisation zwischen Interrupts und Prozessen im Kernelkontext steht eine Variante der Spinlocks zur Verfügung, die die Ausführung von Interrupts auf der lokalen CPU verhindert. Falls es sich um die Schreib- und Lesevariante der Spinlocks handelt und man innerhalb des Interrupthandlers nur lesend auf die Ressource zugreifen will, kann man dort die normale Variante des Spinlocks verwenden.

> Innerhalb des Interrupthandlers muß im Normalfall ebenfalls die _irqsave() Variante der Spinlock-Operationen verwendet werden, denn wenn der Interrupt während der Abarbeitung des Interrupthandlers ein weiteres Mal auftritt[2], wird diese unterbrochen, der Interrupthandler nocheinmal aufgerufen und der erste Aufruf danach wieder fortgesetzt. Im Prozeßkontext ist die _irqsave() Variante der Spinlock-Operationen notwendig, um die Atomizität zu gewährleisten, im Kontext des Interrupthandlers wird dadurch auch noch ein Deadlock verhindert.

[2] Das ist natürlich nur mit flankengetriggerten Interrupts möglich.

Handelt es sich nun um einen Scheib- und Lese-Spinlock und ist im Interrupthandler nur lesender Zugriff erforderlich, kann innerhalb des Interrupts die normale Variante der Operationen verwendet werden. Da mehrere Leser auf einmal erlaubt sind, kann es daher bei einer zweiten, gleichzeitigen Ausführung des Interrupthandlers auf der gleichen CPU nicht zu diesem Deadlock kommen.

4.4 Threads

In vielen anderen Betriebssystemen wurde mit der Einführung von Threads der Prozeß derart verändert, daß er nur mehr eine Umgebung ist, in der eine beliebige Zahl von Threads laufen kann. Der Prozeß stellt dabei keine Einheit für den Scheduler dar, ist aber noch eine Einheit für viele andere Subsysteme des Betriebssystems, wie z.B. für die Speicherverwaltung und Filesysteme.

Da dieser Übergang eine einschneidende Veränderung für ein Betriebssystem darstellt, wurde unter Linux ein anderer Ansatz gewählt. Dabei wurde der Speicherkontext und der Dateisystemkontext aus dem Prozeß herausgelöst, sodaß nun Prozesse diese gemeinsam benutzen können. Dies spiegelt sich auch in der Implementierung wieder; das Interface der Systemrufe hat sich nur dadurch verändert, daß eine verallgemeinerte Form des fork(2) Systemrufes hinzugekommen ist.

Mit Hilfe dieser Kernelerweiterung wurde eine POSIX-konforme Threading Library geschaffen. Diese stellt den Applikationsprogrammen mit einer kleinen Ausnahme[3] das POSIX-konforme Verhalten, in dem Threads Bestandteile eines Prozesses sind, zur Verfügung.

4.4.1 Kernel-Threads

Diese Auskoppelung des Speicherkontextes und des Dateisystemkontextes ermöglichte auch die problemlose Einführung von Kernel-Threads. Diese sind Prozesse, die innerhalb des Kernels erzeugt werden können, um dort bestimmte Aufgaben zu übernehmen. Da sie nie Programmcode im Userspace ausführen, haben sie auch keinen Speicherkontext.

[3] Während laut POSIX Signale, die an einen Prozeß adressiert sind, an alle Threads geliefert werden, wird unter Linux ein Signal nur an jenen Thread geliefert, an den es adressiert wurde.

4.5 Module

Vor der Version 2.0 des Linux-Kernels konnten Gerätetreiber nur statisch in den Kernel eincompiliert werden. Damals war es üblich, nach der erfolgreichen Installation des Linux-Systems, den Kernel selbst neu zu compilieren, und dabei nur jene Gerätetreiber in den Kernel einzubinden, die man auch tatsächlich brauchte.

Nicht nur für den Anwender war diese Situation nicht zufriedenstellend, auch war damals die Entwicklung eines neuen Gerätetreibers sehr umständlich, denn die einzige Möglichkeit, den Treiber neu zu starten, war es, das Zielsystem mit dem neuen Kernel neu zu starten.

Mit Version 2.0 wurde der Kernel um einen Loader erweitert, der es erlaubt, in den laufenden Kernel Module einzufügen bzw. zu entfernen.

Die Schnittstelle, der sich ein Autor eines Moduls bedienen kann, ist aber kein Framework, wie das bei Treiberschnittstellen vieler Betriebssysteme der Fall ist. Es stehen eine Menge Prozeduren und Makros zur Verfügung, die neben den Funktionen, die speziell für Module zur Verfügung stehen (z.B. register_filesystem(), restister_blkdev(), usw.), auch eine gute Auswahl der allgemeinen Funktionen des Kernels aufweisen.

Diese Kernelschnittstelle ist zwischen verschiedenen Versionen des Kernels nicht binärkompatibel, man kann jeoch Module anderer Kernelversionen laden und wird zumindest darauf aufmerksam gemacht, daß dieser Vorgang zu einem Systemabsturz führen kann.

Doch beim Versuch, ein Modul, das für eine Ein-Prozessor-Maschine übersetzt wurde, in einen Mehr-Prozessor-Kernel zu laden, wird man leider nicht gewarnt. Sowohl diese, als auch die umgekehrte Kombination (SMP-Modul in UP-Kernel) ist leider sehr instabil.

Kapitel 5

Implementierung

Es gibt zwar ein paar wenige, jedoch in der Regeln nicht aktuelle, Unterlagen, die einem bei der Implementierung eines Treibers für Linux behilflich sind. Nun wurde endlich eine Neuauflage des diesbezüglich besten Buches [Rub98] angekündigt, die den Linux-Kernel in der Version 2.2.x abdecken soll. Es wird sich allerdings erst herausstellen, ob diese zweite Auflage wirklich erhältlich ist, bevor die Version 2.4.x des Linux-Kernels fertig ist.

Auch ist das Framework, in das sich ein Treiber eingliedern muß, verständlicherweise auf Treiber ausgerichtet, die richtige Hardware ansprechen, was nicht zuletzt zu einigen Problemen bei der Implementierung dieses virtuellen Massenspeichergerätes führte.

Dieses Kapitel bezieht sich auf Version 0.5.3 von DRBD. Alle veröffentlichten Versionen von DRBD sind unter http://www.complang.tuwien.ac.at/reisner/drbd/ erhältlich.

5.1 Struktur

Den Hauptteil des Programmcodes, knapp 2500 Zeilen C, bildet ein Modul für den Kernel, das ein Blockgerät implementiert. Da für die Konfiguration Information vom Anwender erforderlich ist und DRBD auch auf externe Ereignisse reagieren muß, kann das Kernelmodul, das aus dem Userspace nur als Blockgerät sichtbar ist, über den ioctl() Systemruf Informationen entgegennehmen.

Die für den Anwender greifbare Seite der ioctl() Systemrufe ist im drbdsetup (rund 350 Zeilen) Utility implementiert. Es wird unter anderem dazu verwendet, die notwendigen IP-Adressen an das Kernelmodul weiterzugeben. Es wurde allerdings so ausgelegt, daß sich kein für das Funktionieren des Gerätes notwendiger Programmteil im drbdsetup Utility befindet.

Als Kontrast dazu sei das nbd-client Utility von NBD[1] erwähnt. Nbd-client baut die TCP/IP-Verbindung zum Server, der im Falle von NBD ebenfalls ein normales Anwendungsprogramm ist, auf und übergibt die fertige Verbindung an den Kernel-Code.

Da drbdsetup nur Information weitergibt, wird es einfacher sein, Root-Filesystem-Unterstützung in DRBD zu implementieren, wenn DRBD in den Linux-Kernel eingebunden wird.

Daneben gibt es noch Shell-Skripts, die die Integration von DRBD mit der Cluster-Management-Software Heartbeat ermöglichen, sowie eine Benchmark-Applikation.

5.2 Kernel-Modul

Als Treiber muß DRBD natürlich eine Instanz der file_operations im System registrieren, und DRBD muß eine do_request Prozedur dem Buffer-Cache zur Verfügung stellen. Weiters registriert DRBD einen neuen Eintrag im /proc Filesystem, über den sich der Administrator ein Bild über die internen Zustände der DRBD-Geräte verschaffen kann.

5.2.1 Buffer-Cache

Wenn der Buffer-Cache Anforderungen durch einen Aufruf der do_request Prozedur übergibt, müssen diese möglichst schnell erfüllt werden. Handelt es sich um eine Aufforderung zum Lesen eines Blockes, muß DRBD den Datentransfer von der zugrundeliegenden Festplatte durchführen. Dies wird mit Hilfe einer temporären Kopie des buffer_heads erledigt. Diese Methode wird auch beim ext3 Dateisystem verwendet. [Twe98]

Durch dieses Verhalten befinden sind immer nur Blöcke des DRBD-Gerätes im Buffer-Cache, Blöcke des physischen Festplattengerätes liegen nie im Buffer-Cache.

Bei der Verarbeitung einer Schreibanforderung muß der Datenblock allerdings auch über das Netzwerk versendet werden. Da die do_request() Prozedur immer im Kontext eines Prozesses aufgerufen[2] wird (vgl. Abschnitt 4.3.1 auf Seite 35), ist das Versenden des Datenpaketes über die TCP-Verbindung ohne weiteres möglich. Sollte der Pufferspeicher des Sockets voll werden, blockiert der Aufruf von sock_sendmsg(), und der Prozeß, der die Schreiboperation verursacht hat, wird blockiert. Diese Eigenschaft ist auch sehr

[1] Network block device, http://atrey.karlin.mff.cuni.cz/~pavel/nbd/nbd.html

[2] Entweder im Kontext eines Prozesses, der eine IO-Operation durchführt, oder im Kontext des kflushd oder kupdate Kernel-Threads.

hilfreich, wenn das Computersystem, das die Datenblöcke empfängt, langsamer ist als das sendende System, da in diesem Fall ebenfalls der Prozeß am sendenden System so lange blockiert wird, bis das empfangende System wieder bereit ist, Daten entgegenzunehmen.

5.2.1.1 Deadlock

Da Linux eine fixe Anzahl an Einträgen für IO-Anforderungen hat (siehe 4.2.2.2 auf Seite 33), kann es zu einem Deadlock kommen, wenn die Request-Queue keine Schreibanforderungen mehr aufnehmen kann.

Kann die Request-Queue keine Schreibanforderungen mehr aufnehmen[3], wird der Prozeß, dessen Schreibanforderungen nicht mehr Platz haben, einfach so lange in den Zustand TASK_UNINTERRUPTIBLE gebracht, bis wieder Einträge frei sind. Dieses Schema funktioniert mit normalen Festplattentreibern ohne Probleme.

Wenn nun alle Einträge mit Schreibanforderungen an DRBD belegt sind, wird der Prozeß, der versucht, die Anforderungen abzuarbeiten, beim Versuch, eine Schreibanforderung an den darunterliegenden Festplattentreiber zu senden, blockiert. Da aber keine Schreibanforderung abgeschlossen werden kann, ist der Computer derart blockiert, daß er auch nicht mehr heruntergefahren werden kann.

Das loop-block-device, das schon sehr lange Bestandteil des Kernels ist, hat im Grunde dasselbe Problem. Es wurde beim loop-block-device dadurch gelöst, daß Schreibanforderungen des loop-block-devices und des NBDs nur ein Drittel der Request-Queue belegen dürfen. Es handelt sich um einen Sonderfall, der in linux/drivers/block/ll_rw_block.c fest verankert wurde.

DRBD ist bis jetzt als Modul implementiert, da dies die Installation wesentlich vereinfacht. Daher kann DRBD keinen eigenen Sonderfall in linux/drivers/block/ll_rw_block.c haben. Ich habe mich dazu entschlossen, bis auf weiteres die Major Nummer des network-block-devices zu verwenden. Dadurch können Schreibanforderungen an DRBD nur ein Drittel der Request-Queue belegen, und es kann nicht zu dem erwähnten Deadlock kommen.

Dadurch entsteht der Nachteil, daß NBD und DRBD nicht gleichzeitig verwendet werden können. Der Vorteil liegt in einer wesentlich einfacheren Installation, da die Implementierung als Modul möglich ist.

Mit Linux 2.4 wird diese Einschränkung aufgehoben werden können, weil dann DRBD seine eigene Request-Queue haben kann und die Verwendung einer eigenen Major Nummer wieder möglich ist.

[3] Nur 66 % der Request-Queue können mit Schreibanforderungen gefüllt werden, das letzte Drittel steht immer für Leseanforderungen zur Verfügung.

5.2.2 Threads

Am Anfang der Entwicklung wurde vorgesehen, daß der Empfang von Datenpaketen durch einen normalen Prozeß im Userspace vorgenommen werden sollte. Diese Vorgangsweise hätte neben schlechterer Performance und komplizierterer Administration vor allem mit folgendem Problem zu kämpfen:

> Im folgenden wird ein Knoten eines Clusters bestehend aus zwei Computern betrachtet.
>
> 1. Das Gerät des Knotens befindet sich im primären Zustand, daher kann eine Applikation auf das Blockgerät zugreifen. Bei der Durchführung von Schreiboperationen werden Kopien der Blöcke im Buffer-Cache gespeichert. Diese Blöcke tragen die Major Number des DRBD-Gerätes.
> 2. Der Zustand des Clusters wird geändert, und das betrachtete Blockgerät wird von nun an in der sekundären Funktion verwendet.
> 3. Datenblöcke werden vom Prozeß im Userspace empfangen und auf die dazugehörige Festplatte geschrieben. Dabei werden sie im Buffer-Cache mit der Major Number des Festplattentreibers gespeichert.
> 4. Nun wird der Cluster wieder in den ursprünglichen Zustand gebracht.
> 5. Wird nun vom DRBD-Gerät gelesen, liefert der Buffer-Cache die Blöcke aus Punkt 1, obwohl sie in Punkt 3 durch neue Blöcke überschrieben wurden.

5.2.2.1 Empfang

Um nicht auf dieses Problem zu stoßen, wurde das Empfangen von Datenblöcken zu einem integralen Bestandteil von DRBD, der im Modul als Kernel-Thread implementiert ist.

Dieser Thread wartet in sock_recvmsg(), bis ein neuer Datenblock vom Netzwerk entgegengenommen werden kann. Danach wird vom Buffer-Cache Pufferspeicher für den entsprechenden Block angefordert, und die Daten werden direkt[4] in diesem Speicher empfangen. Bei der Lösung mit dem Daemon-Prozeß im Userspace hätten die Daten zuerst in den Adreßraum des Daemons kopiert werden müssen, um dann von dort wiederum in den Buffer-Cache kopiert zu werden. Danach wird der Befehl gegeben, den Block im Buffer-Cache auf die Festplatte zu schreiben.

[4] Wenn die Netzwerkkarte Daten empfängt, werden diese zuerst im Pufferspeicher des Sockets abgelegt und von dort dann mit sock_recvmsg() kopiert.

Sollte DRBD einmal erweitert werden, um den Einsatz von GFS über DRBD zu ermöglichen, so muß unter anderem hier eine Änderung vorgenommen werden. DRBD müßte beim Empfang von Datenblöcken direkt auf die physische Festplatte schreiben und dürfte den Block nicht zuerst im Buffer-Cache unterbringen, da GFS ebenfalls auf diese Blöcke des Buffer-Caches zugreifen würde. Ohne diese Änderung würde auch jeder Datenblock endlos hin und her gesendet werden.

Der Thread ist, wenn das DRBD-Gerät mit einem Partnergerät verbunden wurde, mit dem ps-Kommando sichtbar und trägt z.B. den Namen „drbdd_0", wobei die Ziffer immer die Minor Number des Gerätes anzeigt, zu dem der Thread gehört.

5.2.2.2 Senden

Während das Senden der Daten im Kontext jenes Prozesses durchgeführt wird, der den Aufruf der do_request() Prozedur verursacht hat, gibt es auch Ereignisse, die das Senden eines Paketes auslösen, die allerdings nicht innerhalb eines Prozeßkontextes auftreten. Beispiele:

- Der Treiber der Festplatte meldet, daß eine Schreiboperation abgeschlossen ist. Wenn nun dieser Block in der momentanen Menge für die Bestimmung der Write-Barriers enthalten ist, so muß ein Write-Barrier-Paket gesendet werden.
- Der Treiber der Festplatte meldet, daß eine Schreiboperation abgeschlossen ist. Wenn dieser Block von einer Übertragung mit Protokoll C (oder dem Resynchronisationsprozeß) stammt, muß ein Write-Ack-Paket gesendet werden.

Festplattentreiber rufen diesen Callback, über den sie diese Ereignisse signalisieren, normalerweise aus einem Bottom-Half aus auf, der an den Interrupt, den die Festplatte beim Abschluß der Operation erzeugt, gekoppelt ist. Ein Bottom-Half hat keinen Prozeßkontext, daher können auch keine Prozeduren, die eventuell schlafen, aufgerufen werden.

Um die notwendigen Pakete doch versenden zu können, gibt es einen zweiten Kernel-Thread. Dieser Thread wartet in der asender_wait Wait-Queue, bis er von einem Bottom-Half aus geweckt wird. Um welches Ereignis es sich handelt und welche Aktion daher zu setzen ist, stellt dieser Thread über Variablen fest, die mit den anderen Programmteilen gemeinsam verwendet werden.

Diese Threads sind im System unter der Bezeichnung „drbd_asender_X" zu finden, wobei das X ebenfalls die Minor Nummer des Gerätes wiedergibt.

5.2.2.3 Synchronisation

Falls neben der automatisch durchgeführten Replikation der Datenblöcke eine Synchronisation wie in Abschnitt 3.4 auf Seite 26 beschrieben durchgeführt wird, so wird dies durch einen eigenen Kernel-Thread erledigt. Dieser Thread läuft nur während des Synchronisationsvorganges auf jenem Knoten, von dem aus die Daten versendet werden.

Bei der üblichen Vorgangsweise, um Daten von einem Blockgerät zu bekommen, wird zuerst der Block vom Buffer-Cache angefordert und — falls dieser nicht aktuell ist — werden die Daten vom Gerät mit ll_rw_block() eingelesen.

Eine der Anforderungen an den Synchronisationsmechanismus ist, daß er parallel zum Normalbetrieb laufen muß. Im normalen Betrieb arbeitet allerdings ein Dateisystem aktiv mit dem Buffer-Cache. Es kann z.B. Blöcke anfordern, sie einlesen, Modifikationen durchführen, sie freigeben[5] oder explizit das Zurückschreiben verlangen.

Würde der Kernel-Thread, der die Synchronisation durchführt, die übliche Vorgangsweise beim Einlesen von Daten anwenden, würde der Buffer-Cache einfach die Adressen der Blöcke im Speicher auch an das DRBD-Gerät bekanntgeben, und es könnten Daten auf die gespiegelte Festplatte geschrieben werden, die das Filesystem noch gar nicht schreiben will[6].

Daher muß der Buffer-Cache beim Einlesen der Daten für die Synchronisation umgangen werden. Dabei wird, ähnlich wie in 5.2.1 mit einem selbst erzeugten buffer_head gearbeitet.

Die Implementierung des Bitfeldes wurde vom Rest durch eine Sprungtabelle komplett getrennt. Das ermöglicht, daß das Bitfeld durch andere Implementierungen ersetzt wird. So könnte eine Implementierung, die mit einem Filesystem integriert ist, Informationen aus dem Filesystem heranziehen, um den Synchronisationsvorgang besser zu steuern.

5.2.3 Datenstrukturen

Wie bereits erwähnt, greifen die Threads auf einige gemeinsame Ressourcen zu. Für Datenstrukturen wurden ausschließlich Spinlocks verwendet. Da die Zugriffe auf diese Datenstrukturen sehr kurz sind, wären Semaphore zu aufwendig.

[5]Ein Block, der modifiziert und freigegeben wurde, wird später durch kflushd oder kupdate zurückgeschrieben.

[6]Da genau dieser Fehler vom Kernel-Thread für die Synchronisation des Software-Raid-Subsystems unter Linux gemacht wird, ist momentan (Linux 2.2.x) die Verwendung von JFSs auf Software-Raid unter Linux nicht möglich.

transfer_log Im transfer_log, das als Ringpufferspeicher organisiert ist, werden Referenzen auf die gesendeten Blöcke gespeichert. Ebenso werden gesendete write-barriers im transfer_log abgelegt. Wird ein Bestätigungspaket für ein Write-Barrier-Paket empfangen, können alle Einträge im transfer_log, die vor der dazugehörigen Write-Barrier liegen, freigegeben werden. Die Menge für die Bestimmung der Write-Barriers sind all jene Einträge, die nach der jüngsten Write-Barrier ins transfer_log eingetragen wurden.
Auf das transfer_log wird somit aus der do_request() Prozedur, vom Empfangs-Thread, vom Sende-Thread und vom Kontext eines Bottom-Half aus zugegriffen. Um auch die Atomizität gegenüber dem Bottom-Half gewährleisten zu können, muß ein Spinlock mit den _irqsave() / _irqrestore() Operationen eingesetzt werden.

epoch_set Das epoch_set wird beim Empfang von Datenblöcken verwendet, um dort Referenzen auf die empfangenen Blöcke abzulegen. Wird ein Write-Barrier-Paket empfangen, muß auf die Beendigung der IO-Operationen aller im epoch_set enthaltenen Blöcke gewartet, das epoch_set geleert und anschließend die Bestätigung der Write-Barrier an den Absender zurückgeschickt werden. Der Sende-Thread muß auf das epoch_set zugreifen, um festzustellen, für welche Blöcke eine Schreibbestätigung gesendet werden kann.
Das epoch_set muß daher vom Empfangs-Thread und vom Sende-Thread verwendet werden. Ein Spinlock ist für die Synchronisation der Zugriffe ausreichend.

sync_log Das sync_log ist ebenfalls eine Datenstruktur, die auf der Empfängerseite verwendet wird. Da Datenblöcke, die vom Synchronisations-Thread stammen, nicht am Write-Barrier Protokoll teilnehmen, werden Referenzen auf diese Blöcke hier gespeichert, bis deren Schreibbestätigung an den Absender zurückgeschickt werden kann.
Das sync_log wird ebenfalls vom Empfangs-Thread und vom Sende-Thread verwendet, daher ist ein normaler Spinlock ausreichend.

BitMap Das vom schnellen Synchronisationsmechanismus verwendete Bitfeld (siehe Punkt 3.4.1 auf Seite 26) muß vor gleichzeitigen Zugriffen aus mehreren Prozeßkontexten geschützt werden, daher wird ein Spinlock verwendet.

need_to_issue_barrier Von dieser Variable wird nur ein einziges Bit verwendet. Ist dieses Bit gesetzt, wird vor dem Senden des nächsten Datenpaketes ein Write-Barrier-Paket verschickt. Doch der Sende-Thread versucht ebenfalls, wenn dieses Bit gesetzt ist, das Write-Barrier-Paket zu verschicken. Im Quellcode sind diese Read-Modify-Write-Zugriffe auf dieses Bit durch atomare Operationen ausgeführt,

damit ein need_to_issue_barrier Bit nicht zweimal verschickt werden kann.
Doch dies ist nur mehr zu Dokumentationszwecken im Quelltext vorhanden, denn auf das Bit darf erst nach Erhalt der send_mutex zugegriffen werden, da man das Paket sowieso senden muß und diese Aktion an der richtigen Stelle im transfer_log eingetragen werden muß.

request.rq_status Da bei einer Scheibanforderung auf den Abschluß von zwei Operationen gewartet werden muß, wird im Statusfeld des Requests festgehalten, welche Operation bereits abgeschlossen ist. Die Prozedur, die dieses Feld bearbeitet, liest zuerst den Wert ein und schreibt ihn anschließend zurück. Weil es zumindest einen Festplattentreiber, der diese Prozedur (über den Callback, der den Abschluß einer IO-Operation signalisiert,) aus dem Kontext eines Interrupts aufruft, gibt, müssen der Lesezugriff und der Schreibzugriff mit einem Spinlock zusammengefaßt werden.

Socket Da aus mehreren Prozeßkontexten[7] und dem Sende-Thread über den Socket gesendet wird, muß die Sendeoperation mit einem Semaphor gesichert werden, da sonst ein zufälliges Gemisch der Datenblöcke entsteht.
Dieses Semaphor, mit dem Namen send_mutex, stellt ebenfalls sicher, daß die Eintragungen in das transfer_log in genau derselben Reihenfolge erfolgen, in der auch die Datenblöcke versendet werden.

5.2.4 Protokoll

Da der Linux-Kernel auf einer Vielzahl von Hardwareplattformen läuft, wurde vom Anfang an darauf geachtet, daß DRBD so implementiert ist, daß DRBD-Geräte, die auf unterschiedlichen Hardwareplattformen laufen, miteinander kommunizieren können. Daher wird für die Daten in den Paketen die Big-Endian-Byteorder verwendet.

Allen Paketen ist folgender Header gemeinsam:

```
typedef struct {
__u32 magic;
__u16 command;
__u16 length;
} Drbd_Packet;
```

Die verwendeten Datentypen sind in den Include-Dateien von Linux definiert, wobei u16 bedeutet, daß es sich um ein unsigned int mit 16 Bit handelt.

[7] Es könnten beliebig viele Prozesse das Blockgerät gleichzeit öffnen, schreiben, fsync(2) oder fdatasync(2) aufrufen.

magic Es handelt sich dabei um eine Konstante. Auf der Empfangsseite wird diese Zahl immer auf ihre Korrektheit überprüft. Ist sie einmal nicht korrekt vorhanden, wird davon ausgegangen, daß ein Fehler im Programm vorliegt, und die Verbindung wird unterbrochen.

command Dieses Feld dient dazu, die Funktion der folgenden Daten, also den Typ des gesamten Paketes, festzulegen. Folgende Werte sind möglich: Data, Barrier, RecvAck, WriteAck, BarrierAck, ReportParams, BlkSizeChanged und CStateChanged.

length Gibt die Länge des Datenteils des Paketes an.

5.2.4.1 Datenpaket

```
typedef struct {
__u64 block_nr;
__u64 block_id;
} Drbd_Data_P;
```

Datenpakete beinhalten die eigentliche Nutzinformation. Die Daten selbst folgen diesem Header.

block_nr Die Blocknummer des Datenblockes.

block_id Mit Hilfe dieser 64 Bits wird einerseits unterschieden, ob es sich um einen Datenblock, der am Write-Barrier-Protokoll teilnimmt, oder einen Block des Synchronisationsprozesses handelt. Der Inhalt dieses Feldes wird auch bei einem Bestätigungspaket wieder zurückgeschickt. Bei Protokoll C legt hier der sendende Knoten die Adresse der IO-Aufforderung ab. Dadurch kann beim Empfang des zugehörigen Bestätigungspaketes, das dieses Feld ebenfalls hat, der Abschluß der IO-Aufforderung schneller durchgeführt werden.

5.2.4.2 Write-Barrier-Paket

```
typedef struct {
__u32 barrier;
__u32 _fill;
} Drbd_Barrier_P;
```

Über dieses Paket darf kein Datenblock verschoben werden. Auf dem Netzwerk ist dies ohnehin nicht möglich, da TCP die Datenpakete in der richtigen Reihenfolge überträgt. Doch im Buffer-Cache muß dies erst mit entsprechenden Maßnahmen sichergestellt werden.

barrier Hierbei handelt es sich um eine Nummer zur Identifizierung dieser Write-Barrier. Sie ist für die prinzipielle Funktion nicht notwendig, wird aber dennoch verwendet, um die korrekte Funktion des Protokoll ständig überprüfen zu können.

_fill Da gcc auf 64-Bit-Plattformen die Größe des Datentypes auf ein Vielfaches von 8 Byte auffüllt, ist dieses Feld notwendig, damit der Datentyp auch auf 32-Bit-Plattformen die gleiche Größe hat.

5.2.4.3 Parameterpaket

```
typedef struct {
__u64 size;
__u32 state;
__u32 blksize;
__u32 protocol;
__u32 version;
} Drbd_Parameter_P;
```

Dieses Paket wird bei den Kommandos ReportParams und BlkSizeChanged gesendet. Es wird unmittelbar nach einem erfolgreichen Verbindungsaufbau ausgetauscht. Stimmen das Protokoll oder die Version der beiden Kommunikationspartner nicht überein, wird die Verbindung sofort unterbrochen.

size In diesem Feld wird die Größe der lokalen Festplatte übermittelt.

state Hier wird der Zustand des DRBD-Gerätes übermittelt, entweder primär oder sekundär. Treffen zwei DRBD-Geräte, die sich im primären Zustand befinden, aufeinander, wird die Verbindung unterbrochen.

blksize Die momentane Blockgröße.

protocol Das Protokoll, mit dem dieses DRBD-Gerät die Verbindung betreiben will.

version Die Version der DRBD-Implementierung.

5.2.4.4 Bestätigungspaket

```
typedef struct {
__u64 block_nr;
__u64 block_id;
} Drbd_BlockAck_P;
```

Dieses Paket wird mit dem Kommando RecvAck im Protokoll B und mit WriteAck im Protokoll C verwendet. Die beiden Datenfelder werden dabei aus dem Datenpaket, auf das sich diese Bestätigung bezieht, übernommen.

5.2.4.5 Write-Barrier-Bestätigungspaket

```
typedef struct {
__u32 barrier;
__u32 set_size;
} Drbd_BarrierAck_P;
```

Wenn alle Blöcke, die vor einer Write-Barrier empfangen wurden, auch geschrieben sind, wird dieses Paket versendet. Beide Datenfelder dieses Paketes tragen nicht zur prinzipiellen Funktion bei, werden aber für interne Konsistenzüberprüfungen verwendet.

barrier Die Nummer, die beim Erzeugen dieser Write-Barrier generiert wurde.

set_size Gibt an, wie viele Blöcke vor dieser Write-Barrier empfangen wurden.

5.2.4.6 C-Zustandspaket

```
typedef struct {
__u32 cstate;
} Drbd_CState_P;
```

Die einzige Verwendung dieses Paketes besteht darin, dem Kommunikationspartner den Anfang und das Ende des Synchronisationsvorganges mitzuteilen.

cstate Folgende Werte sind möglich:

SyncingAll Jetzt werden alle Blöcke kopiert.

SyncingQuick Jetzt beginnt ein schneller Synchronisationsvorgang.

Connected Der Synchronisationsvorgang ist beendet.

5.2.4.7 Timeouts

Da aus dem Kontext eines schreibenden Prozesses gesendet wird, wird dieser blockiert, falls das Senden über den Socket blockiert. Dieser Fall tritt auch dann ein, wenn das Netzwerk zwischen den verbundenen DRBD-Geräten unterbrochen wurde.

Damit eine Applikation nicht angehalten wird, gibt es eine obere Schranke für die Dauer der Sendeoperation. Vor dem Aufruf von sock_sendmsg() wird ein

Timer gestartet. Läuft dieser Timer ab bevor sock_sendmsg() zurückkehrt, wird der Systemruf mit Hilfe eines Signals abgebrochen. Der Prozeß sieht dieses Signal nicht, da es noch innerhalb des Treibers wieder gelöscht wird.

Wenn ein Timeout aufgetreten ist, wird der Socket geschlossen, und DRBD setzt den Betrieb im nicht verbundenen Zustand fort. Alle weiteren Schreiboperationen werden im Bitfeld für spätere Synchronisation vermerkt.

Im Falle von Protokoll B und C muß auch für alle im transfer_log enthaltenen Blöcke der Abschluß der IO-Operation signalisiert werden, da nach dem Schließen des Sockets keine Bestätigungspakete mehr empfangen werden können.

Ebenfalls nur für Protokoll B und C muß die Zeit, die auf das Eintreffen von Bestätigungspaketen gewartet wird, begrenzt werden. Die Notwendigkeit dafür kann am folgenden Beispiel erläutert werden:

> Die Request-Queue kann keine weiteren Schreibanforderungen mehr aufnehmen und blockiert daher alle Prozesse, die weitere Schreibanforderungen erzeugen wollen. Während nun auf den Abschluß der laufenden IO-Operationen gewartet wird, wird das Netzwerk unterbrochen. Da allerdings keine neuen IO-Operationen erzeugt werden, wird auch nichts über die TCP-Verbindung gesendet, und die Unterbrechung der Verbindung bleibt unentdeckt. Das IO-Subsystem bleibt allerdings blockiert, da die Bestätigungspakete, die die aktuellen IO-Operationen abschließen würden, nicht mehr eintreffen können.

Für die Implementierung des Timeouts beim Senden ist genau ein Timer pro DRBD-Gerät notwendig, da immer nur genau ein Prozeß senden kann.

Das Timeout für den Empfang von Bestätigungspaketen wurde ebenfalls mit Hilfe eines einzigen Timers implementiert. Dabei wird der Timer beim Senden eines Datenpaketes zurückgesetzt und ein Zähler (pending_cnt) erhöht. Wird ein Bestätigungspaket empfangen, wird der Zähler verringert und der Timer ebenfalls neu gestartet (falls der Zähler noch positiv ist). Durch diese Implementierung kann zwar eine einzelne Bestätigung länger dauern als der Wert des Timeouts, aber die im obigen Beispiel beschriebene Situation wird erkannt.

5.2.4.8 Prioritäten

Im folgendem werden zwei DRBD-Geräte-Paare auf einem aus zwei Knoten bestehenden Cluster betrachtet, bei denen die primären Geräte auf den jeweils anderen Knoten liegen (siehe Abbildung 5.1).

Kommt es bei einer solchen Konfiguration dazu, daß das Netzwerk langsamer ist als die Festplatten, kann es zu Timeouts und zum Abbruch der Verbindung kommen.

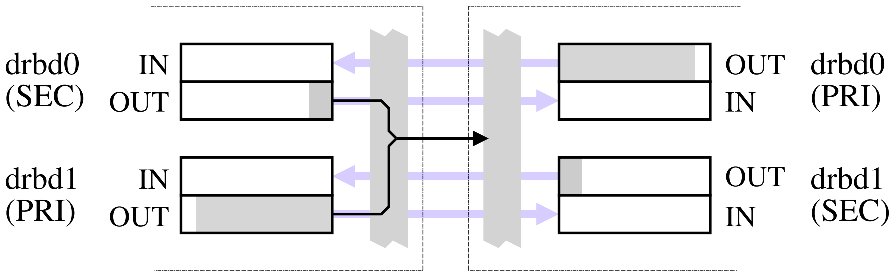


Abbildung 5.1: Zwei DRBD-Geräte-Paare

Bei hoher Last ist der Sendepufferspeicher des Sockets des Gerätes im primären Zustand normalerweise voll, während der Sendepufferspeicher des Gerätes im sekundären Zustand nur gering gefüllt ist, da Bestätigungspakete wesentlich kleiner sind als Datenpakete.

Das Netzwerksubsystem, in der Abbildung als grauer vertikaler abgerissener Balken dargestellt, arbeitet normalerweise mit einer einfachen FIFO-Strategie. Die beiden DRBD-Geräte auf einem Knoten stehen beim Senden in einer Konkurrenzsituation. Das primäre Gerät versendet wesentlich mehr und größere Pakete als das DRBD-Gerät im sekundären Zustand. Die Bestätigungspakete des sekundären DRBD-Gerätes werden daher vom Netzwerksubsystem länger verzögert.

Um diesem Problem entgegenzuwirken, wird in den Geräten im sekundären Zustand das TCP_NODELAY Flag des Sockets gesetzt, um den Nagle Algorithmus[8] auszuschalten. Abgesehen davon wird der Socket mit einer höheren Priorität versehen TC_PRIO_INTERACTIVE.

Dem Socket des primären Gerätes wird hingegen die Priorität TC_PRIO_BULK zugewiesen. Die Prioritätseinstellungen haben auf das Verhalten des Netzwerksubsystems allerdings nur dann Einfluß, wenn CONFIG_NET_SCHED bei der Compilierung des Kernels angegeben wurde. Diese Priorität wird auch im TOS-Feld der IP-Pakete weitergereicht und könnte von der Netzwerkinfrastruktur verwendet werden.

5.3 Drbdsetup

5.3.1 Konfiguration

Bevor ein DRBD-Gerät verwendet werden kann, muß es konfiguriert werden.

[8] Der Nagle Algorithmus verzögert kleine Pakete (<MTU) bis zu 200ms, um sie zu einem größeren zusammenzufassen.

5.3.1.1 Syntax

```
drbdsetup dev l_dev prot local_a[:port] remote_a[:port] [options]
```

dev Das DRBD-Gerät, z.B. /dev/nb0.

l_dev Das Gerät auf dem DRBD die Daten ablegen soll.

prot Das Protokoll, das dieses DRBD-Gerät verwenden soll.

local_a Die IP-Adresse des lokalen Netzwerkinterfaces.

remote_a Die IP-Adresse des zweiten Computers.

Diese Syntax steht zur Verfügung, um das DRBD-Gerät zu konfigurieren. Folgende Optionen können dabei angegeben werden:

--timeout -t Mit dieser Option kann eingestellt werden, wie lange gewartet wird, bis eine blockierte Verbindung abgebrochen wird. Die Standardeinstellung beträgt 3 Sekunden.

--sync-rate -r Diese Option erlaubt es, den Anteil der Bandbreite festzulegen, der maximal für den Synchronisationsprozeß zur Verfügung steht. Wird die Einstellung nicht vorgenommen, stehen 250KB/s zur Verfügung.

--skip-sync -k Unterbindet automatisches Starten des Synchronisationsprozesses.

--tl-size -s Hier kann die Größe des transfer_logs festgelegt werden. Netzwerkverbindungen mit hoher Bandbreite und langer Verzögerung könnten ein transfer_log von mehr als 256 Einträgen erfordern. Dies ist allerding nur notwendig, wenn sich im Systemlog Warnungen bezüglich eines zu kleinen transfer_logs befinden.

--disk-size -d Mit dieser Option kann die Größe des DRBD-Gerätes angegeben werden. Wenn diese Option nicht angegeben wird, nimmt das DRBD-Gerät die Größe der kleineren Festplatte an.
Diese Option macht es nun möglich, auf ein DRBD-Gerät zuzugreifen, bevor noch eine Verbindung zustandegekommen ist, da sonst das DRBD-Gerät die Größe 0 hat.

--do-panic -p Tritt bei einer IO-Operation auf der lokalen Festplatte ein Fehler auf, so wird dieser normalerweise weitergegeben, als ob beim Zugriff auf das DRBD-Gerät ein Fehler aufgetreten wäre.
Wird das DRBD-Gerät in einem HA-Cluster eingesetzt, so kann es

erwünscht sein, daß, wenn so ein Fehler auftritt, der Kernel sofort seine Aktivitäten einstellt und damit einem anderen Knoten im Cluster die Möglichkeit gibt, die Aufgaben des Knotens mit der defekten Festplatte zu übernehmen.

Die Größenangaben können mit den Multiplikatoren K (1024), M (2^{20}) oder G (2^{30}) versehen werden. Wird kein Multiplikator angegeben, wird bei den Argumenten von --disk-size und --sync-rate der Multiplikator 1024 angenommen.

5.3.1.2 Verwendung

Die DRBD-Geräte müssen natürlich auf beiden Knoten konfiguriert werden. Falls die Knoten mit mehreren Netzwerkkarten ausgestattet sind, muß darauf geachtet werden, daß die angegebenen IP-Adressen in verbundenen Netzen liegen.

Nachdem das DRBD-Gerät die Konfiguration bekommen hat, versucht es, zuerst zum Partnergerät eine TCP-Verbindung aufzubauen. Falls dies aber nicht funktioniert, weil an dieser Adresse keine Verbindung entgegengenommen wird, wartet DRBD darauf, daß das Partnergerät versucht, eine Verbindung aufzubauen.

Falls das Netzwerk zwischen den beiden Knoten ausfällt, muß eines der beiden Geräte neu konfiguriert werden, damit es wieder versucht, die TCP-Verbindung aufzubauen.

5.3.2 Steuerung im Betrieb

```
drbdsetup dev {PRI|SEC|WAIT|REPL}
```

PRI Mit diesem Kommando kann man das DRBD-Gerät in den primären Zustand bringen. Dieses Kommando kann sowohl vor der Konfiguration als auch danach verwendet werden.

SEC Dieses Kommando bringt das DRBD-Gerät in den sekundären Zustand.

WAIT Dieses Kommando wartet so lange, bis jeglicher Synchronisationsvorgang abgeschlossen ist. Es kann sowohl auf Geräten im primären als auch auf Geräten im sekundären Zustand verwendet werden. Läuft kein Synchronisationsvorgang, wird das Kommando sofort beendet.
Dieses Kommando wurde speziell für die Integration mit dem Heartbeat Cluster-Manager eingeführt.

REPL Mit diesem Kommando kann man eine vollständige Synchronisation verlangen. Es muß auf dem Gerät im primären Zustand gegeben werden und wird dann benötigt, wenn eine der Festplatten ausgefallen ist und durch eine neue ersetzt wurde. Wenn dieser Knoten wieder in den Cluster aufgenommen wird, aktualisiert DRBD mit dem schnellen Synchronisationsmechanismus alle Blöcke, die während des Ausfalls verändert wurden. Da es sich aber um eine neue Festplatte handelt, ist das nicht ausreichend. Mit Hilfe dieses Kommandos kann die neue Festplatte auf den Datenstand des Clusters gebracht werden. Bei der Replikation werden die Daten vom primären auf das sekundäre Gerät kopiert.

5.4 Performance

5.4.1 Durchsatz

Da DRBD, wie der Linux-Kernel selbst, frei[9] zum Herunterladen angeboten wird, kann bei Messungen der Leistung von DRBD auf einige Meßergebnisse, die von DRBD-Anwendern zur Verfügung gestellt wurden, zurückgegriffen werden.

Damit diese Leistungsmessungen vergleichbar sind, wurde eine Applikation entwickelt, der die Leistung des DRBD-Gerätetreibers auf der vorhandenen Installation automatisch bestimmt.

Diese Applikation besteht aus einem Kommando, das im wesentlichen dd(1) nachempfunden ist, aber die erzielte Geschwindigkeit der Kopieroperation ausgeben kann, und einem Shell-Skript, das folgende Eigenschaften des vorhandenen Clusters bestimmt:

- Versionen der Linux-Kernels
- Prozessorarchitekturen
- BogoMips-Wert[10] der Prozessoren
- Durchsatz der Festplatten bei sequentiellen Schreibzugriffen
- Durchsatz der DRBD-Geräte, ohne Verbindung
- Bandbreite der Netzwerkverbindung

[9]Die Rechte und Pflichten des Linzenznehmers sind in der GPL eindeutig definiert.

[10]Die BogoMips werden durch die Ausführung einer kurzen Schleife ermittelt. Da moderne Prozessoren diese Schleife verschieden gut optimieren können, ist dieser Wert kein exakter Indikator für die Leistungsfähigkeit eines Prozessors.

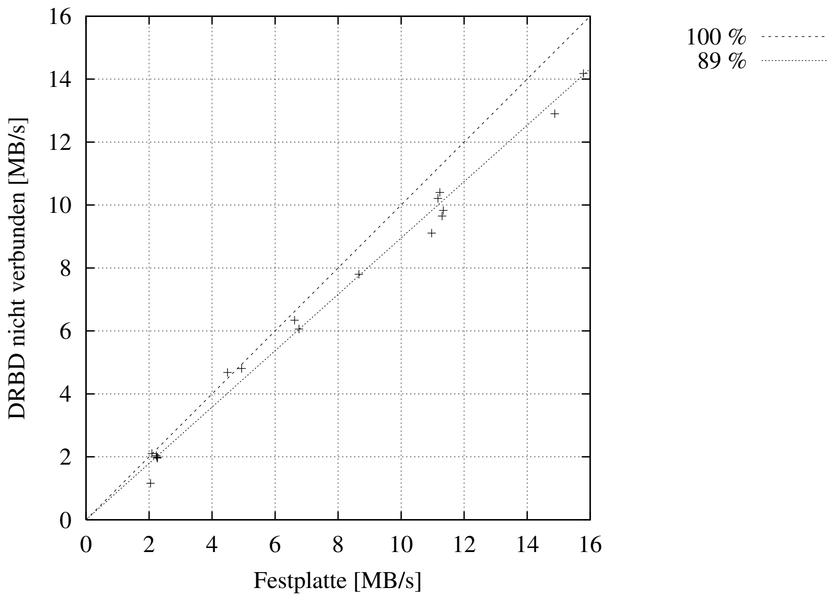


Abbildung 5.2: Durchsatz von DRBD ohne Verbindung

- Minimal-, Durchschnitts- und Maximalzeit, um ein ICMP-Ping Paket zwischen den beiden Knoten hin- und zurückzuschicken (bei 50 Paketen).
- Durchsatz aller DRBD-Protokolle.

Auf die Messung des Durchsatzes beim Lesen wurde verzichtet, da die Ergebnisse durch den Buffer-Cache, druch in Festplatten und Kontroller integrierte Caches und Read-Ahead des Kernels stark verfälscht werden.

Da Lesezugriffe immer lokal durchgeführt werden, kann davon ausgegangen werden, daß der Durchsatz im gleichen Verhältnis zum Durchsatz der Festplatte steht, wie das bei Schreibzugriffen im nicht verbundenen Zustand der Fall ist.

5.4.1.1 Ergebnisse

Die numerischen Ergebnisse der Messungen sind in Anhang A aufgelistet.

Arbeitet DRBD ohne Verbindung, muß es also nur die Schreibaufforderungen an die lokale Festplatte weitergeben, werden im Durchschnitt 89,5 % des Durchsatzes erreicht, den die Festplatte bieten kann. Die Standardabweichung beträgt nur 1,41 %. (Siehe Abbildung 5.2.)

So ein einfacher und genereller Zusammenhang zwischen dem Durchsatz der Festplatten und des Netzwerkes und dem resultierenden Durchsatz eines

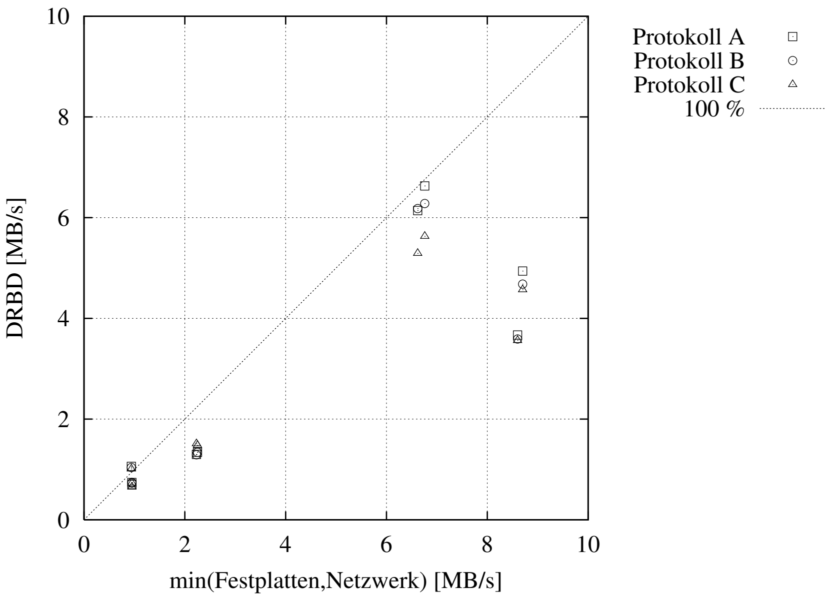


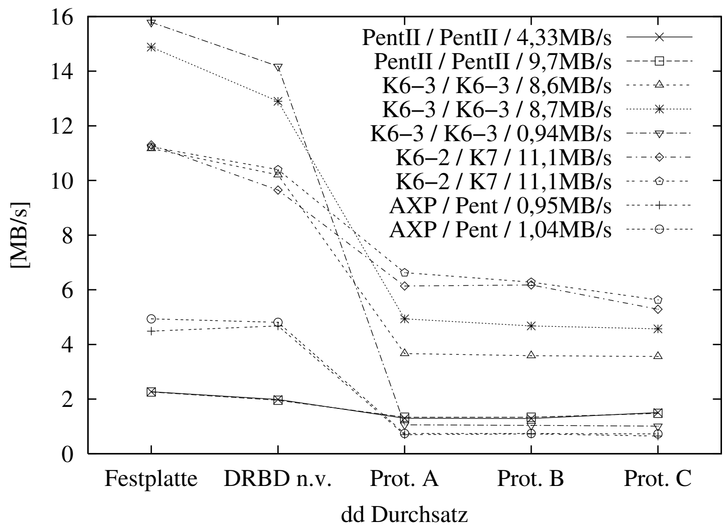
Abbildung 5.3: Durchsatz von DRBD mit Verbindung.

DRBD-Gerätes im verbundenen Zustand ist leider nicht vorhanden. Der Durchsatz wird auf jeden Fall durch den Durchsatz der Festplatten und des Netzwerkes begrenzt. Protokoll A erreicht im Durchschnitt 66,8 % des maximal möglichen Wertes mit einer Standardabweichung von 11,91 %. (Siehe Abbildung 5.3.)

Beim Vergleich der Protokolle ist erwartungsgemäß Protokoll A am schnellsten, gefolgt von Protokoll B und C. Die Unterschiede sind aber minimal. Protokoll B erreicht im Durchschnitt 99,98 % des Durchsatzes von A und Protokoll C 97,11 % des Durchsatzes von B. Dies ist darauf zurückzuführen, daß DRBD bis zu 42[11] Schreibaufforderungen gleichzeitig bearbeiten kann, und die Latenzzeiten aller getesteten Netzwerkkonfigurationen sehr niedrig waren.

In Abbildung 5.4 ist ebenfalls erkennbar, daß es auch vorkommt, daß Protokoll C einen höheren Durchsatz erzielt als Protokoll A und B. Bei der Verwendung von Protokoll C wird am sekundären Knoten nie versucht, mehr als 42 Schreibanforderungen in Auftrag zu geben. Der Festplattentreiber beginnt aber erst nach dem Ablauf einer Zeitspanne, oder wenn die Request-Queue keine Einträge mehr aufnehmen kann, mit der Abarbeitung der Schreibanforderungen. Daher wird bei Protokoll C die Verarbeitung der Schreiban-

[11] Ab der Kernelversion 2.2.13 hat die Request-Queue 128 Einträge. Wegen der in Abschnitt 5.2.1.1 (Seite 42) beschriebenen Gründe kann DRBD nur 33 % dieser Einträge für Schreibaufforderungen verwenden.



Bei der Beschriftung der Meßwerte wurde der Prozessor des primären Systems und des sekundären Systems sowie der Durchsatz des Netzwerkes angegeben.

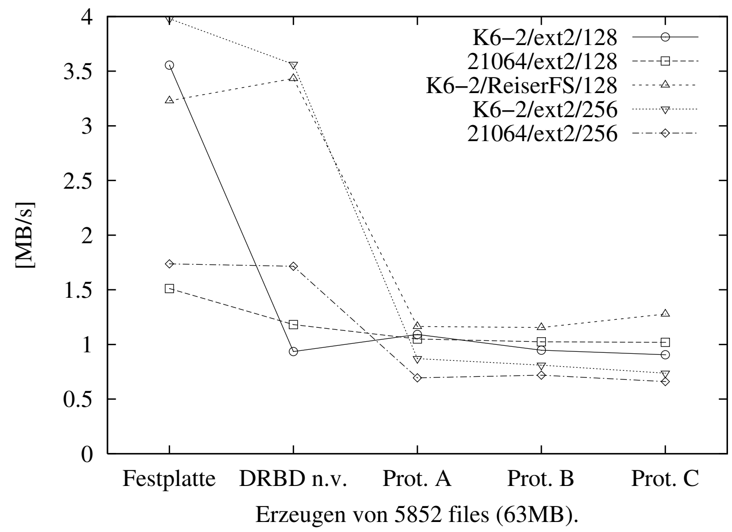
Abbildung 5.4: Die Protokolle im Vergleich

forderungen in der Request-Queue explizit gestartet, wenn mindestens 32[12] Schreibanforderungen noch nicht abgeschlossen sind. Weil bei Protokoll C die Abarbeitung der Schreibaufforderung bereits früher gestartet wird, kann Protokoll C in einzelnen Fällen einen besseren Durchsatz als Protokoll B erzielen.

Aus den im Anhang A aufgelisteten Daten läßt sich der Trend ablesen, daß, wenn der sekundäre Knoten mit der besseren Festplatte ausgestattet ist, DRBD eine höhere Performance erreicht. Dies deutet darauf hin, daß der empfangende Knoten den Durchsatz über die Flußkontrolle des TCP-Protokolls limitiert. Der Empfangsprozeß seinerseits kann, neben dem Empfang vom Netzwerk, beim Erzeugen der Schreibaufforderungen und beim Warten auf Abschluß der Schreiboperationen blockieren.

Würde man allerdings diese limitierenden Faktoren ausschalten, also auf das sofortige Schreiben auf die Festplatte verzichten, und das Schreiben auf die Festplatte asynchron durch die Kernel-Daemonen erledigen lassen, wäre die richtige Reihenfolge der Schreiboperationen nicht mehr garantiert.

[12] 25 % von 128.



21064 ist der Prozessor von Knoten 2; K6-2 ist der Prozessort von Knoten1

Abbildung 5.5: DRBD und Filesystem

5.4.2 Dateisystem

Meistens wird ein blockorientiertes Gerät in Zusammenarbeit mit einem Dateisystem verwendet. Im folgenden wird die Leistung von DRBD mit Ext2 und ReiserFS (siehe auch Abschnitt 6.2 auf Seite 69) untersucht. Für diese Messungen wurde folgende Installation verwendet:

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.14	i586	901,2	11,87	11,57	1,75	1,74	2,06
2	Linux	2.2.14	alpha	295,69	4,16	3,74	3,54	3,35	3,56

Netzwerkbandbreite:	6,38 MB/s	Datengröße:	50 MB
Netzwerklatenz:	0,1/0,1/0,3 ms	DRBD Version:	0.5.5

In Abbildung 5.5 ist die Zeit dargestellt, die benötigt wird, um 5812 Datein mit einer Gesamtgröße von 63MB zu erzeugen:

- Knoten 1 ist mit einer IDE-Festplatte mit aktiviertem Write-Cache

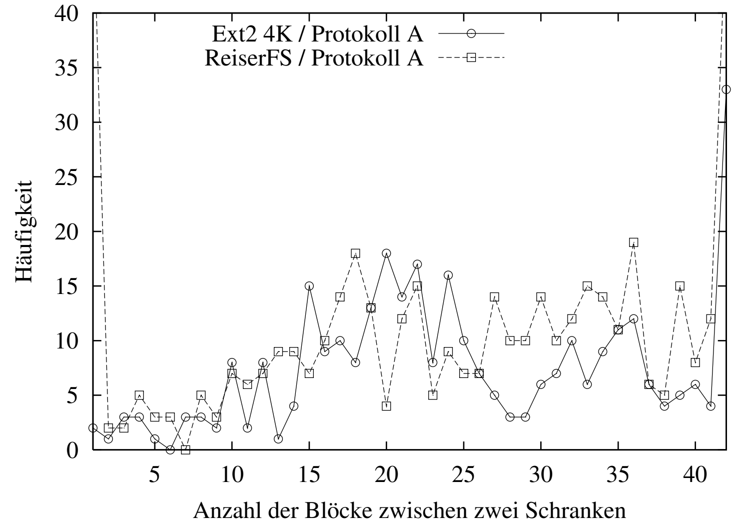


Abbildung 5.6: Abstand der Write-Barriers

ausgestattet. Dieser beschleunigt Operationen auf der lokalen Festplatte und auf DRBD im nicht verbundenen Zustand.

- Die beste Leistung für Ext2 in Kombination mit Protokoll B und C konnte beim Schreiben auf Knoten 2 erzielt werden. Dies untermauert die These, daß, wenn das schnellere System im sekundären Zustand verwendet wird, bessere Leistungsergebnisse erzielbar sind.
- Mit ReiserFS kann die beste Gesamtleistung vorgewiesen werden. Es muß allerdings erwähnt werden, daß ext mit einer Blockgröße von 1K verwendet wurde, während ReiserFS eine Blockgröße von 4K verwendet.
- Die Vergrößerung der Request-Queue von 128 auf 256 Einträge erhöht zwar die Leistung im lokalen Fall, verschlechtert allerdings die Leistung, wenn DRBD die Daten spiegelt.

5.4.2.1 Abstand der Write-Barriers

Um den Auswirkungen des Write-Barrier-Protokolls besser verstehen zu können, wurde auch die Anzahl der Blöcke, die zwischen zwei Write-Barriers versendet werden, untersucht. Für die beiden Messungen wurden 3449 Dateien mit einer Gesamtgröße von 24MB erzeugt. Beide Filesysteme wurden mit einer Blockgröße von 4K betrieben.

In den Abbildungen 5.6 und 5.7 ist das Auftreten der Abstände zwischen zwei Write-Barrier Paketen dargestellt. Bei Protokoll A hat ext2 33 mal 42

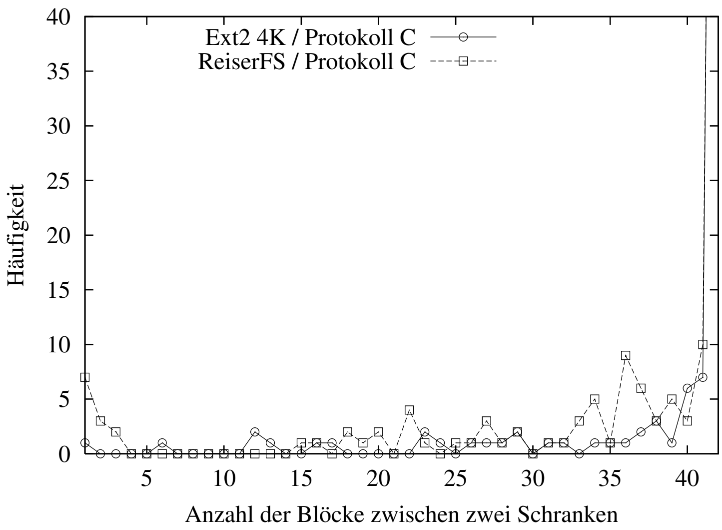


Abbildung 5.7: Abstand der Write-Barriers

Blöcke und ReiserFS 49 mal 42 Blöcke ohne Write-Barrier versendet, bei Protokoll C waren es bei ext2 162 und bei ReiserFS 156.

Daß mit Protokoll A wesentlich mehr Write-Barriers generiert werden, kommt daher, daß die einzelne Schreiboperation deutlich schneller abgeschlossen wird als bei Protokoll C. (Siehe Abschnitt 3.3.1.1 auf Seite 24)

Im Vergleich der Filesysteme ist sichtbar, daß mit ReiserFS (siehe auch Abschnitt 6.2.3 auf Seite 70) mehr kurze Abstände zwischen Write-Barrier-Paketen zustandekommen als bei Ext2. Der Grund dafür dürften die Schreiboperationen in das Journal sein. ReiserFS muß auf den Abschluß der Schreiboperationen ins Journal warten, bevor es die Aktualisierung der Datenstrukturen des Dateisystems vornehmen darf. Der besonders hohe Wert beim Abstand 1 könnte von diesen Schreiboperationen ins Journal stammen. Die höheren Werte bei Abständen 25 bis 41 könnten von Aktualisierungen, die mit der nächsten Schreiboperation ins Journal zusammengefasst wurden, stammen.

Kapitel 6

HA unter Linux

Neben DRBD werden noch eine Reihe anderer Fähigkeiten und Komponenten benötigt, um einen HA-Cluster zu implementieren. Dieses Kapitel gibt einen Überblick darüber, wie sich DRBD in die anderen, für die Implementierung eines HA-Cluster notwendigen, Komponenten eingliedert.

6.1 Cluster Management

Die wohl wichtigste Komponente eines HA-Clusters ist der Cluster-Manager. Dabei handelt es sich um eine verteilte Applikation, die bestimmt, welche Knoten Teil des Clusters sind, und die zur Laufzeit notwendige Konfiguration der Subsysteme vornimmt, die den Cluster bilden.

6.1.1 Struktur

6.1.1.1 Membership

Der Cluster-Manager muß ständig überprüfen, ob alle am Cluster beteiligten Knoten noch funktionieren. Dies wird in der Regel dadurch realisiert, daß auf jedem Knoten des Clusters eine Instanz eines Daemons läuft und diese Programme sogenannte Heartbeat-Pakete austauschen. Stellen diese Programme fest, daß von einem Knoten keine Heratbeat-Pakete mehr empfangen werden, wird dieser Knoten aus dem Cluster ausgeschlossen.

Werden von einem Knoten wiederum Heartbeat-Pakete empfangen, z.B. nach erfolgreicher Reparatur durch die Systemadministratoren, so wird dieser wieder in den Cluster aufgenommen.

Wenn der Cluster durch einen Netzwerkausfall in zwei Cluster zerfällt (sog. split brain problem), stellt dies die Cluster-Management Software vor eine Herausforderung. Mögliche Lösungen:

- Einführung des Mehrheitsprinzips
  Jener Teil-Cluster, der über 50 % der Stimmen hat, übernimmt die Services. Falls der Cluster aus einer ungeraden Anzahl von Knoten besteht, bekommt jeder Knoten eine Stimme, andernfalls muß ein Knoten zwei Stimmen haben. Tritt der Netzwerkausfall gleichzeitig mit dem Ausfall eines Knotens auf, dann kann es dazu kommen, daß keiner der beiden Teil-Cluster die Mehrheit hat und daher auch keiner die Arbeit aufnimmt.

- Globaler Lock
  In einem Cluster mit einer Shared Disk, übernimmt der Teil-Cluster die Arbeit, die einen Lock auf der Shared Disk belegen kann.

- Redundante Kommunikationswege
  Abgesehen von redundanter Auslegung des Netzwerkes sind auch nicht IP-basierende Verbindungen sehr beliebt, z.B. serielle Kabel, USB. Sollte aus irgendeinem Grund das doch komplexe IP-Subsystem des Kernels nicht mehr funktionieren, so ist Kommunikation über die serielle Verbindung noch möglich.

6.1.1.2 Service

Der Vorteil eines HA-Clusters für den Anwender ist, daß ihm bestimmte Services (od. Applikationen) permanent zur Verfügung stehen. Hier muß der Cluster-Manager eine Schnittstelle bieten, sodaß er dieses Service auf verschiedenen Knoten starten und stoppen kann. Weiters muß der Cluster-Manager über die für die Services notwendigen Ressourcen informiert sein und feststellen können, ob das Service korrekt arbeitet. Eine Ressource, die eigentlich jedes Service hat, ist zumindest eine IP-Adresse.

6.1.1.3 Übernahme einer IP-Adresse

Normalerweise bekommt jede Netzwerkkarte eine fixe IP-Adresse, die unter anderem auch von den Heartbeat-Paketen des Cluster-Managers adressiert wird.

Zusätzlich kann, wenn das optionale Feature IP_ALIAS bei der Kernelkonfiguration angegeben wurde, jede Netzwerkkarte eine Reihe von zusätzlichen IP-Adressen bekommen. Diese auch virtuelle IP-Adressen genannten Adressen können nun vom Cluster-Manager beliebig von einem Knoten zum anderen verschoben werden.

Da aber bei der Datenübertragung auf Ethernet MAC-Adressen[1] verwendet

[1] Media Access Control

werden müssen, hat jedes Gerät am Netz einen ARP-Cache[2]. Wenn ein Gerät an ein anderes Gerät im selben Netz ein Paket senden will, muß es zuvor dessen MAC-Adresse erfragen, indem es ein ARP-Request-Paket an alle Geräte im Netzwerksegment schickt. Das Gerät, das die gesuchte IP-Adresse hat, antwortet darauf mit einem ARP-Reply-Paket, das an den Absender des Request-Paketes gerichtet ist. Die so erhaltenen Zuordnungen zwischen IP-Adressen und MAC-Adressen werden im ARP-Cache abgelegt.

Diese ARP-Caches in den anderen Geräten im Netzwerksegment sind ein Problem bei der Verschiebung von virtuellen IP-Adressen. In diesen Caches kann natürlich die MAC-Adresse, die der IP-Adresse vor der Verschiebung zugeordnet war, enthalten sein.

Durch das Versenden von ARP-Reply-Paketen an die Broadcast-Adresse des LAN-Segmentes (sog. gratuitous ARP) kann den ARP-Caches der Geräte im Segment die neue MAC-Adresse mitgeteilt werden. Diese Methode geht ursprünglich auf ein Posting zurück, das ARP-Spoofing als Methode beschreibt, um in Computersysteme einzubrechen. [Hor98][Vol97]

Dieses ARP-Reply-Paket muß regelmäßig versendet werden, wobei das Zeitintervall kürzer sein muß als die Zeit, für die ein ARP-Cache seine Einträge behält. Denn wenn die Zuordnung aus einem der ARP-Caches gelöscht wird, das betroffene Gerät einen ARP-Request aussendet und das Gerät, das die virtuelle IP-Adresse zuvor hatte, diese nicht ordnungsgemäß entfernt hat, könnte es dazu kommen, daß vom ARP-Subsystem dieses Gerätes ein ARP-Reply mit der alten MAC-Adresse zurückgesendet wird.

6.1.1.4 MAC-Adreßübernahme

Eine alternative Methode wäre es, MAC-Adressen zu übernehmen. Bei dieser Methode ist die Aussendung von ARP-Reply-Paketen nicht notwendig, aber bei normalen Netzwerkkarten führt dies dazu, daß die fixe IP-Adresse der Karte nicht verläßlich erreichbar ist.

Diese Methode kann daher nur dann zuverlässig eingesetzt werden, wenn für jede virtuelle IP-Adresse eine eigene Karte in jedem Knoten des Clusters vorhanden ist.

Die Netzwerkkarten, die auf dem sog. Tulip Chip von DEC basieren, haben die Fähigkeit, auf bis zu 14[3] zusätzliche MAC-Adressen reagieren zu können. Diese Fähigkeit könnte man ausnützen, um MAC-Adreßübernahme für bis zu 14 virtuelle IP-Adressen auf einer einzigen Tulip-basierten Karte zu implementieren.

[2] Address Resolution Protocol

[3] Ein Tulip Chip kann 16 MAC-Adressen, eine wird für die Broadcastadresse und eine für die fixe IP-Adresse benötigt.

6.1.2 Cluster-Management-Software

Da es lange Zeit überhaupt keine fertigen Lösungen auf diesem Gebiet gab, sind zwei spezielle Cluster-Manager auf der Basis von Shell-Skripts entstanden. Einer verwendet MAC-Adreßübernahme [Lew99], während der andere ARP-Pakete verwendet [Hor98].

6.1.2.1 Heartbeat

Heartbeat [Rob00] ist ein Cluster-Manager, der — wie der Linux-Kernel — nach den Regeln der GPL frei erhältlich ist.

Die Konfiguration von Heartbeat wird in Konfigurationsfiles vorgenommen, die im /etc/ha.d/ aller Knotenrechner abgelegt werden müssen. Heartbeat stellt die Konsistenz dieser Konfigurationsfiles nicht sicher.

Heartbeat bietet wie in den vorigen Abschnitten beschrieben:

- Überwachung der einzelnen Knoten.
  Dies kann über mehrere IP-basierte Netzwerke oder über serielle Verbindungen erfolgen. Solange Heartbeat-Pakete über eine der möglichen Verbindungen ausgetauscht werden können, betrachten sich die Knoten als lebendig.

  Aus diesem Verhalten kann folgendes Beispiel konstruiert werden: Zwei Knoten sind mit einem Ethernetsegment und einem seriellen Kabel verbunden, die Clients sind ebenfalls an das Ethernetsegment angeschlossen. Wenn nun die Netzwerkkarte des Knotens, auf dem das Service läuft, ausfällt, beläßt Heartbeat das Service auf diesem Knoten, weil die beiden Knoten immer noch über das serielle Kabel kommunizieren können.

  Die Heartbeat-Pakete, die versendet werden, können nach verschiedenen Methoden authentifiziert werden, sodaß auch der Betrieb über nicht vertrauenswürdige Netze möglich ist.

- Verwaltung hochverfügbarer Services.
  Eine laufende Heartbeat-Installation kann mehrere Services verwalten, wobei eine beliebige Anzahl von Services einer virtuellen IP-Adresse zugeordnet werden kann. Abhängigkeiten zwischen den einzelnen Services können nicht definiert werden, doch werden die Services in einer definierten Reihenfolge gestartet, während sie in der genau umgekehrten Reihenfolge gestoppt werden.

  Die Services werden über Shell-Skripten, wie sie auch vom init Prozeß verwendet werden, gesteuert, wobei auch fixe Parameter übergeben werden können.

Die Übernahme von IP-Adressen wird ebenfalls von einem Shell Skript implementiert und gliedert sich so in das Konfigurationsschema ein.

- Überwachung der Services ist nicht implementiert.
  Die einzige Möglichkeit, dies zu erreichen ist der Einsatz eines eigenständigen Programmes, das die Verfügbarkeit des Services überprüft und, falls diese nicht mehr gegeben ist, den lokalen Heartbeat-Prozeß mit einem Signal beendet. Dies hat natürlich den Nachteil, daß nicht nur das ausgefallene Service sondern alle Services, die auf diesem Knoten laufen, von dem Knoten abgesiedelt werden.

- Dynamische Änderung der Konfiguration.
  Es gibt eine Kommandozeilenoption, mit der man eine laufende Heartbeat-Instanz dazu bewegen kann, ihre Konfigurationsdateien neu einzulesen. Wenn dabei ein Service aus der Konfiguration entfernt wurde, kennt Heartbeat dieses Service nicht mehr und sorgt daher nicht mehr für eine ordnungsgemäße Beendigung.

Die bisherige Entwicklung von Heartbeat hat sich auf den Einsatz von Services beschränkt, die ihre Daten entweder selbst replizieren können, oder bei denen die Daten mit Hilfe anderer Mechanismen repliziert werden. So ist Heartbeat bis jetzt vor allem mit folgenden Services eingesetzt worden:

- DNS-Server
  Die Zonenfiles des DNS-Dienstes müssen bei einer Änderung manuell repliziert werden.

- HTTP-Proxy-Cache
  Ein Proxy kann im Falle des Verlustes seines Cache-Inhaltes seine Arbeit wieder aufnehmen, da er die Web-Seiten wieder von ihren ursprünglichen Lokationen holen kann.

- Web-Server
  Die Web-Seiten müssen im Falle einer Änderung manuell repliziert werden, z.B. mit rsync.

- Linux Virtual Server (LVS)
  Der Lastaufteiler (load balancer) eines Web-Server Clusters (siehe Abschnitt 2.2.2 auf Seite 8) ist ein SPOF für den ganzen Cluster. Der Zustand, der für LVS von Bedeutung ist, befindet sich direkt im Kernel. Man nimmt den Verlust dieses Zustandes beim Failover in Kauf.

6.1.2.2 LinuxFailSafe

LinuxFailSave ist die Portierung von SGIs FailSave für IRIX, das ebenfalls unter der GPL veröffentlicht werden soll. Bisher ist allerdings noch kein

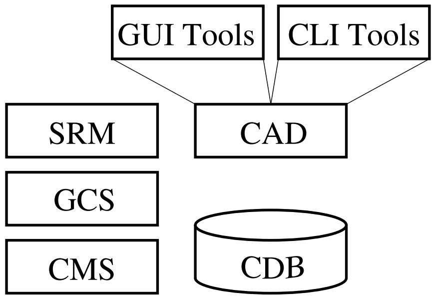


Abbildung 6.1: Architektur von FailSafe

Quelltext veröffentlicht worden.

FailSave besteht aus folgenden Teilen:

- CMS (Cluster Membership Service)
  Dieses Subsystem überwacht alle Knoten durch das ständige Versenden von Heartbeat-Paketen.
- GCS (Group Communication Service)
  Dieses Subsystem baut auf das CMS auf und stellt den Anwendungen eine konsistente Sicht von Prozeßgruppen zur Verfügung und sorgt für eine zuverlässige Zustellung von Nachrichten.
- SRM (System Resource Manager)
  Der SRM baut auf die Funktionalität des GCS auf und verwaltet alle clusterweiten Ressourcen wie: Shared Disks, Dateisysteme, IP-Adressen, Services und frei definierbare Ressourcen.
- CDB (Cluster Configuration Database)
  Diese Datenbank hat ein einheitliches Interface für Applikationen, während am Backend verschiedene Implementierungen vorhanden sind.

  Diese Datenbank wird auf mehreren Knoten des Clusters redundant abgelegt, und, falls ein Knoten nach einem Ausfall einen veralteten Datenstand in seiner Datenbank hat, wird diese komplett durch den Datenstand einer aktuellen Datenbank ersetzt.

FailSafe kann Cluster bis zu 16 Knoten verwalten und bietet unter anderem folgende Features:

- Lokale Überwachung eines Services, sowohl passiv (z.B. Überwachung, ob ein Prozeß noch vorhanden ist) als auch aktiv (z.B. Datenbankabfrage, HTTP-Anfrage, . . .)

- Lokales Failover, wegen Fehler in der Software die das Service anbietet (siehe Abschnitt 2.3.2 auf Seite 11)
- Festlegung der Failover-Reihenfolge
  Für ein Service gibt es eine geordnete Liste an Knoten. Ist die Ausführung des Services auf dem ersten Knoten möglich, so findet sie auch dort statt, andernfalls wird der nächste Knoten der Liste betrachtet.

  Dadurch ist die Realisierung einer sog. Stern-Architektur möglich. Dabei ist ein Knoten des Clusters als Backup-Knoten für mehrere andere Knoten und deren Services abgestellt.
- Prioritäten der Services
  Beispiel: Es laufen zwei Services auf einem Knoten, die beide ein bestimmtes Filesystem (gemeinsame Ressource) benötigen. Wenn ein Service auf diesem Knoten nicht mehr gestartet werden kann, kann festgelegt werden, ob dann beide Services auf einen anderen Knoten migriert werden oder ob auf das ausgefallene Service verzichtet wird und somit das laufende Service keiner Migration unterzogen werden muß.
- Während der Laufzeit des Clusters können Ressourcen hinzugefügt und weggenommen werden.

6.1.2.3 Mon

Mon ist zwar keine Cluster-Managment Software, bietet dafür aber sehr flexible Möglichkeiten, Netzwerkinfrastruktur und Dienste im Netzwerk zu überwachen, und wird aus diesem Grund auch in vielen Artikeln über hohe Verfügbarkeit unter Linux erwähnt. Mon ist ein Daemon, der in Perl geschrieben wurde und unter der GPL weitergegeben wird. Im folgenden eine kurze Beschreibung der Features:

Monitor Ein Monitor ist ein kleines Programm, das vom Mon-Prozeß gestartet wird, eine Ressource überprüft und das Ergebnis an den Mon-Prozeß zurückliefert. Mit Mon werden bereits eine Anzahl solcher Programme mitgeliefert: ICMP echo (ping), SMTP, telnet, FTP, NNTP, HTTP, POP-3, IMAP, Disk space, LDAP, DNS, mSQL, mySQL, RPC und mehrere SNMP-basierte Tests.

Asynchrone Ereignisse Mon kann auch auf asynchrone Ereignisse reagieren. Die Referenzanwendung für diese Fähigkeit ist das Empfangen und die Auswertung von SNMP-Traps.

Alert Ein Alert ist ein Programm, das eine Meldung an einen Systemadministrator weitergeben kann. Bereits vorhanden sind: Email, Pager (via QuickPage, SNPP und Email) und SNMP-Traps.

Abhängigkeiten Es können Abhängigkeiten der Netzwerkressourcen definiert werden, sodaß z.B. beim Ausfall eines Routers nur dieser Ausfall gemeldet wird und nicht die Ausfälle aller Dienste, die in den Netzen, die nur über den eben ausgefallenen Router erreichbar sind, angeboten werden.

Benachrichtigung Beliebige Benachrichtigungen können an jedes Ereignis geknüpft werden. So ist es auch möglich, einen Bereitschaftsplan mit Mon zu realisieren. Fällt z.B. eine bestimmte Ressource zwischen 8:00 und 18:00 aus, wird an alle Pager der Administratoren eine Nachricht gesendet. Tritt der Ausfall in der Nacht auf, wird nur ein Administrator per Pager benachrichtigt.

Frontend Für die Administration und Bestätigung von Nachrichten stehen Kommandozeilen-Tools und ein Web-Frontend zur Verfügung. Die Erstellung weiterer Frontends ist vorgesehen.

6.2 Filesysteme

Ext2 ist das einzige der hier vorgestellten Dateisysteme, das momentan ein Bestandteil von Linux ist. Die anderen werden alle noch extern vom Kernel weiterentwickelt.

Die wichtigste Eigenschaft, die ein Dateisystem mitbringen muß, um für den Einsatz auf einem HA-Cluster geeignet zu sein, ist die rasche Einsatzfähigkeit nach einem Absturz.

6.2.1 ext2

Das Grundkonzept von ext2 [BBDK97][HHMK94][Rus98] ist dem des BSD FFS sehr ähnlich. Das Design von BSD FFS stammt aus einer Zeit, als eine Konsistenzprüfung des Filesystems aufgrund der damals üblichen Größe der Festplatten in annehmbarer Zeit durchgeführt werden konnte.

Mit ext2 können theoretisch Filesysteme mit einer Größe von bis zu 16 Terrabyte angelegt werden. Dies ergibt sich daraus, daß ext2 32 Bit für die Adressierung der Blöcke verwendet und mit einer maximalen Blockgröße von 4K arbeiten kann ($2^{32} * 2^{12} = 2^{44}$). Berücksichtigt man jedoch, daß das IO-Subsystem des Kernels alle Blocknummern in Sektornummern umrechnet, so ist die maximale Größe einer Partition auf 32-Bit-Plattformen auf 2 Terabyte beschränkt ($2^{32} * 2^{9} = 2^{41}$).

Nimmt man an, daß die Konsistenzprüfung pro GB etwa 60 Sekunden dauert, würde die Konsistenzprüfung eines 2-TB-Filesystems etwa 33 Stunden in

Anspruch nehmen[4].

Diese Abschätzung soll nur die mögliche Größenordnung verdeutlichen, in Wirklichkeit ist die Dauer der Konsistenzprüfung von der Anzahl der Dateien abhängig. Bei einem Dateisystem, das voll mit vielen kleinen Dateien ist, kann die Zeit, die die Überprüfung eines Gigabytes in Anspruch nimmt, über den angenommenen 60 Sekunden liegen.

6.2.2 ext3

Um das Problem der langen Konsistenzprüfung zu umgehen, ist ext3 [Twe98] ein Journaling-Filesystem. Dabei werden die beabsichtigten Änderungen vor einer Änderung der Metadatenstrukturen im Filesysstem in ein Log geschreiben. Nach der erfolgreichen Durchführung der Änderungen wird dies ebenfalls im Log vermerkt.

Fällt der Computer irgendwann während dieser Operationen aus, kann durch erneute Durchführung der letzten Aktionen im Log die Konsistenz des Filesystems wieder hergestellt werden, ohne daß ein aufwendiger Konsistenzcheck notwendig ist.

Ext3 wird sowohl Metadata-Journaling, das nur die Konsistenz des Dateisystems gewährleisten kann, als auch volles Journaling, bei dem auch der richtige Inhalt aller Dateien im Falle eines Systemabsturzes gewährleistet werden kann, unterstützen.

Ext3 ist eine Erweiterung des ext2-Filesystems, das Log wird in einer Datei auf dem ext2-Filesystem abgelegt. Jedes ext2-Filesystem kann durch Anlegen dieser Logdatei in ein ext3-Dateisystem umgewandelt werden. Es ist sogar möglich, dieses später wieder mit ext2 zu betreiben.

Leider befindet sich ext3 noch in einem sehr frühen Entwicklungsstadium, und die Aufnahme in den Linuxkern ist in nächster Zeit nicht zu erwarten.

6.2.3 ReiserFS

Die Lizenz von ReiserFS [Reis99] ist die GPL mit einem Zusatz[5].

ReiserFS ist ein komplett neues Design und weist keine Ähnlichkeiten zum BSD FFS auf. In der momentanen Implementierung[6] werden 32 Bit für

[4] Dies ist auch der Grund, weshalb das neueste Feature des fsck.ext2 Utilities eine Fortschrittsanzeige ist.

[5] Die Erweiterung der GPL durch Zusätze ist in der GPL ausdrücklich verboten.

[6] Leider haben die Entwickler von ReiserFS schon öfters das Layout der Daten auf der Festplatte zwischen Versionen ihres Filesystems geändert. Die einzige Möglichkeit, so einen Versionssprung mitzumachen, war die Neuerstellung des Filesystems.

Blocknummern verwendet; und es gelten somit die gleichen Größeneinschränkungen für das gesamte Filesystem wie bei ext2.

ReiserFS verwendet keine vorallokierten Inodes, und es werden Filenamen, Directories und die Daten der Files mit Hilfe von B*Bäumen verwaltet. Es unterstützt momentan nur 4K-Blöcke, kann aber kleine Dateien und Dateienden zusammenlegen.

Weiters ist ReiserFS mit Metadata-Journaling ausgestattet. Das Filesystem kann vergrößert werden während es in Verwendung ist. Verkleinern des Filesystems ist nur möglich, wenn das Filesystem nicht im System angemeldet ist.

ReiserFS hat gute Chancen, in die 2.4.x Versionen von Linux aufgenommen zu werden.

6.2.4 XFS

XFS ist jenes Filesystem, mit dem auch IRIX von SGI ausgeliefert wird. Die Linuxportierung dieses Dateisystems wird nun unter der GPL veröffentlicht. Die Eckdaten laut SGI:

- 64-Bit-Blocknummern
- Effiziente Speicherung von Dateien mit Löchern
- Journaling
- Blockgröße bis 256 KB
- Änderung der Größe während es in Verwendung ist
- Verwendet B-Bäume
- Applikationen können garantierten Durchsatz bei Schreib- oder Lesezugriffen verlangen

6.2.5 JFS

Das JFS von IBM wurde vor kurzem ebenfalls für Linux angekündigt, gleichzeitig wurde auch der Quelltext unter der GPL veröffentlicht. Bis jetzt ist die Portierung auf Linux allerdings noch nicht fertig. Laut IBM:

- 64-Bit-Blocknummern
- Effiziente Speicherung von Dateien mit Löchern
- Journaling
- Verwendet B+Bäume

6.2.6 LinLogFS

LinLogFS ist ein Log-Structured-Filesystem für Linux. Bei so einem Filesystem werden weder Datenblöcke noch Metadaten jemals direkt überschrieben. Aktualisierungen werden durch Anlegen fortlaufender Segmente gespeichert. Auf diese Architektur aufbauend kann man Snapshots und vorhergehende Versionen von Dateien anbieten.

Mit Hilfe von Snapshots kann man konsistente Backups durchführen, während das Filesystem von laufenden Applikationen verändert werden kann.

Dadurch, daß die Daten bei einer Aktualisierung nicht überschrieben werden, kann man auf ältere Versionen einer Datei zugreifen. Es handelt sich um eine verallgemeinerte Version des „undelete” Kommandos.

Die Freigabe von Speicher erfolgt bei dieser Art von Dateisystem durch einen sog. Cleaner. Dies ist ein Prozeß, der im Hintergrund abläuft und nach Segmenten sucht, deren Inhalt zum Großteil von keinem Snapshot oder keiner alten Version einer Datei referenziert wird. Solche Segmente werden dann freigegeben, und der Teil, der von ihnen noch gebraucht wird, wird in neue Segmente geschrieben.

Wie ein Journaling-Filesystem benötigt ein Log-Structured-Filesystem keinen Konsistenzcheck nach einem Absturz des Computers.

Wie viele anderen der hier vorgestellten Dateisysteme befindet sich LinLogFS leider ebenfalls noch im Entwicklungsstadium, es gibt z.B. noch keinen Cleaner.

6.2.7 GFS

Die herausragende Eigenschaft von GFS [PBB+99] ist es, daß ein physisches Dateisystem von mehreren angeschlossenen Computern gleichzeitig verwendet werden kann.

Weiters kann GFS ebenfalls 64-Bit-Blocknummern vorweisen, an Journaling wird leider noch gearbeitet. Das heißt, wenn einer der Knoten während des Schreibens abstürzt, muß das Dateisystem auf allen beteiligten Knoten abgemeldet werden, ein Konsistenzcheck durchgeführt werden und anschließend das Dateisystem wieder angemeldet werden.

6.2.7.1 Übersicht

Dateisystem	Größe des Quellcodes	Status
ext2	150KB	stabil
ext3	320KB	alpha
ReiserFS	750KB	beta
XFS	3.8MB	alpha
JFS	602KB	Entwicklung
LinLogFS	580KB	Entwicklung

6.3 Implementierung

6.3.1 Struktur

Die einzige Clusterinstallation, die bisher ernsthaften Tests unterzogen wurde, war eine NFS-Installation. Während in letzter Zeit sich der Kernelbasierende NFS-Server als Standard durchsetzen konnte, wurde für diese Installation der ältere, auf normalen User-Prozessen basierende NFS-Dienst eingesetzt. Dieser legt nämlich seinen Zustand in /var/lib/nfs/ ab. Diese Zustandsinformation besteht aus der Liste der momentan authentifizierten Clients. Bei der Realisierung wurden sowohl das Dateisystem, das NFS exportiert, als auch die Dateien, die den Zustand des NFS-Servers beinhalten, auf einem durch DRBD replizierten Dateisystem untergebracht.

Als Cluster-Manager kam Heartbeat zum Einsatz. Dabei mußte ein Shell-Skript geschrieben werden, das DRBD als Service für Heartbeat darstellt.

Wird dieses Service von Heartbeat gestartet, wird das DRBD-Gerät des Knotens in den primären Zustand gebracht und das Filesystem auf diesem angemeldet. Da in dieser Testserie ausschließlich das ext2-Dateisystem verwendet wurde, muß vor dem Anmelden natürlich ein Konsistenzcheck auf dem Filesystem durchgeführt werden.

Bei Aufruf des Service-Skriptes mit dem Stopp-Parameter, wird das Filesystem abgemeldet, und das DRBD-Gerät wird in den sekundären Zustand gebracht.

Die Konfiguration des DRBD-Gerätes wird durch ein weiteres Skript übernommen, das bereits beim Bootvorgang ausgeführt wird. Dieses Skript stellt sicher, daß das DRBD-Kernelmodul geladen wird, und konfiguriert die IP-Adressen für die Verbindung.

Die eigentlichen Angaben zur Konfiguration stehen in einer Konfigurationsdatei, die von den Skripten eingelesen wird. Diese Konfigurationsdatei ist so aufgebaut, daß sie auf beiden Knoten des Clusters identisch ist. Die Skripten stellen anhand der Konfigurationsdatei fest, welche Rolle der Knoten übernehmen soll, auf dem sie laufen, und können so die zutreffenden Angaben aus der Konfigurationsdatei lesen.

Wenn der Heimatknoten eines Services wieder erscheint, bringt Heartbeat das Service sofort auf diesen zurück. Beim Einsatz von DRBD kann es aber sein, daß der Heimatknoten noch damit beschäftigt ist, seinen Datenstand zu aktualisieren. In diesem Fall muß das DRBD-Skript den Boot-Vorgang des Knotens so lange anhalten, bis der Synchronisationsvorgang abgeschlossen ist.

6.3.2 Ergebnisse

Die Abschaltung des aktiven Servers führt erwartungsgemäß dazu, daß der zweite Server nach kurzer Zeit einspringt. Wenn ein Client während des Failovers gerade eine Datei schreibt, ist diese Datei allerdings nach dem Failover nicht fehlerfrei vorhanden.

Mit der synchronen[7] Verwendung des Filesystems konnte dieses Problem behoben werden. Da bei diesen Tests natürlich Protokoll B und C verwendet wurde, vermute ich, daß der Fehler im NFS-Server liegt. Dieser sollte erst dann den Abschluß der Schreiboperation dem Client mitteilen, wenn die Daten wirklich auf nicht flüchtigen Speicher geschrieben[8] sind. Bei der synchronen Durchführung aller Schreibzugriffe geht natürlich einiges an Performance verloren.

Wenn man auf die synchrone Verwendung des Filesystems verzichtet, kann die Aktualisierung der Festplatte mit dem update(8) Kommando gesteuert werden. Mit diesem Kommando kann man steuern, wie lange Datenblöcke, die von einem Filesystem, das mit asynchronen Schreiboperationen arbeitet, stammen, im Buffer-Cache bleiben dürfen, bevor sie auf die Festplatte geschrieben werden müssen. Die Standardeinstellung beträgt 5 Sekunden für Datenblöcke, die Metadaten des Filesystems enthalten, und 30 Sekunden für Datenblöcke mit Userdaten.

Bei der Erprobung des ReiserFS-Dateisystems hat sich herausgestellt, daß dieses keinen generell synchronen Betrieb unterstützt und daß es in der damals verfügbaren Version auch die Parameter, die mit update(8) einstellbar sind, ignoriert.

Der Versuch, einen hochverfügbaren SAMBA-Server zu implementieren, zeigt deutlich, daß das SMB-Protokoll nicht stateless ist. Bei mehreren Versuchen,

[7]Das ext2-Filesystem unterstützt die Option -sync, die beim mount-Kommando angegeben werden kann.

[8]Das kann ein User-Prozess mit fdatasync(2) oder fsync(2).

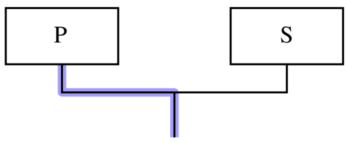
einen Failover während eines Schreibzugriffs durchzuführen, ist der Client, der mit Windows 98 installiert war, abgestürzt.

6.3.3 Heartbeat Erweiterungen

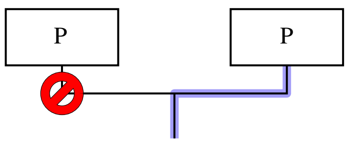
Im Zuge der Auseinandersetzung mit Heartbeat wurde auch deutlich, daß für einen zuverlässigen Produktionseinsatz Heartbeat erweitert werden muß. Vor allem transiente Netzwerkfehler können mit dem heutigen[9] Heartbeat zu katastrophalen Fehlern führen.

Hier ein Beispiel, in dem die Transaktionseigenschaften verloren gehen:

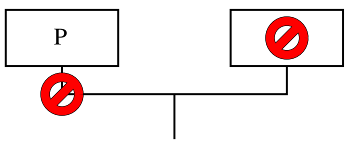
1. Es handelt sich um einen Cluster bestehend aus zwei Knoten und nur einer Netzwerkverbindung.



2. Fällt die Netzwerkverbindung zum primären Server temporär aus, übernimmt der andere Knoten das Service.

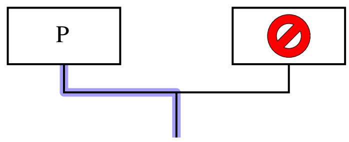


3. Nun fällt der zweite Server aus. Der Cluster kann das Service jetzt nicht mehr anbieten.



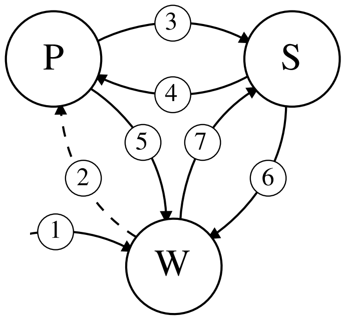
4. Ein Anwender, dessen Arbeit durch den Totalausfall unterbrochen wurde, behebt den temporären Netzwerkausfall. Nun ist das Service zwar wieder verfügbar, die Veränderungen, die die Transaktionen in Phase 2 vorgenommen haben, sind aber verloren!

[9]Version 0.4.7



Um dieses Fehlverhalten zu verhindern, müßte Heartbeat auf einem isolierten Server in einen Zustand übergehen, in dem kein Service laufen kann. Diesen Zustand kann Heartbeat erst dann wieder verlassen, nachdem es seinen Datenstand an den aktuellen Datenstand des Clusters anpassen konnte.

Der folgende Zustandsgraph müßte für jedes Service vorhanden sein:



P Primary In diesem Zustand läuft das Service.

S Secondary Das Service läuft nicht. Der Knoten, der das Service anbietet, wird überwacht, und die Daten, die er schreibt, werden auf diesen Knoten gespiegelt.

W Waiting Da der aktuelle Stand der clusterweiten Daten nicht bekannt ist, wird vom Knoten, der gerade das Service anbietet, der aktuelle Datenstand bezogen.

Übergänge:

1 Wenn Heartbeat startet, geht es zuerst in den Zustand W über.

2 Wenn ein Cluster zum ersten Mal gestartet wird, gehen alle Knoten in den Zustand W. Der Administrator hat die Möglichkeit, einen der Knoten in den Zustand P überzuführen.

3 Wenn ein anderer Knoten das Service übernehmen will, geht dieser Knoten in den Zustand S über.

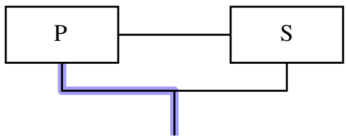
4 Der primäre Knoten eines Services ist ausgefallen.

5, 6 Der Knoten ist vom Netzwerk isoliert, d.h. er kann weder einen anderen Knoten des Clusters, noch Netzwerkressourcen, von denen bekannt ist, daß sie normalerweise sichtbar sind, erreichen.

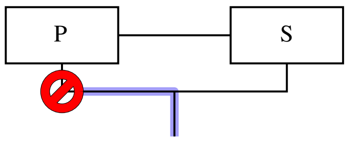
7 Die Aufsynchronisation des Datenstandes mit dem Datenstand des aktuellen primären Knotens war erfolgreich.

Ein weiteres Beispiel:

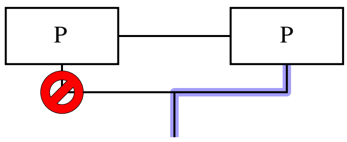
1. Eine Konfiguration mit zwei Netzwerkkarten; Heartbeat sind beide Netzwerkverbindungen bekannt.



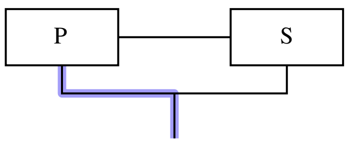
2. Da die Heartbeat-Instanz auf dem sekundären Knoten den primären Knoten immer noch erreichen kann, kommt es zu keinem Failover. Das Service ist nicht mehr erreichbar.



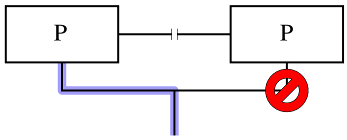
- Würde man Heartbeat nur das Client-Netzwerk bekanntgeben, würde der Failover funktionieren.



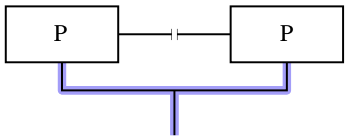
1. Zwei Knoten. Heartbeat kennt nur das Client-Netzwerk.



2. Nun kommt es beim sekundären Knoten zu einem temporären Netzwerkausfall. Da dieser nun keine Heartbeat-Pakete mehr mit dem primären Knoten austauschen kann, startet Heartbeat die Services ebenfalls auf diesem Knoten. DRBD bricht die Verbindung ab, da es auf beiden Knoten im primären Zustand ist.



3. Wenn der temporäre Netzwerkausfall wieder verschwindet, wird vermutlich nichts mehr funktionieren, da jetzt beide Knoten ARP-Pakete aussenden, um die virtuelle IP-Adresse auf ihre MAC-Adresse abzubilden.



Um dieses Problem zu beseitigen, müßte Heartbeat eine Menge von normalerweise sichtbaren Ressourcen (wie z.B. Router, Gateways, ...) überwachen. Kann Heartbeat keine dieser Ressourcen mehr erreichen, muß es sich bei anderen Knoten des Clusters erkundigen, ob diese noch die Ressourcen erreichen können. Wenn diese die Ressourcen noch sehen können, muß der Knoten davon ausgehen, daß seine Netzwerkkarte oder seine Netzwerkanbindung ausgefallen ist. Alle Services müssen abgegeben werden, der Knoten kann aber die Spiegelung von Daten weiterhin durchführen.

Kapitel 7

Zusammenfassung

HA-Cluster, die mit dem Prinzip des Failovers arbeiten, konnten bereits früher unter Linux implementiert werden. Bis jetzt war es allerdings nicht möglich, Services anzubieten, die ihren Datenstand dynamisch verändern (z.B. Fileserver, Datenbankserver, ...), da die in Abschnitt 2.3.3.1 (Seite 12) vorgestellten Shared SCSI-Devices unter Linux (noch) nicht implementiert sind.

Mit DRBD, das das Spiegeln von Festplatten über IP-basierte Netzwerke erlaubt, wird die Möglichkeit geschaffen, solche Services mit einem HA-Cluster unter Linux anzubieten.

Im Zuge der Entwicklung von DRBD wurde ein Algorithmus entwickelt, der dem Disk-Scheduler beim Schreiben die größtmögliche Freiheit einräumt, Blöcke umzuordnen, dabei aber die Reihenfolge, die das Filesystem vorgibt, nicht verletzt (siehe Abschnitt 3.3.1 auf Seite 24).

Der Durchsatz eines DRBD-Gerätes wird natürlich durch den Durchsatz der beteiligten Festplatten und den Durchsatz des Netzwerkes beschränkt. In Versuchen konnte ein Durchsatz zwischen 50 % und 98 % des maximal möglichen Wertes erreicht werden (siehe Abschnitt 5.4 auf Seite 55).

Sowohl die Cluster-Management-Software als auch die neuen Filesysteme stecken noch in ihren Kinderschuhen, doch die ständig präsente Nachfrage nach HA-Lösungen wird die Entwicklungen auch in Zukunft vorantreiben.

Da PC-Hardware wesentlich billiger ist und DRBD auch die kostspielige Hardware für Shared Disks ersetzt, könnten sich HA-Clustern auf der Basis von Linux neue Anwendungsgebiete eröffnen, die ihnen bisher wegen der hohen Kosten verwehrt waren.

Ob sich DRBD als Alternative zu Shared Disks behaupten können wird, wird sich noch herausstellen. Dies wird zu einem Großteil davon abhängen, ob DRBD von den Linux-Distributoren in ihre Distributionen aufgenommen wird; und genau das zeichnet sich bereits ab. DRBD wird als Bestandteil der nächsten Connectiva-Linux-Distribution erscheinen, und auch SuSE zeigt Interesse.

Doch DRBD wird sicher nicht die letzte interessante Entwicklung auf dem Gebiet des Clusterings sein. Vor allem Anbieter von Web-Applikationen bauen Cluster, bei denen die Netzwerklast auf mehrere Knoten verteilt wird. Für diesen Cluster-Typ bräuchte man ein verteiltes Filesystem, das sowohl Redundanz, hohe Performance, Sperren von Dateien und simultanen Zugriff von allen Knoten des Clusters aus erlaubt.

Anhang A

Meßergebnisse

Die Angaben in den Spalten Festplatte, DRBD n. v., Prot. A, Prot. B und Prot. C sind in MB/s.

1.

(a)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.12-20	i386	265,42	2,27	1,99	1,3	1,29	1,51
2	Linux	2.2.12-20	i386	265,42	2,23	2,04	1,3	1,29	1,59

Netzwerkbandbreite: 4,33 MB/s — Datengröße: 10 MB
Netzwerklatenz: 0,1/0,8/36,4 ms — DRBD Version: 0.5.3

(b)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.12-20	i386	265,42	2,26	1,96	1,34	1,29	1,48
2	Linux	2.2.12-20	i386	265,42	2,25	1,99	1,31	1,3	1,49

Netzwerkbandbreite: 9,7 MB/s — Datengröße: 100 MB
Netzwerklatenz: 0,1/0,8/34,3 ms — DRBD Version: 0.5.3

2.

(a)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.13	alpha	528,48	4,49	4,68	0,7	0,73	0,65
2	Linux	2.2.13	i586	47,82	2,09	2,11	0,95	0,94	0,91

Netzwerkbandbreite:	0,95 MB/s	Datengröße:	10 MB
Netzwerklatenz:	0,5/0,5/0,8 ms	DRBD Version:	0.5.3

(b)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.13	alpha	528,48	4,94	4,81	0,74	0,74	0,73
2	Linux	2.2.13	i586	47,82	2,05	1,16	0,95	0,94	0,93

Netzwerkbandbreite:	1,04 MB/s	Datengröße:	55 MB
Netzwerklatenz:	0,5/0,5/0,7 ms	DRBD Version:	0.5.3

3.

(a)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.14-9cl	i586	897,84	11,17	10,21	3,67	3,59	3,56
2	Linux	2.2.14-14cl	i586	799,54	8,66	7,8	3,8	3,69	3,99

Netzwerkbandbreite:	8,6 MB/s	Datengröße:	100 MB
Netzwerklatenz:	0,0/0,0/0,3 ms	DRBD Version:	0.5.3

(b)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.14-9cl	i586	897,84	14,88	12,9	4,94	4,68	4,57
2	Linux	2.2.14-14cl	i586	799,54	10,97	9,11	5,35	5,3	5,2

Netzwerkbandbreite: 8,7 MB/s — Datengröße: 100 MB
Netzwerklatenz: 0,0/0,0/0,2 ms — DRBD Version: 0.5.3

(c)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.14-9cl	i586	897,84	15,79	14,18	1,06	1,06	1,01
2	Linux	2.2.14-14cl	i586	799,54	11,34	9,83	0,87	0,92	0,87

Netzwerkbandbreite: 0,94 MB/s — Datengröße: 100 MB
Netzwerklatenz: 0,3/0,3/0,5 ms — DRBD Version: 0.5.3

4.

(a)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.14	i686	498,07	11,23	10,4	6,63	6,28	5,63
2	Linux	2.2.14	i586	466,94	6,76	6,06	6,26	6,46	5,4

Netzwerkbandbreite: 11,1 MB/s — Datengröße: 100 MB
Netzwerklatenz: 0,0/0,0/0,2 ms — DRBD Version: 0.5.3

(b)

Knoten	System	Version	Architektur	BogoMips	Festplatte	DRBD n. v.	Prot. A	Prot. B	Prot. C
1	Linux	2.2.14	i686	498,07	11,3	9,65	6,14	6,18	5,29
2	Linux	2.2.14	i586	466,94	6,62	6,34	6,27	6,27	6,08

Netzwerkbandbreite:	11,1 MB/s	Datengröße:	100 MB
Netzwerklatenz:	0,0/0,0/0,2 ms	DRBD Version:	0.5.3

Literaturverzeichnis

[BBDK97] Michael Beck, Harald Böhme, Mirko Dzadzka, Ulrich Kunitz, Robert Magnus, Dirk Verworner. *Linux-Kernel-Programmierung Algorithmen und Strukturen der Version 2.0*, 4. Auflage. Addison Weseley Longman, Bonn 1997.

[HHMK94] Sebastian Hetze, Dirk Hohndel, Martin Müller, Olaf Kirch. *LinuX Anwenderhandbuch und Leitfaden für die Systemverwaltung.* 4. erweiterte und aktualisierte Auflage. LunetIX Softfair 1994.

[Hor98] Simon Horman. *Creating Redundant Linux Servers.* http://www.us.vergenet.net/linux/redundant_linux_paper-/talk/redundant_linux.ps.bz2

[Kop97] Hermann Kopetz. *REAL TIME SYSTEMS Design Principles for Distributed Embedded Applications.* Kuwler Academic Publishers, Boston, 1997.

[Lew99] Phil Lewis. *A High-Availability Cluster for Linux. Linux Journal*, Issue 64, August 1999. Specialized Systems Consultants, Seattle.

[Mil98] Harald Milz. *Linux High Availability HOWTO.* http://metalab.unc.edu/pub/Linux/ALPHA/linux-ha/High-Availability-HOWTO.html

[MPI97] *MPI-2: Extensions to the Message-Passing Interface.* 1997 University of Tennessee, Knoxville, Tennessee. http://www.mpi-forum.org/docs/mpi-20.ps.Z

[Pfi98] Gregory F. Pfister. *In search of Clusters.* Prentice-Hall PTR, Upper Saddle River 1998.

[Reis99] Hans Reiser. *ReiserFS.* http://www.devlinux.com/project/reiserfs/res_whol.shtml

[Rob00] Alan Robertson. *High-Availability Linux Project.* http://linux-ha.org

[PBB+99] Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland und Matthew T. O'Keefe. *A 64-bit, shared disk File System for Linux.* Proceedings of the Sixteenth IEEE Mass Storage Systems Symposium, March 1999. (Alternativ http://www.globalfilesystem.org/pubs/NASA_GFS_1999.ps)

[Rub98] Alessandro Rubini. *Linux Device Drivers.* O'Reilly & Associates, Inc. Sebastopol 1998.

[Rub97] Alessandro Rubini. *The virtual filesystem in Linux.* Linux Journal, Issue 37, May 1997. Specialized Systems Consultants, Seattle.

[Rus98] David A Rusling. *The Linux Kernel.* http://www.linuxdoc.org/LDP/tlk/tlk.html

[Russ00] Rusty Russel. *Linux Kernel Locking HOWTO.* http://netfilter.kernelnotes.org/unreliable-guides/kernel-locking-HOWTO.html

[Sha00] Rawn Shah. *Linux clustering cornucopia.* http://www.linuxworld.com/lw-2000-03/lw-03-clustering.html

[Sta95] William Stallings. *Operating Systems*, second edition. Prentice-Hall, Englewood Cliffs, New Jersey 1995.

[Sun90] V. S. Sunderam. *PVM: A Framework for Parallel Distributed Computing.* Journal of Concurrency: Practice and Experience, 2, 4, Seiten 315-339, Dezember 1990. http://www.netlib.org/ncwn/pvmsystem.ps.

[Twe98] Stephen C. Tweedie. *Journaling the Linux ext2fs Filesystem.* LinuxExpo 1998. ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz

[Vol97] Yuri Volobuev. *Playing redir games wirh ARP and ICMP.* http://www.rootshell.com

[Weis97] Peter Weiss. *Dynamische Treiber unter Linux.* IX Magazin, Januar 1997. Verlag Heinz Heise, München.

[Wey96] Peter S. Weygant. *Clusters for High Availability.* Prentice Hall PTR, Englewood Cliffs, New Jersey 1996.

[ZJW99] Wensong Zhang, Shiyao Jin, Quanyuan Wu. *Creating Linux Virtual Servers.* Proceedings of the 5th Annual Linux Expo, pages

101-110. Webcom, Technical Documentation Division. (Alternativ http://www.linuxvirtualserver.org/clvs.ps.gz)