Linux Hochverfügbarkeit

Einsatzszenarien und Praxislösungen für Linux-Server

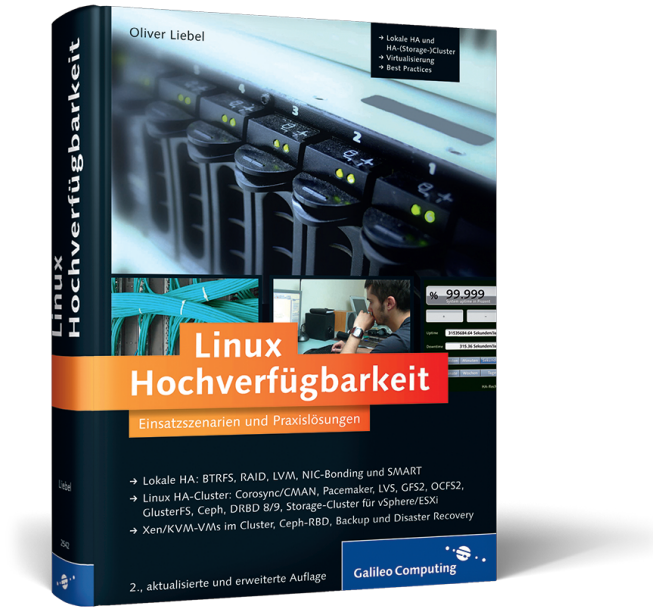
Bearbeitet von
Oliver Liebel

erweitert 2013. Buch. 848 S. Hardcover
ISBN 978 3 8362 2542 7
Format (B x L): 16 x 24 cm

Oliver Liebel

Linux Hochverfügbarkeit

Einsatzszenarien und Praxislösungen



Galileo Press

Auf einen Blick

Inhalt

14 Cluster-Beispiele aus der Praxis 377

Vorwort

Linux Hochverfügbarkeit – »Reloaded«

Ja, nee – is' klar. »Reloaded«..., das kennen wir schließlich noch aus »Matrix«. Aber im Gegensatz zu einer in unnötigen Sequelen wiederaufbereiteten virtuellen Welt, die im Kern nur mit einer höheren Schlagzahl an pseudo-philosophischen Dialogen und n-Mal mehr Agents Smith' ausgestattet war, handelt es sich in diesem Fall um eine neue, deutlich erweiterte und komplett überarbeitete Ausgabe meines HA-Buches, das in den letzten Jahren viele Unternehmen auf dem Weg zur Sicherstellung ihrer Hochverfügbarkeit erfolgreich begleitet hat.

Die Gründe, warum dieser umfassende »Reload« – bei dem allein die Kapitel über HA-(Storage-)Clustering mit zusammengenommen rund 520 Seiten nun fast den dreifachen Umfang haben und für sich allein genommen schon deutlich umfangreicher sind als die komplette letzte Ausgabe dieses Buches – mehr als notwendig war?

Ganz einfach: Viele HA-relevante Komponenten, wie zum Beispiel BTRFS im Bereich der lokalen HA, oder auch Pacemaker und Corosync im Bereich des HA-Clusterings haben in den letzten drei Jahren einiges dazugelernt. Neben der famosen *crm*-Shell existiert nun mit *pcs* ein brandneues Admin-Tool für Corosync und Pacemaker. Und im Sektor des hochskalierbaren Distributed Storage Clustering ist z. B. neben GlusterFS mit »Ceph« ein (nur namentlich aquamariner) Kopffüßler hinzugekommen, der mit seinen Features jede Menge hohe Wellen schlägt.

Dazu kommen etliche neue Konzepte und Setup-Szenarien aus der Praxis, die ich nach der Veröffentlichung der letzten Ausgabe im Rahmen von Workshops und Beratungen für Kunden aus dem Mittelstand, der Großindustrie, dem Bankensektor, dem Bund selbst sowie auch international agierender Konzerne umgesetzt habe, und die sicher für viele Administratoren in der dargestellten oder ähnlichen Form praxisrelevant sind.

Aber zunächst, wie beim letzten Mal, ein kurzer Blick auf den Kern unserer Betrachtungen, und dem kleinen Wörtchen mit der große Bedeutung. *HA – High Availability – Hochverfügbarkeit*. Ein Wort mit mächtig hohem Anspruch. Ein Anspruch, den sich viele Unternehmen auf die Fahne schreiben, der jedoch bei genauerer Betrachtung leider oft genug Lücken im Detail aufweist. Anderen fehlen oftmals einfach die technischen und personellen Ressourcen, das Know-how und damit schlichtweg die Einarbeitungszeit, um überhaupt grundlegende Sicherheitsmaßnahmen zur Prävention ihrer Systeme zu implementieren. Oft genug mit fatalen Konsequenzen.

Ein unterbrechungsfreier Betrieb kann im Extremfall entscheidend für den Fortbestand eines Unternehmens sein. Längere Ausfallzeiten, die aus fehlerhafter bzw. nicht vorhandener Redundanz resultieren, noch dazu kombiniert mit etwaigen Datenverlusten nach einem Crash, können – je nach Größenordnung – sogar das komplette Aus für ein Unternehmen bedeuten.

Hochverfügbarkeit war und ist dabei nicht nur allein ein Punkt, der von rein technischen Aspekten bzw. Defekten abhängt. In der Praxis spielen viele Faktoren eine Rolle, die die Verfügbarkeit eines Systems gravierend beeinflussen können: angefangen bei den gerade genannten technischen Defekten über menschliches Versagen oder gar Naturkatastrophen, ständig steigende Komplexität und wachsende Datenvolumen der Systeme bis hin zu gezielten Attacken auf genau jene Systeme – und das ist nur ein Teil der Faktoren, die jedem Admin den Tag vermiesen können.

Um Hochverfügbarkeit zu gewährleisten, benötigen wir in der Praxis vor allem eine sorgfältige Planung, bevor es losgeht: redundante Hardware-Komponenten in unseren Servern, ausfallsichere Services, sowie Prozeduren zur lückenlosen Überwachung der Systeme, die sich in logischer Konsequenz zu einem Gesamtpaket ergänzen, das eine maximale Ausfallsicherheit bietet. Und da kein isoliertes, technisches System auf diesem Planeten eine hundertprozentige Ausfallsicherheit garantieren kann, müssen wir – mit entsprechend gut dokumentierten und vor allem regelmäßig getesteten Notfall-Prozeduren und zugehörigen Tools – auf den Fall der Fälle stets so gut wie möglich vorbereitet sein. Warum? Ganz einfach – *alles was schiefgehen kann, wird irgendwann schiefgehen*. Wetten? Fragt den guten alten Murphy – oder lest einfach jetzt schon einmal die das Intro des letzten Abschnitts dieses Buches über Backup und Disaster-Recovery.

In den Anfängen meines Jobs Mitte der Neunziger drehte es sich zumeist noch um einfache Fileserver, deren Platten in einem einfachen Software-RAID-SFT3-Mirror (*Security Fault Tolerance Level 3* nannte sich das einfache, aber robuste Spiegel-Konstrukt zu Novell-3.11-Zeiten) zusammengefasst werden konnten. Je weiter die Zeit fortschritt, desto komplexer wurden – aufgrund der an sie gestellten Anforderungen – auch die Systeme. Zur lokalen Hochverfügbarkeit einfacher Fileserver kamen verteilte und redundant vorzuhaltende Verzeichnisdienste und schließlich ausfallsichere Cluster mit verschiedensten Storage-Konzepten.

Und nicht immer waren Konzepte, Software und/oder Hardware dabei wirklich ausgereift – wie auch heute noch viel zu oft. Wenn ich zurückblicke, fallen mir einige Episoden aus den frühen Tagen ein, die man rückblickend nur sehr wohlwollend unter der Rubrik »Wie macht man sein Leben spannender« ablegen kann. Wie die mit dem Micropolis Level 5 Hardware-Raid, von dessen fünf Platten (inklusive zwei Spares) sich drei innerhalb von nur zehn Stunden verabschiedeten. Eigentlich eine logische

Konsequenz, bei qualitativ absolut miserabler Hardware, deren Disks noch dazu aus exakt ein und derselben Baureihe stammten.

Die Nacht war endlos lang und das Sodbrennen vom vielen Kaffee nicht wirklich schön, aber bevor sich die letzte der Platten verabschiedete, die den Raid-Verbund noch aufrechthielt, konnte ich mit Ach und Krach gerade noch die letzten Bits auf einen anderen, notdürftig zusammengestrickten Ersatz-Server schaufeln. Und das Backup? Ja, sicher … das gute, alte Backup. Es war natürlich auf den Bändern, wie es sich gehört. Nur leider in den letzten zwei Monaten ungeprüft, sonst hätte ich damals sehr wahrscheinlich nicht erst in jener Nacht bemerkt, dass trotz unauffälliger Logs die Daten durch einen mechanischen Fehler im Streamer beschädigt und nicht mehr zu gebrauchen waren. Wie so oft im Leben sind es immer die kleinen Dinge, die richtig Freude bereiten.

Danach schwor ich mir, jedes System so optimal wie möglich vor Ausfällen zu schützen – und es im Ausfall-Fall so schnell wie möglich wiederherstellen zu können. Eine aufwändige Angelegenheit, klar. Aber ein ausgefallenes System ohne Redundanz und entsprechende Backups, Prozeduren und *Disaster-Recovery*-Tools verleiht dem Begriff »aufwändige Angelegenheit«, was die Wiederherstellung angeht, ganz locker eine völlig neue Dimension. Und jedem Admin sehr schnell ein sehr persönliches, sehr unerfreuliches und sehr lautes Gespräch mit der Chefetage.

Leider wird sich kaum jeder Leser dieses Buches in der glücklichen Lage befinden, bereits mit einem Multi-Node-Cluster und redundanten High-End-SAN oder -NAS gesegnet zu sein, die je nach Typ schon aus redundanten RAID-Leveln bestehen und ebenfalls über eine redundante Anbindung verfügen.

Und genau das ist der Punkt: Dieses Buch kann – und wird, wie beim letzten Mal – keine speziellen Hardware-Lösungen berücksichtigen; dies ist das Spezialgebiet von kommerziellen Storage-Anbietern, die kostspielige High-End-Lösungen auf diesem Sektor zur Genüge anbieten. Gleiches gilt für proprietäre Cluster-Stacks und -Dateisysteme, und kommerzielle Backup-/Disaster-Recovery- und Virtualisierungslösungen. Die einzige Ausnahme bei letzteren bildet hier (nicht zuletzt aufgrund des mittlerweile sehr hohen Verbreitungsgrades und der sehr oft anzutreffenden Kombinationen, s. u.) das Thema VMware bzw. ESXi/vSphere, das ich in verschiedenen, praxistypischen und -erprobten Szenarien ergänzend vorstelle, so z. B. im Bereich der sehr wichtigen STONITH-Thematik virtueller Pacemaker-Cluster auf vSphere-Hosts, oder auch von realen Pacemaker-Storage-Cluster-Systemen, die wiederum in Verbindung mit DRBD oder Distributed File Systems (wie z. B. Ceph/GlusterFS) und iSCSI oder NFS als hochskalierbare und hochverfügbare Storage-Backends für eben jene ESXi/vSphere/vCenter-Umgebungen dienen können.

Zudem noch ein wichtiger Hinweis an dieser Stelle: Ich beziehe mich in diesem Buch – aus Gründen der Stabilität, des Planungshorizontes und der dadurch längerfristi-

gen Reproduzierbarkeit – primär auf die Enterprise-/LTS-(Long Term Support-)Systeme/Versionen der großen Distributoren. Diese erfordern – natürlich gerade beim Einsatz in Produktivumgebungen – in der Regel eine kostenpflichtige Lizenzierung, allein schon hinsichtlich des Supports und/oder der Upgrades/Patches, sind aber in der Regel dennoch frei verfügbar und zeitlich uneingeschränkt einsetzbar (wenn auch, wie z. B. im Fall von SUSE bzw. dem SLES, nach Ablauf des Testzeitraums ohne Support/Patches). Das proprietäre RHEL stellt hier einen Grenzfall dar, für den uns jedoch alternativ das voll binärkompatible CentOS kostenlos zur Verfügung steht. Aber: Was klar außen vor bleibt, ist nach wie vor *proprietäre Closed Source*, wie z. B. Symantecs VCS (*Veritas Cluster Server*) und ähnliches. Es geht hier um Open Source – *offene* HA Lösungen, und im Fall bzw. der Ausnahme von VMware – wie bereits zuvor kurz erläutert – primär um die Storage-Anbindung an diese (bzw. den Einsatz von virtualisierten Clustern auf diesen) Systemen.

Mir geht es zum einen primär wieder darum, auf der Basis von praxiserprobten Szenarien, bewährten Standard-Tools und Software-Lösungen aus dem Open-Source-Umfeld – auch in »normalen« Server-Welten, in denen mit halbwegs moderaten Budgets und/oder Hardwarelösungen gearbeitet werden muss – jederzeit ausfallsichere und hochverfügbare Server bereitstellen zu können. Lösungen, die es in einigen Belangen durchaus mit kommerziellen Produkten aufnehmen können und ihnen in anderen zum Teil sogar überlegen sind.

Zum anderen geht es mir ebenso wieder darum, grundlegende Konzepte zu vermitteln. Ein stures Abarbeiten der erlernten Prozeduren A, B und C zur Sicherstellung der Hochverfügbarkeit mag auf System X völlig ausreichend sein, System Y benötigt vielleicht jedoch eine ganz andere Weise des Herangehens. Es geht darum, eine gewisse Sensibilität für neuralgische Punkte zu entwickeln, die eine Schwachstelle innerhalb der Hochverfügbarkeitslösung bilden könnten.

Das Ganze gilt insbesondere unter der Prämisse, dass auch die eingesetzten Software-Lösungen einem ständigen Evolutionsprozess unterworfen sind – und das leider nicht immer nur zum Positiven hin, wie wir sehen werden und wie die Erfahrung zeigt. Und daher stellt eine extrem sorgfältige Evaluierung und Abwägung des Für und Wider eines bestimmten Konzepts bzw. der korrespondierenden HA-Lösung *vor ihrer Implementierung* einen der wichtigsten Punkte überhaupt dar. Denn eine bestehende HA-Infrastruktur im Nachhinein *mal eben* (ohnehin eine der bei den Admins beliebtesten Redewendungen in der IT) wieder komplett umzukrempeln, dürfte erfahrungsgemäß in den wenigsten Fällen realisierbar sein, geschweige denn auch nur annähernd im Budget liegen.

Aufgrund der fortschreitenden Entwicklung bringt eine alleinige Fokussierung auf ganz bestimmte Versionen bestimmter Tools und Pakete – die wir im Folgenden natürlich dennoch benötigen, um praxisnahe Setups konkret zu erläutern – nur

unter dem Aspekt etwas, dass die dahinterliegenden theoretischen Basics immer sorgfältig verinnerlicht werden. Nur so lassen sich im Hier und Jetzt anwendbare und angewendete Verfahren auf die nächste Evolutionsstufe portieren. Linux bringt eine gewaltige Palette an Tools mit, die es uns erlauben, unsere Systeme hochverfügbar zu halten, aber das am Ende vorliegende Gesamtkonstrukt ist immer nur so gut oder schlecht wie das ihm zugrunde liegende Konzept. Und deswegen werden wir vor jedem neuen Abschnitt zunächst die theoretischen Basics erörtern, dann mögliche Konzepte und Ansätze für Lösungsstrategien unter die Lupe nehmen, und die am besten geeigneten konkret anhand von praxiserprobten Setups verifizieren.

Ein Begriff ist und bleibt dabei aber immer der Kernpunkt unserer Betrachtungen: *Redundanz*. Und hierbei reden wir nicht nur von Festplatten, Netzteilen oder Netzwerkkarten, sondern von ganzen Maschinen und ihren Services, die per Clustering und vollautomatisches Monitoring ausfallsicher gehalten werden, das Ganze im idealen Fall natürlich noch ergänzt über netzwerkweit redundante Storage-Lösungen wie z. B. DRBD und/oder Ceph.

Im Folgenden werden wir daher Schritt für Schritt alle erforderlichen Punkte und Konzepte detailliert betrachten, die die Redundanz und Hochverfügbarkeit unserer Systeme – zunächst auf lokaler Server-Ebene und später im Netzwerk bzw. Cluster – sicherstellen, und sie anschließend zu einem hochfunktionellen und sicheren Paket zusammenschnüren, das uns im Endeffekt die größtmögliche Ausfallsicherheit bietet.

Ich habe wiederum versucht, mit diesem Buch und den aktuellsten, verfügbaren HA-relevanten OpenSource-Produkten und Konzepten einen einfachen, aber dennoch möglichst kompletten, und vor allem stets reproduzierbaren Einstieg in das Thema zu ermöglichen; ebenso wie die Umsetzung komplexerer und vor allem praxisorientierter Setup-Szenarien. Das Ganze gehe ich – wie in meinen anderen Publikationen – auch dieses Mal wieder Schritt für Schritt anhand von praktischen Beispielen durch, und zwar exakt mit den von der Verlagsseite herunterladbaren Konfigurationsdateien (*http://www.galileocomputing.de/3420*), die ihre Funktionalität auf meinen Testsystemen und etlichen Produktivumgebungen in der Praxis unter Beweis gestellt haben. Aber schließlich bin ich auch nur ein Mensch, und daher – so wie wir alle – sicherlich nicht perfekt. Sollten sich daher also Fehler in bestimmten Konfigurationen eingeschlichen haben, oder sollte ich an irgendeiner Stelle im Buch ein Detail übersehen haben: Geben Sie mir ein kurzes Feedback, damit ich reagieren und eine fehlerbereinigte Version des entsprechenden Kontextes über die Website des Verlages zur Verfügung stellen kann.

Die Anforderungs- und Themenliste, die diesem Buch zugrundeliegt, entstammt wiederum ebenfalls – so wie bei meinen anderen Publikationen – primär den Fragen, Themen und Problematiken, die mir im Rahmen von Beratungs-Aufträgen, Praxis-

Workshops und Schulungen von den Administratoren genannt wurden; ebenso den Anregungen der Rezensenten und Administratoren zur letzten Ausgabe. Daher soll auch dieses Buch vor allem eines sein: ein praxisorientierter Leitfaden für hochverfügbare Systeme unter Linux, und zwar auf der Basis von zahlreichen, praxiserprobten Szenarien und Setups. Ein Leitfaden, der dem Administrator beim Setup des Systems, und vor allem auch bei der Lösung von real anfallenden Problematiken helfen kann. Ich hoffe, dass ich die Aufgabe wiederum so gut wie möglich gelöst habe.

Schon der Vorgänger dieses Buches hatte einen sehr langen Weg hinter sich, und genau genommen sind es mittlerweile nunmehr fast 20 Jahre. Was mit losen Aufzeichnungen aus meinen Anfängen als Administrator begann, verdickte sich im Laufe der Zeit zu kompletten Manuals, und neben Seminarleitfäden für HA-Schulungen im Enterprise-Segment entstanden nach und nach immer umfangreichere und komplexere Howtos, die wiederum verschiedensten, realen Projekten mittelständischer Unternehmen, der Großindustrie und international agierender Konzerne, Bankengruppen, von Städten und Institutionen auf kommunaler-, Landes- und Bundes-Ebene entstammten.

Nun denn – letztlich ging es mir wieder vor allem um eines: Dass ich Ihnen *die* Nachtschichten ersparen kann, aufgrund derer dieses Buch existiert. Also, gehen wir's an – Runde 2 ...

Oliver Liebel

Danksagungen

Mein ganz spezieller Dank geht auch dieses Mal – jedoch nun leider posthum – an meinen Vater, der mir das Redundanzprinzip schon als Teenager anschaulich demonstrierte, während er im knietiefen Wasser unseres vollgelaufenen Kellers hart daran arbeitete, die einzige, defekte Pumpe wieder in Gang zu bekommen – seit jener Nacht hatten wir zwei – und dazu einen Generator, die allesamt (über die Jahre natürlich »upgegraded«) bis zum heutigen Tage brav ihren Dienst verrichten. Und obwohl er erst fast 30 Jahre später damit begonnen hatte, sich mit einem Computer (natürlich mit einem Linux-OS, was sonst) auseinanderzusetzen, war gerade die Ausfallsicherheit ein Thema, das ihn bis zuletzt immer fasziniert hatte. Und auch einer seiner letzten »HA«-Ratschläge für eine zweite, zusätzliche Regenkombi, hat sich als richtig erwiesen, und mich auf meiner Motorrad-Tour im Sommer 2011 zumindest vor den gröbsten Auswirkungen des – den schottischen Highlands ganz eigenen – »Lovely Weathers« bewahrt, das doch mit deutlich größeren submarinen Qualitäten glänzte als erhofft. Vielen Dank für die Geduld und die Unterstützung durch meine Freunde

während der ganzen Zeit, und für ihr Verständnis für die eine oder andere ausgefallene Motorradtour oder gesellschaftliche Verpflichtung, der ich in dieser Zeit einmal mehr aufgrund meines extrem zeitaufwändigen »Zweitjobs« als Autor nicht nachkommen konnte. Mein besonderer Dank geht an meinen alten Freund Hanspeter Schulze, der meinen Weg in die IT-Welt maßgeblich mit bereitet hat. Und last but not least möchte ich mich wie immer beim Galileo-Team bedanken, und insbesondere meinem Lektor Sebastian Kestel, mit denen ich nun mittlerweile das vierte Buchprojekt auf den Weg gebracht habe.

Ich widme dieses Buch dem Andenken an meinen im Herbst 2012 verstorbenen Vater *Georg Liebel*. Mach's gut, alter Bluesman ...

Kapitel 1
Mission Critical – ausfallsichere Server

»He, Ridley?«
»Ja?«
»Hast Du noch 'nen Streifen Kaugummi für mich? Ich leih' ihn mir nur. Ich geb ihn Dir nachher wieder.«
»Na klar, Chuck.«
– The Right Stuff, USA 1980

Wie sich an dem realen – vor über 60 Jahren und danach noch sehr oft geführten – Kurzdialog zwischen dem legendären Testpiloten, der als erster die Schallmauer durchbrach, Air Force Captain Charles »Chuck« Yeager, und seinem Flugingenieur und Freund Jack Ridley, grob erahnen lässt, geht es um eine Mission, bei der der gute alte Chuck wirklich hervorragende Chancen hatte, nicht zurückzukommen. Aber eine andere Antwort hätte ihm sein Freund nie gegeben.

Was uns angeht und die hochverfügbaren Systeme, die wir konzeptionieren, aufbauen und verwalten müssen, so mag die Mission vielleicht nicht unbedingt so potentiell lebensverkürzend sein wie die unzähligen von Chuck, aber: je nachdem *was* unsere Pacemaker HA-Cluster hosten, geht es oft genug um hochsensible Daten und kritische Anwendungen – und ein Ausfall der Systeme kann fatale Folgen haben. *So kritisch?*

Yepp. Denn da wir gerade schon mal in der Luft bei Chuck waren: das »*Phoenix*«-Radar/Data – ATC- (Air Traffic Control-)System der DFS, der Deutschen Flugsicherung, arbeitet auf allen internationalen, deutschen Flughäfen auf der Basis von SUSE Linux Enterprise Servern mit höchstverfügbaren Pacemaker-Clustersystemen.

Da wir jetzt einen der vielen, realen Praxisbezüge kennen und uns grob vorstellen können, welche hohen, lebenswichtigen Anforderungen ein HA-System je nach Einsatzzweck erfüllen muss, können wir uns mit voller Aufmerksamkeit den theoretischen Grundlagen, der Konzeption und der Umsetzung widmen. Damit wir uns niemals in einem Szenario wiederfinden, das einigen leidgeprüften Admins schon widerfahren ist. Etwas, das wir genau aus diesem Grund daher auch nur zu gern aus unseren Gedanken verbannen: Der firmentechnische IT-Super-GAU – angefangen

vom einfachen Service-Crash bis zum Totalausfall aller unternehmenskritischen Systeme.

Und das betrifft nicht nur die Admins kleinerer und mittelständischer Betriebe – auch ganze Serverfarmen in größeren Rechenzentren wurden schon Opfer von Bauteilen im Cent-Bereich, die während der Planung eben nicht redundant ausgelegt wurden. Gibt's nicht? Von wegen ...

1.1 Grundsätzliche Überlegungen zur Redundanz

Wir Admins wissen: Nichts ist absolut sicher. (Gut, die stete Zusage alle Politiker vor der Wahl, Steuern zu senken, und sie danach deftig zu erhöhen, sei hier mal außen vor gelassen.) Denn neben dem Ausfall einzelner Software-Dienste kann es im Hardware-Bereich eines jeden Servers jederzeit zu einem Ausfall einer oder mehrerer vitaler Komponenten kommen. Und um dem hard- und softwaretechnischen Desaster vorzubeugen, müssen wir uns mit dem Begriff *HA* in all seinen Facetten auseinandersetzen.

HA steht – wie bereits im Vorwort erläutert – für den englischen Begriff *High Availability*. Auf Deutsch: Hochverfügbarkeit. Und dieser Terminus wird in freier Wildbahn üblicherweise immer in Verbindung mit etwas Hochprozentigem angetroffen – jedoch nicht dem Stoff, von dem wir im Ausfall-Fall gern mal ein Glas oder mehr hätten.

Nun gut: Was bedeuten dann eigentlich die ganzen 99,*irgendwas* Prozent (Hoch-) Verfügbarkeit, die uns irgendwelche Reseller oder Consulter andauernd versprechen? Pro Jahr gerechnet, entsprechen 99,9 % beispielsweise einer Downtime von grob gerechnet 9 Stunden, 99,9999 % hingegen nur noch einer Downtime von rund 30 Sekunden. Wir sehen schon, auch ein paar kleine Nachkommastellen können einen mächtig großen Unterschied machen – und das gilt in exponentieller Form leider auch in der Regel für das aufzuwendende Budget: jede kleine »9« mehr hinter dem Komma kostete deutlich mehr als die vorhergehende ... Und wenn wir wissen, dass schon ein normaler Reboot-Vorgang auf einem Linux-System – je nach gestarteten Diensten – ein paar Minuten in Anspruch nehmen kann, ahnen wir, wie klein ein Downtime-Zeitfenster von 30 Sekunden wirklich ist.

Was soll nun mit HA erreicht werden? Ausfallsicherheit, klar. Und die beginnt, was die Hardware angeht, schon bei den typischen Einzelkomponenten wie Festplatten, Netzwerkkarten oder Netzteilen. Jede Komponente, die hohen Belastungen unterworfen ist und keinesfalls den Geist aufgeben darf, muss in Produktiv-Serverumgebungen mit dem Anspruch der Hochverfügbarkeit zwangsweise redundant ausgelegt sein. Dabei liegen ein paar Lösungen zur redundanten Auslegung systemkritischer Komponenten natürlich nahe:

Mehrere lokale Platten lassen sich zur Erhöhung der Redundanz sehr einfach per Hard- oder Softraid zusammenfassen, zwei oder mehr Netzwerkkarten sorgen für die nötige Konnektivität, selbst beim Wegfall einer der Verbindungen, und weitere redundante Komponenten runden die Ausfallsicherheit unseres Servers ab. Daneben kann die Gesundheit des Mainboards und der CPU, zumindest was Temperatur und Spannung angeht, üblicherweise ebenfalls relativ einfach überwacht werden. Aber irgendwann stößt eben auch jede lokal redundant ausgelegte Hardware an ihre Grenzen: Spätestens dann, wenn sich Mainboard, CPU und/oder RAM aus dieser Welt verabschieden, oder unser Server dank höherer Gewalt oder unüberlegter menschlicher Interaktion geschreddert ist.

Je nach thermischer und mechanischer Beanspruchung gibt jede Komponente irgendwann den Geist auf, soviel ist sicher. Was uns dabei vor allem interessiert: Es soll möglichst spät passieren und am besten erst dann, wenn wir gerade nichts mit der Administration des Systems zu tun haben. Und fernab der Hardware kann noch ein typisches Problem dem – wenn auch hardwaretechnisch redundant ausgelegten – Standalone-Server den Garaus machen; und zwar auf Software-Ebene. Denn ist der Prozess, der zur Kernaufgabe unseres Servers gehört, so festgefressen, dass er ohne Reboot des Systems nicht wieder zu beleben ist, stehen wir ebenfalls dumm da.

Diese Beispiele stellen nur eine sehr schmale Palette der möglichen Ereignisse dar, die den Betrieb unserer Systeme stören können, und das bringt uns zu folgender Erkenntnis: Lokale Redundanz, bezogen auf einen Server, ist ein guter Ansatz, aber mit Sicherheit nicht die finale Lösung für hochverfügbare Produktivumgebungen. Ohne eine im Idealfall räumlich getrennte zweite (oder dritte bis *n*-te) Maschine, die im Fehlerfall der ersten für sie einspringen kann, haben wir ein Problem. Echte Hochverfügbarkeit beginnt bei redundanten Einzelkomponenten der Server und endet frühestens bei redundanten, physikalischen Servern, die zudem räumlich bzw. brandschutzzonentechnisch klar getrennt sind.

Geht es darum, bestimmte Services ausfallsicher zu machen, müssen wir zunächst analysieren, ob diese Services nicht schon selbst für die Replikation ihrer Daten sorgen. Typische Beispiele hierfür wären MySQL und OpenLDAP. File-Content-basierte Dienste wie Apache oder Samba 3 (letzter ausschließlich bezogen auf die Shares, *nicht* auf das Active Directory in Samba 4, das sich ebenfalls eigenständig um die Replikation seiner Objektdaten kümmert) wären hingegen in der Regel Fälle, die das nicht selbst erledigen, sondern andere, semi-intelligente Replikationslösungen benötigen, wie z. B. DRBD *(Distributed Replicated Blockdevices),* oder redundante und verteilte Dateisysteme wie z. B. Ceph oder GlusterFS.

Allerdings löst dies auch nur das Problem der reinen Datenverfügbarkeit – eine echte, applikationsbezogene Hochverfügbarkeit erreichen wir nur in Verbindung mit einer entsprechenden Cluster-Software, die sich um das Monitoring und

Management unserer Ressourcen kümmert und sie im Notfall auf einen anderen Node »transferiert«. »Transfer« in Anführungszeichen? Ja, weil im eigentlichen Sinne kein Transfer stattfindet, sondern die Ressourcen auf dem Failover-Node neu gestartet werden bzw. eine dort bereits aktive, zweite Instanz der Services die Requests stellvertretend entgegennimmt. Diese und andere Konzepte und Funktionsweisen werden wir ab Kapitel 7 genau betrachten, und anhand zahlreicher Beispiele aus der Praxis, die für viele, bereits erfolgreich in Betrieb befindliche Produktivlösungen entwickelt wurden, vertiefen. Fassen wir die zuvor gemachten Betrachtungen zusammen, müssen wir uns in logischer Konsequenz im Folgenden auf fünf Schwerpunkte konzentrieren, von denen die vier folgenden den größten Teil des Spektrums »Hochverfügbarkeit« ausmachen:

1. Lokale HA
2. HA-Cluster
3. HA-Storage-Cluster
4. STONITH und Debugging

Der fünfte Bereich tritt dann in Kraft, wenn die Kuh längst ins Eis gekracht, der Eimer in den Brunnen gefallen ist, das rote Telefon mit Direktleitung zum Chef Sturm klingelt und der firmentechnische Super-GAU trotz aller Redundanz dennoch eingetreten sein sollte. Und falls wir nicht gerade zufällig ein One-Way-Ticket nach Maui oder Pago-Pago in der gepackten Tasche haben sollten, müssen wir uns ganz fix um die Wiederherstellung unserer Systeme kümmern:

5. Disaster Recovery

Aber: es geht hierbei nicht nur um isolierte, rein technische Betrachtungen, sondern vor allem auch darum, wie wir uns der jeweiligen Thematik und ihren Problemstellungen optimal nähern können. Um zu verstehen, was Hochverfügbarkeit im Detail bedeutet, müssen wir uns zunächst mit dem Konzept als Ganzem und einigen seiner Fachtermini auseinandersetzen.

1.1.1 Parallelität, MTBF, MTTR und einiges mehr ...

Parallelität? Richtig gelesen. Allerdings kein banaler Exkurs in Grundschul-Geometrie, sondern die erste, unabdingbare und grundsätzliche Voraussetzung für ein redundantes System jedweder Art. Dazu ein ganz einfaches Beispiel: Jeder von uns kennt eine Kette, egal, ob es sich nun um eine Fahrrad-, Motorrad- oder Halskette handelt (okay – je nach musikalischer Vorliebe und/oder Geschlecht können die ersten beiden durchaus auch der letzten Gruppe zugeordnet werden).

Fakt ist jedoch: Jede von ihnen ist nach dem gleichen Prinzip aufgebaut; ein Kettenglied greift in das nächste. Irgendwann – bei zunehmender Last und/oder mechani-

schem bzw. alterungsbedingtem Verschleiß – reißt das schwächste von ihnen, und die gesamte Kette ist unbrauchbar; sie kann der ihr zugedachten Funktion nicht mehr nachkommen.

Beziehen wir die Kette nun auf ein konkretes System in der IT-Welt, setzt sich diese Kette der Informationsverarbeitung – hier stark simplifiziert – aus verschiedenen Komponenten zusammen: Der Stromversorgung, dann dem Server selbst mit all seinen Komponenten, dem Betriebssystem, der Applikation, der Informationsweiterleitung über die Netzwerkkarten und schließlich den Switches und Routern. Bei nur *einem* fehlerhaften Glied in dieser Kette ist die Informationsübermittlung zum Ziel unterbrochen.

Wir sehen also: Einfache Serialität und redundante Systeme sind Begrifflichkeiten, die sich gegenseitig ausschließen. Allein parallel arbeitende Systeme bringen uns die notwendige Redundanz. Und diejenigen, die schon einmal einen Motorradmotor zerlegt haben, erinnern sich vielleicht noch an die guten alten Duplex- oder Triplex-Steuerketten. Und damit an eine ganz einfache und klare Manifestation einer Form des Redundanz-Prinzips.

Prinzipiell sind die beiden Verfahren, seriell und parallel, auch mit logischen Verknüpfungen zu vergleichen: Das serielle Konzept funktioniert nur dann, wenn Komponente 1 *und* Komponente 2 *und* Komponente (n) funktionieren. Jede zusätzliche Komponente verringert die Wahrscheinlichkeit eines funktionierenden Konstrukts, da jede von ihnen als *SPoF (Single Point of Failure)* das komplette Gesamtkonstrukt lahmlegen kann. Völlig anders hingegen das einer Oder-Verknüpfung entsprechende, parallel arbeitende System: Jede zusätzliche Komponente erhöht die Verfügbarkeit, da das System erst nach dem Ausfall der letzten verfügbaren Komponente den Dienst einstellt. Allerdings müssen in diesem Parallelitäts-Beispiel für jede der eben genannten Komponenten aus dem Beispiel der IT-Kette ein oder mehrere redundante Gegenparts existieren: multiple USV (Unterbrechungsfreie Spannungsversorgung), redundante Komponenten im Server selbst, redundante Server, Switches usw. Genau genommen reden wir also von redundanten, seriellen Ketten, die den Ausfall von defekten Komponenten verkraften können, ohne dass die Kette »reißt«.

Statistische Spielereien

Und das bringt uns direkt zum nächsten Punkt: Dem Ausfall einer Komponente. Und da wir längst wissen, dass die Frage niemals lauten muss, *ob* eine Komponente ausfällt, sondern immer nur *wann*, stellt sich die nächste Frage ebenso logisch: *Wann also?* Und dafür gibt es wie so oft in unserer IT-Welt zumindest Näherungswerte. Diese Näherungswerte spezifizieren zum Beispiel, wie lange eine Komponente im Durchschnitt arbeitet, bis sie ausfällt. Je weniger komplex die Komponente, desto exakter kann dabei in der Regel die Zeitspanne bis zum Ausfall angegeben werden.

Dabei existieren zwei Begriffe, die die Zuverlässigkeit bzw. die Lebensdauer einer Komponente oder eines Systems spezifizieren, MTBF und MTTF.

MTBF (engl.: Mean Time Between Failures)

liefert nach allgemein gültiger Definition die durchschnittliche Zeit in Stunden zwischen zwei Ausfällen eines reparierbaren Systems oder einer Komponente. MTBF kann als Inversion der Fehlerrate für konstante Fehlerraten von Systemen berechnet werden, z. B.:

Hätten wir eine Komponente, die statistisch betrachtet eine Fehlerrate von zwei Ausfällen in einer Million Stunden hätte, wäre die MTBF die Inversion der Fehlerrate:

MTBF = (1.000.000 Stunden) / (2 Fehler) = 500.000 Stunden

MTTF (engl.: Mean Time To Failure)

MTTF stellt die Maßeinheit der Zuverlässigkeit für nicht-reparable Systeme dar, welche z. B. nach dem Romberg-Verfahren berechnet werden kann und, stark vereinfacht ausgedrückt, definiert, wie lange die vakante Komponente in unserem System im Durchschnitt hält, bevor sie sich und die Materialien, aus denen sie gefertigt wurde, wieder in den Recycling-Zyklus eingliedert. In der Praxis hat sich jedoch eher der eben erläuterte Begriff MTBF durchgesetzt, da er grob gesehen nichts anderes definiert als eben die Zeitspanne bis zum Ausfall einer beliebigen Komponente.

Kommen wir zurück zur Komplexität der Komponenten. Typischerweise geben viele Festplattenhersteller statt der MTBF dennoch die MTTF ihrer Platten an, da es sich um eine relativ isoliert arbeitende und in der Regel nicht reparable Komponente handelt. Hierbei spielen natürlich etliche Faktoren hinein, z. B. Anzahl der Zugriffe auf die Platte, Kühlung und Umgebungstemperatur, Spannungsschwankungen oder -spitzen, mechanische Stöße und/oder Vibrationen, denen sie ausgesetzt ist. Dennoch lässt sich die MTTF/MTBF für eine Komponente wie eine Harddisk relativ gut extrapolieren, insofern die eben beschriebenen externen Faktoren wie Wärme, Vibration, Zugriffe etc. innerhalb gewisser Normwerte spezifiziert sind, die dem Profil einer üblichen Nutzung entsprechen.

Bei SSDs, den Solid State Disks, die sich in den letzten Jahren mehr und mehr zu Standard-Komponenten, auch in Serversystemen entwickelt haben, sieht die Sache natürlich schon wieder etwas anders aus: Konventionelle Faktoren wie Erschütterung/Vibrationen lassen sie z. B. relativ kalt – natürlich nur, wenn sie sich nicht in der Liga *Hey, lass uns doch einfach mal mit dem Hammer draufhauen und sehen, ob das Ding immer noch funktioniert* abspielen.

Ähnliche Parameter gelten z. B. für Netzteile, USV, Netzwerkkarten (insofern es sich nicht um Onboard-Komponenten handelt). Und das bringt uns wiederum zum

nächsten Punkt. Denn die Sache sieht schon ganz anders aus, wenn wir ein typisches Mainboard betrachten. Jede Menge hochkomplexer Komponenten auf dem einen (Zahl: 1) Board, die miteinander interagieren, dazu separate Einzelkomponenten, die unter Umständen von anderen Herstellern stammen, jedoch integraler Bestandteil des Boards sind, wie RAM oder Multi-Core-CPU. Eine MTBF für eine solche Gesamt-Komponente bzw. ein Konstrukt der vorgenannten Komponenten anzugeben bzw. auszurechnen, ist logischerweise oft nur schwer möglich bis illusorisch, insofern der Hersteller des Gesamtpakets selbst keine durch Tests fundierten Näherungswerte vorgibt.

Eine weitere Abkürzung, die dann relevant wird, wenn eine Komponente bzw. das System ausgefallen ist, stellt die MTTR dar. MTTR beziffert die *Mean Time to Repair*, also die erforderliche Reparaturzeit. Die *Verfügbarkeit* einer Komponente oder eines Systems lässt sich anhand der bereits bekannten MTBF und der MTTR einfach mittels einer mathematischen Formel ableiten:

$$Verfügbarkeit\ (\%) = \left(\frac{MTBF}{MTBF + MTTR} \right) \cdot 100$$

Als Ergebnis dieser einfachen Berechnungsvorschrift erhalten wir die Verfügbarkeitsklasse des Systems in Prozent, die sich entsprechend der folgenden Tabelle 1.1 aufschlüsselt:

Verfügbarkeits-Klasse	Prozentuale Verfügbarkeit	Ausfallzeit pro Jahr
2	99 %	3,6 Tage
3	99,9 %	8,76 Stunden
4	99,99 %	52 Minuten
5	99,999 %	5 Minuten
6	99,9999 %	30 Sekunden
7	99,99999 %	3 Sekunden

Tabelle 1.1 Verfügbarkeits-Klassen nach IEEE

Die Verfügbarkeit eines Systems wird von der Harvard Research Group (HRG) in sechs verschiedene Klassen, entsprechend ihrer *Availability Environment Classification* (kurz: AEC-0–5), eingeteilt. Schauen wir uns die zugehörigen Definitionen in Tabelle 1.2 an:

Verfügbarkeits-Klasse (AEC)	Bezeichnung	Beschreibung
AEC-0	*Conventional*	Funktion kann unterbrochen werden, Datenintegrität ist nicht essentiell.
AEC-1	*Highly Reliable*	Funktion kann unterbrochen werden, Datenintegrität muss jedoch gewährleistet sein.
AEC-2	*High Availability*	Funktion darf nur innerhalb festgelegter Zeiten oder zur Hauptbetriebszeit minimal unterbrochen werden.
AEC-3	*Fault Resilient*	Funktion muss innerhalb festgelegter Zeiten oder während der Hauptbetriebszeit ununterbrochen aufrechterhalten werden.
AEC-4	*Fault Tolerant*	Funktion muss ununterbrochen aufrechterhalten werden, 24/7-Betrieb (24 Stunden, 7 Tage die Woche) muss gewährleistet sein.
AEC-5	*Disaster Tolerant*	Funktion muss unter allen Umständen verfügbar sein.

Tabelle 1.2 Verfügbarkeitsklassen nach AEC

Gehen wir von den Spezifikationen der Tabellen aus, würde bereits eine durchschnittliche Downtime von 3,6 Tagen (!) pro Jahr in die Klassifikation »AEC-2, High Availability« fallen, entsprechend einer Downtime von gut 1,5 Stunden pro Woche. Unvorstellbar, aber wahr.

Fakt ist natürlich, dass jede kleine 9 *hinter* dem Komma – wie bereits eben angerissen – in Euro eine ganze Menge mehr an Zahlen *vor* dem Komma kostet. Im Durchschnitt sind mit ca. 10.000 € bereits Systeme realisierbar, die nur mit rund 10 Stunden Downtime pro Jahr betrieben werden können, was bereits einer Verfügbarkeit von rund 99,9 % entspricht. Der Sprung auf die nächste Nachkommastelle wird in der Regel aufgrund der Sicherstellung der Redundanz von »externen« Komponenten – wie der redundanten Anbindung externer Außenstellen oder zusätzlicher Rechner, Klimaanlagen und USV – wesentlich teurer; hier sind Beträge im hohen, fünfstelligen Bereich und mehr keine Seltenheit. Ein Betrag, den wohl die wenigsten von uns im Firmenbudget, geschweige denn »mal eben« in der Brieftasche haben dürften.

Aber das ist auch nicht unbedingt die Liga, in der wir spielen müssen. Wir bleiben dabei, mit relativ moderaten Hardware-Ausstattungen und Open-Source-Software-Paketen unsere Systeme in der Praxis so hochverfügbar wie eben möglich aufzuset-

zen. Viele der im Folgenden vorgestellten Verfahren sind ebenso für den Einsatz in größeren Serverfarmen und Rechenzentren geeignet, und in etlichen auch bereits erfolgreich und zuverlässig in Betrieb.

1.2 Tool-Time zum Ersten: Das richtige Werkzeug

»Real men don't need instructions«
– Tool Time, USA 1991–1999

Nein, kein Review von Tim Taylors tiefschürfenden Weisheiten über Werkzeuge, Frauen, das Leben im Allgemeinen und den Rest des Universums. Aber nichtsdestotrotz sollten wir uns mit einem prinzipiellen Aspekt beschäftigen, der sich um genau *die* Werkzeuge dreht, mit denen wir unser System überwachen wollen.

Als Leser dieses Buches werden Sie vielleicht schon wissen, dass Linux für die meisten Aufgaben die passenden Werkzeuge mitbringt, und zwar von Haus aus. Und das bringt uns zum nächsten Aspekt: Dem Lebenszyklus der Werkzeuge. Denn die meisten Linux-Werkzeuge existieren – anders als ihre Pendants aus dem fröhlichen Fensterland – üblicherweise seit etlichen Linux-Generationen, sind beständig gepflegt worden und erfreuen sich in der Regel größter Stabilität und Ausgereiftheit.

Und das sollte uns zu einer weiteren wichtigen, konzeptionellen Überlegung führen. Nämlich der, welches Werkzeug wir für welchen Zweck einsetzen. Sicher existieren zur Überwachung des Systems relativ umfangreiche Tools aus der Liga »eierlegende Wollmilchsäue« – aber: macht es wirklich in jedem Fall Sinn, ein gegebenenfalls sehr komplexes Tool zur Überwachung eines recht simplen Teils bzw. Teilbereiches unseres Systems einzusetzen? Wohl kaum. Denn mit dem Grad der Komplexität steigt immer gleichzeitig die Wahrscheinlichkeit möglicher Fehlkonfigurationen und/oder -funktionen; das gilt sowohl für das Zielsystem als auch für das System, das es überwacht. *Who watches the Watchmen – wer überwacht die Wächter?*

Sicher gilt – zumindest ab einer gewissen Komplexität des zu überwachenden Systems –, dass die Überwachung nur noch mit relativ komplexen, zentralisierten Tools erschlagen werden kann. Dennoch: Der größte Feind der Hochverfügbarkeit ist die Komplexität, wie wir bereits aus unseren Vorbetrachtungen im letzten Abschnitt wissen. Und so sollten wir, bevor wir unser System mit hochkomplexen Funktionalitäten zur Überwachung ausstatten, immer zuerst die Frage der Verhältnismäßigkeit stellen. Und das bringt uns zurück zu den Werkzeugen, die Linux von Haus aus mitbringt. Verquicken wir nun den Grundgedanken: »Keep it simple« (*KIS*, oder auch *KISS* – »Keep it simple and stupid«) mit ein wenig Hirnschmalz und den richtigen Standard-Werkzeugen, erhalten wir in der Regel die effizienteste und vor allem stabilste Überwachung für den jeweiligen Anwendungsfall.

Anmerkung

Stark vereinfacht geht es vor allem darum, ein Gespür dafür zu erlangen, *wie* an ein Problem bzw. eine Aufgabenstellung herangegangen werden kann. Das *Womit* ergibt sich nach der richtigen Analyse der Aufgabenstellung ohnehin meist in logischer Konsequenz. Wir werden in den folgenden Kapiteln etliche Szenarien aus der Praxis mithilfe der vorgenannten Tools durcharbeiten – das Wichtigste ist und bleibt jedoch, das *Warum* der dahinterliegenden Thematik zu verinnerlichen. Auch wenn die Tools aus unserem Linux-Universum üblicherweise eine längere Halbwertszeit als ihre kommerziellen Gegenparts haben: Irgendwann werden auch sie abgelöst oder ersetzt, und dann bringt uns nur das richtige Verständnis weiter, nicht die sture Abarbeitung auswendig gelernter Prozeduren.

Und dabei geht es auch darum, ebenfalls ein Gespür dafür zu entwickeln, welche HA-relevanten Tools und/oder Komponenten für uns bzw. unsere Aufgabenstellung die richtigen sind; ob sie z. B. den gewünschten Funktionsumfang und einen entsprechenden, langfristigen Planungshorizont abdecken können, oder ob ihre Entwicklung früher oder später schon rein konzeptbedingt schlicht und einfach im Sande verlaufen muss. Nicht alles in der HA-Welt ist toll, und nicht jede HA-relevante Komponente und/oder jedes Tool muss der Weisheit letzter Schluss sein. Im Klartext: Fernab von markigen Zusicherungen und ellenlangen Featurities-Listen gilt für uns immer: »Nein danke« zur rosa Brille und dann selbst sorgfältigst zu analysieren.

Bevor wir in den folgenden Kapiteln ans Eingemachte gehen, vorab ein paar Worte zu den verwendeten Paketen, Prozeduren und Termini – denn entgegen der Philosophie des Heimwerker-Kings Tim Taylor lesen auch echte Kerle trotz allem hier und da mal eine Betriebsanleitung.

1.2.1 Päckchen

Bei den Paket- und Konfigurationsangaben beziehe ich mich größtenteils auf drei der in Unternehmen am häufigsten anzutreffenden Linux-Distributionen: Ubuntu 12.04.2 LTS »Precise Pangolin«, CentOS/RHEL 6.4 und SUSE Linux Enterprise Server 11 SP2, letzterer im Folgenden üblicherweise auch oft nur als »SLES« bezeichnet. Als Basis der durchgeführten Betrachtungen beziehe ich mich auf die jeweils zum Zeitpunkt der Erstellung des Buches aktuellen Paketversionen, die entweder über die Repositories der jeweiligen Distribution bezogen werden können oder alternativ als Tarball von der Seite des jeweiligen Maintainers. Aber: von eigenkompilierten Komponenten sollte im professionellen HA-Bereich stets Abstand genommen werden; weitere Betrachtungen der Thematik im Hinblick auf Distributionen, Pakete, Versionsstände und Support finden sich ab Kapitel 7.

Hinweis

Dabei sollte allerdings jedem Leser klar sein, dass ich kaum jede noch so kleine Besonderheit aller möglichen Versionen eines Tools und/oder einer bestimmten Distribution »erschlagen« kann; ebenso wenig alle Abweichungen zu anderen Distributionen, dies ist schlichtweg nicht realisierbar. Hier hilft wie üblich wirklich nur ein Studium der versionsbezogenen Manpages der jeweiligen distributionsspezifischen Komponenten.

An dieser Stelle weise ich auch explizit darauf hin, dass ich für alle folgenden Konfigurationen keines der distributionsspezifischen Konfigurationstools, wie z. B. SUSEs *YaST*, verwende, sondern stets ein »manuelles« Setup beschreibe.

Ebenfalls im Anhang findet sich – neben den Paketlisten – eine Übersicht der wichtigsten Manpages, jeweils nach den entsprechenden Abschnitten dieses Buches unterteilt (z. B. Softraid, LVM, Clustering, Virtualisierung usw.).

1.2.2 ... und Betriebsanleitungen

Apropos Manpages:

Wenn ich im Folgenden auf *Manpages* verweise, dann geschieht dies üblicherweise in den Formen:

man (<Manpage-Sektion>) <Programmname | Konfigurationsdatei>

oder

<Programmname | Konfigurationsdatei> (<Manpage-Sektion>)

Bezogen auf den *smartd*-Daemon zur Überwachung der Festplatten verweise ich also z. B. auf *smartd(8)*, was der ausgeschriebenen Form *man 8 smartd* oder `smartd(8)` entspricht. Infos zu den Manpages, selbst, ihren Sektionen und vielem anderen mehr, finden sich in *man man*. Mann, Mann ...

Um die Existenz eines bestimmten Pakets zu prüfen, hilft bei SUSE z. B. ein schnelles

```
#> rpm -qa | grep <Teil des Paketnamens>
```

oder

```
#> zypper search <Teil des Paketnamens>
```

bzw. bei Ubuntu:

```
#> aptitude search <Teil des Paketnamens> | grep ^i
```

oder RHEL / CentOS und Konsorten:

```
#> yum search <Teil des Paketnamens>
```

So kann einfach bestimmt werden, welche Pakete bereits installiert sind, und welche noch nachinstalliert werden müssen.

Hinweis

Das verwendete **#>** steht hier und in allen folgenden Beispielen für den Kommandoprompt der Shell.

Kapitel 15
Pacemaker 1.1.8, Corosync 2.x und pcs

»Natürlich gibt es die eierlegende Wollmilchsau. Und die sprechenden Gummibärchen im Fernsehen sind ebenfalls echt.«

Ja klar, eierlegende Wollmilchsäue gehören zur Grundausstattung eines jeden Admins: Ein richtig fetter Server, der Fileservices bereitstellt, als Verzeichnisdienst, Mailserver und Proxy arbeitet, das Wetter voraussagt, den Körperfettanteil berechnet, den Müll rausbringt und Pizza holt. Na klar. Im Ernst: Das Bestreben, Programme bzw. Applikationen/Services – oder nennen wir es einfach mal »Werkzeuge« – funktional aufzubohren, liegt in der Natur der Dinge, bzw. des Menschen. So auch das Bestreben von Beekhof & Co., mit *pcs* ein Allroundtool zu kreieren, das sowohl die Pacemaker- als auch die Corosync- (also sprich: unsere gesamte Cluster-)Konfiguration erschlagen kann.

Fast alle Erfahrungen aus der Praxis zeigen jedoch, dass nahezu alle *All-Around-ich-kann-alles*-Tools, vom Multitool aus dem Baumarkt bis zum universellen Quantengravitations-Flux-Kompensator für den Hausgebrauch, zwar nach vorne mit ihrer Ambition glänzen, die reale Umsetzung jedoch fast immer stark zu wünschen übrig lässt – weil einfach an bestimmten Ecken und Enden immer Kompromisse gemacht werden müssen. Sollte die Erde nicht zu einer Scheibe mutieren, wird der Admin auch weiterhin tiefgreifendes Know-How über die Konfigurationsdateien und Stellschrauben seines Clusters benötigen, und das aus gutem Grund, wie wir gleich sehen werden.

Aber schauen wir uns zunächst einmal konkret in der Praxis an, ob pcs, also das neue *Pacemaker Configuration System*, das halten kann, was die ambitionierten Ankündigungen versprechen. Zunächst zu den Voraussetzungen, die im Titel des Abschnitts schon benannt sind: Pacemaker ab Version 1.1.8 und Corosync 2.x.

15.1 Vorbetrachtungen

Als Ausgangsbasis für diese Demo verwende ich eine – zum Zeitpunkt der Erstellung dieses Buches – aktuelle Fedora 18. Wie – Fedora 18? Hatten wir nicht vor ein paar Abschnitten noch ausdrücklich darüber philosophiert, wie endlos ungeeignet Short-Term-Support-Versionen von Distributionen für Cluster-Umgebungen sind? Korrekt.

Und warum dann jetzt ausgerechnet eine Fedora 18, die erfahrungsgemäß mehr Patches und Upgrades am Tag herunterwürgt als ein ausgehungerter Kojote vitaminreiche Fast-Food-Reste aus dem Abfall eines Schnellimbisses? Ganz einfach: Um das komplette Feature-Set des pcs (inklusive der Corosync-Konfiguration) wenigstens durchspielen und damit vorstellen zu können, ist dies derzeit (und damit nochmals: zum Zeitpunkt der Erstellung dieses Buches) die einzige, »sauber« reproduzierbare Option, da Andrew Beekhof derzeit nun einmal bei Red Hat seine Brötchen verdient, und Fedora 18 (und folgende) das Testbed für die nächste RHEL-Major-Release darstellt. RHEL/CentOS 6.3/6.4 unterstützen pcs zwar schon, jedoch nur bezogen auf die reine Pacemaker(1.1.8)-Konfiguration. Um per pcs ebenfalls Corosync managen zu können, wird neben dem `systemctl`-Tool und den *pcsd*-Services auf jedem Node auch Corosync 2.x benötigt (letzteres wäre kompilationstechnisch unter RHEL/CentOS zwar umzusetzen, aber nur mit jeder Menge nicht wirklich gut auflösbarer Abhängigkeiten. Zudem stünde uns der pcsd dort nicht zur Verfügung). Da pcs das von Beekhof & Co präferierte Cluster-Tool der nächsten Generation ist, ist davon auszugehen, dass so gut wie alle neuen Major-Releases der Enterprise/LTS-Distros dieses Feature-Set – inklusive Corosync 2.x – verwenden werden.

Ans Werk, werfen wir nun einen Blick in die mögliche Zukunft unserer Clusterkonfiguration. Nach der Installation der erforderlichen Pakete (siehe Anhang – konkret verwenden wir in diesem Setup die Versionsstände Pacemaker 1.1.9 und Corosync 2.3, bezogen auf Fedora 18) und der Konfiguration der Netzdevices etc. starten wir die Basiskonfiguration unseres Clusters.

Achtung

Zuvor sicherstellen, dass SELinux und Firewalls/Paketfilter auf den Nodes deaktiviert sind! Zudem sollten Cluster der Corosync ≥ 2.x-Generation keinesfalls mit älteren (< 2.x) gemixt werden, da sie gegebenenfalls nicht miteinander kommunizieren können. Ein »Rolling Upgrade« von Corosync 1.x auf 2.x ist daher ebenfalls *nicht* möglich.

15.2 Corosync-Konfiguration per pcs

Bevor der Cluster komplett per pcs administriert werden kann, müssen wir den pcs-Daemon (pcsd) auf allen Nodes aktivieren und permanent in die Autostart-Konfiguration einbinden:

```
#> systemctl start pcsd.service
#> systemctl enable pcsd.service
```

Im nächsten Schritt muss sichergestellt sein, dass der User *hacluster* auf allen Nodes mit einem identischen Passwort versehen wurde, damit wir die »remote«-Features des **pcs** nutzen können. Danach authentifizieren wir uns an unseren Cluster-Nodes:

```
#> pcs cluster auth jake elwood
Username: hacluster
Password:
jake: Authorized
elwood: Authorized
```

Dann können wir unseren Cluster, d. h. in diesem Fall zunächst Corosync, provisionieren:

```
#> pcs cluster setup pcmkcluster jake elwood
jake: Succeeded
elwood: Succeeded
```

Gut, aber was ist bisher passiert? Das **pcs**-Tool hat über die Befehls-Subsektion **cluster** eine Authentifizierung für den User *hacluster* auf beiden Nodes durchgeführt, sodass alle weiteren **pcs**-Befehle synchron auf den angegebenen Nodes abgesetzt und durchgeführt werden können. Im nächsten Schritt haben wir eine Corosync-Konfiguration auf beiden Nodes eingerichtet; betrachten wir nun, wie gut oder schlecht das Setup von **pcs** durchgeführt wurde, am Inhalt der generierten *corosync.conf*:

```
totem {
  version: 2
  secauth: off
  cluster_name: pcmkcluster
  transport: udpu
}
nodelist {
  node {
        ring0_addr: jake
        nodeid: 1
       }
  node {
        ring0_addr: elwood
        nodeid: 2
       }
}
quorum {
  provider: corosync_votequorum
}
```

```
logging {
to_syslog: yes
}
```

Zunächst fällt ins Auge, dass die Konfiguration Out-of-the-Box unverschlüsselt arbeitet; ein Umstand, der für Produktivumgebungen eher ungeeignet ist, und den wir manuell per **corosync-keygen** (Copy des Key auf den/die andere(n) Nodes nicht vergessen) und den `secauth: on`-Eintrag in der `totem`-Sektion beheben können.

Über das `udpu`-Setting in Verbindung mit den `nodelist`-Settings kann Corosync die Member des Clusters problemlos finden, aber: wenn unser Cluster – wie auf den hier verwendeten Testmaschinen – über zwei Netzwerkverbindungen verfügt, ist die automatische Konfiguration alles andere als optimal. Ein schneller Rapport vom **corosync-cfgtool** zeigt uns nämlich nur genau ein Ring, und zwar das primäre *eth0*-Device für Ring 0:

```
#> corosync-cfgtool -s
Printing ring status.
Local node ID 1
RING ID 0
        id      = 192.168.99.22
        status  = ring 0 active with no faults
```

Um beide Ringe zu aktivieren, müssen wir die *corosync.conf* explizit in der bereits vorgestellten Weise in der `totem`-Sektion anpassen, z. B.:

```
totem {
  version: 2
  threads: 2
  secauth: on
  cluster_name: pcmkcluster
  rrp_mode: passive
  ttl: 255
  interface {
          ringnumber: 0
          bindnetaddr: 192.168.99.0
          mcastport: 5405
  }
  interface {
          ringnumber: 1
          bindnetaddr: 10.0.0.0
          mcastport: 5407
  }
  transport: udpu
}
```

Anschließend sollten sich beide Ringe sauber zeigen. Eine weitere Neuerung ist das optionale **votequorum(5)**-Plugin. In Corosync 2.0 kümmert es sich in bekannter Quorum-Verfahrensweise darum, über die Anzahl der Votes (Default: ein Vote pro Node) und einen Quorums-Entscheid-Split-Brain-Situation zu vermeiden. Die Anzahl der `expected_votes` wird entweder automatisch über die Nodelist berechnet, kann aber auch über das explizite Setzen der `quorum`-Subdirektive `expected_votes` vorgenommen werden.

In einem reinen 2-Node-Cluster liegen die Dinge wie üblich komplizierter, aber auch dafür hat das `votequorum`-Plugin einen Lösungsansatz parat: Über die (aus *CMAN*-Konfigurationen gegebenenfalls bekannte) Zusatz-Subdirektive `two_node: 1` (boolean) können wir dem Cluster mitteilen, dass es sich bei unserem Cluster nur um einen 2-Node-Cluster handelt, der keine Mehrheitsentscheidung treffen kann. Zusatzdirektiven wären in diesem Fall z. B. `votes` und `wait_for_all`, mehr Infos hierzu liefert *votequorum(5)*, s.o.

15.3 Fire it up ...

Nachdem wir nun den Corosync-Part (mit manuellen Nachbesserungen) auf einen praktikablen Stand gebracht haben, können wir den Cluster mit der folgenden Befehlssequenz (auf beiden Nodes auszuführen) starten:

```
#> pcs cluster start
```

Sollte alles okay sein, liefert uns die Rückgabe des Befehls ein:

```
Starting Cluster...
```

Wer noch ökonomischer arbeiten will, startet beide Nodes gleichzeitig:

```
#> pcs cluster start --all
jake: Starting Cluster...
elwood: Starting Cluster...
```

Über den **`pcs cluster start`** -Befehl werden sowohl Corosync als auch Pacemaker abgefeuert. Der Befehl **`pcs status`** liefert uns einen **`crm_mon`**-like Single-Shot des Cluster-Status.

Insgesamt stehen uns unter der `cluster`-Sektion von pcs u. a. die folgenden Subdirektiven zur Verfügung, hier in einer kurzen auszugsweisen Auflistung mit gegebenenfalls erforderlichen Erklärungen: `start/stop/force_stop`, `enable/disable` (Corosync und Pacemaker auf dem betreffenden Node de-/aktivieren), `standby/unstandby`, `pcsd-status` (Status des pcs-Daemons auf den Nodes), `auth`, `sync` (*corosync.conf* auf alle anderen Nodes vom aktuellen aus syncen), `setup`, `cib` (raw-XML auslesen), `(local)node add/remove` (Node(s) in der Pacemaker- und Corosync-Konfiguration ent-

fernen bzw. hinzufügen). Die komplette Liste der verfügbaren Direktiven gibt **pcs cluster** ↵ aus. Über den Zusatzschalter `-f <Filename>` führt pcs die gewünschte Aktion bezogen auf die angegebene Datei durch, nicht mit der Live-CIB, z. B.

```
#> pcs -f dummy123 resource create Dummy \
   ocf:heartbeat:Dummy op monitor interval=10
```

Diese so erzeugte XML(!)-Datei (dummy123) kann dann anschließend per **push**:

```
#> pcs cluster cib dummy123
#> pcs cluster push cib dummy123
```

in die CIB importiert werden.

Achtung

Eine Auto-Completion für **pcs** gibt es derzeit nicht, wäre für zukünftige Releases aber mehr als wünschenswert.

Werfen wir nun noch einen kurzen Blick auf das Membership und die Quorum-API:

```
#> corosync-cmapctl | grep members
runtime.totem.pg.mrp.srp.members.1.config_version (u64) = 0
runtime.totem.pg.mrp.srp.members.1.ip (str) = r(0) ip(192.168.99.22) r(1) ip(10.0.0.22)
runtime.totem.pg.mrp.srp.members.1.join_count (u32) = 1
runtime.totem.pg.mrp.srp.members.1.status (str) = joined
runtime.totem.pg.mrp.srp.members.2.config_version (u64) = 0
runtime.totem.pg.mrp.srp.members.2.ip (str) = r(0) ip(192.168.99.23) r(1) ip(10.0.0.23)
runtime.totem.pg.mrp.srp.members.2.join_count (u32) = 1
runtime.totem.pg.mrp.srp.members.2.status (str) = joined
```

```
#> pcs status corosync
Membership information
----------------------
    Nodeid      Votes Name
         1          1 jake (local)
         2          1 elwood
```

Und nun zur essentiellen Arbeit an unserem Cluster: Der Verwaltung der Properties und Ressourcen ...

15.4 Pacemaker-Konfiguration per pcs

Feuern wir nun ein **crm_verify -LV** ab, so teilt uns der Cluster – wie gewohnt – mit, dass er bitteschön ein paar eindeutige Aussagen zum Thema STONITH wünscht, andernfalls verweigert er den Startup von Ressourcen – ebenfalls wie üblich – komplett. Um nun für unseren pcs-Testcluster die vakante Cluster-Property setzen zu

können, die den Startup von Ressourcen auch ohne STONITH ermöglicht, gehen wir wie folgt vor:

```
#> pcs property set stonith-enabled=false
```

Ein weiteres `crm_verify -LV` sollte nun keine Rückmeldung mehr geben.

Da wir zudem einen 2-Node-Cluster betreiben, der niemals ein Quorum bzw. eine Mehrheitsentscheidung treffen kann, geben wir ebenfalls die entsprechende Policy via pcs mit auf den Weg:

```
#> pcs property set no-quorum-policy=ignore
```

Über `pcs property show` können wir uns die gesetzten Properties anzeigen lassen:

```
#> pcs property show
Cluster Properties:
 dc-version: 1.1.9-0.1.70ad9fa.git.fc18-70ad9fa
 cluster-infrastructure: corosync
 stonith-enabled: false
 last-lrm-refresh: 1366658111
 no-quorum-policy: ignore
```

Um nun eine einfache Service_IP Ressource zu erzeugen, können wir wie folgt vorgehen (Zeile umbrochen):

```
#> pcs resource create Service_IP \
   ocf:heartbeat:IPaddr2 ip=192.168.99.225 \
cidr_netmask=24 op monitor interval=10
```

Löschen können wir die – laufende (!) – Ressource per

```
#> pcs resource delete Service_IP
```

Existiert nur ein einzelner RA mit diesem Namen in allen OCF-Subsektionen (pacemaker/linbit/heartbeat usw.), so reicht es bei der Erzeugung auch aus, einfach nur den RA-Namen anzugeben:

```
#> pcs resource create Service_IP IPaddr2 \
   ip=192.168.99.225 cidr_netmask=24 \
   op monitor interval=10
```

Über `pcs status` bzw. `pcs resource` können wir kontrollieren, ob die Ressource sauber gestartet wurde. Weitere Optionen zum Management der Ressourcen unterhalb der `resource`-Sektion wären z. B. `start/stop` (restart derzeit nicht implementiert), `(un)manage`, `show`, `list`, `describe`, `group {add,remove,list}`, `(un)clone` und `rsc|op defaults`. Eine komplette Liste liefert `pcs resource -h`. Eine Liste der verfügbaren Ressourcen-Standards (LSB, OCF usw.) liefert:

```
#> pcs resource standards
ocf
lsb
service
```

```
systemd
stonith
```

Eine Liste der verfügbaren RA-Subsektionen liefert:

```
#> pcs resource providers
heartbeat
linbit
pacemaker
redhat
```

Eine Agenten-Liste liefert der Befehl:

```
#> pcs resource agents <standard>:<provider>
```

also z. B.:

```
#> pcs resource agents ocf:heartbeat
```

Die Änderung der Parameter einer Ressource können wir per `update` in der *resource*-Subsektion anstoßen:

```
#> pcs resource update Service_IP ip=192.168.99.226
```

Erzeugen wir nun eine zweite Ressource (Apache) und co-lokieren diese mit der Service_IP (Zeilen umbrochen):

```
#> pcs resource create dummy ocf:heartbeat:Dummy \
op monitor interval=20
#> pcs constraint colocation \
add dummy Service_IP INFINITY
```

Anschließend fügen wir noch ein Ordering hinzu:

```
#> pcs constraint order Service_IP then dummy
Adding Service_IP dummy (kind: Mandatory) (Options: first-action=start then-action=start)
```

Über **`pcs constraint all`** können wir uns z. B. die Verbindlichkeiten der Ressourcen untereinander anzeigen lassen:

```
#> pcs constraint all
Location Constraints:
Ordering Constraints:
  start Service_IP then start dummy (Mandatory) (id:order-Service_IP-dummy-mandatory)
Colocation Constraints:
  dummy with Service_IP (INFINITY) (id:colocation-dummy-Service_IP-INFINITY)
```

Wie wir sehen, werden die Constraints nun immer mit ihrem vollen Funktionsbezeichner (colocation/order) als Prefix erzeugt. Wollen wir unsere Dummy-Ressource noch klonen, dann können wir dies z. B. wie folgt erledigen:

```
#> pcs resource clone dummy clone-max=2 clone-node-max=1
```

Der Name des Clones würde in diesem Fall aus `<Ressource>-clone` gebildet. Die Löschung des aktiven Clones erfolgt über:

```
#> pcs resource unclone dummy
```

In analoger Weise zum Cloning lassen sich natürlich auch Multistate-Ressourcen erzeugen. Gruppen würden sich z. B. durch folgende **pcs**-Syntax generieren lassen:

```
#> pcs resource group add dummygroup Service_IP dummy
```

Per `add`/`remove` können weitere Ressourcen der Gruppe hinzuaddiert bzw. entfernt werden.

Über **pcs config** können wir uns eine vollständige Konfigurations-Übersicht auf den Schirm holen. Soweit, so gut – oder auch nicht. Nun, unter dem Strich kehrt der Wunsch des Cluster-Admins nach der **crmsh** (benötigt *pssh* und *python-dateutil*) ganz schnell zurück, und kann – zum Zeitpunkt der Erstellung dieses Buches – für fast alle Distributionen auch hier erfüllt werden: *http://download.opensuse.org/repositories/network:/ha-clustering/*, sowie *https://savannah.nongnu.org/projects/crmsh/*. Allerdings bringen – in der getesteten Umgebung – schon ein simples **crm resource cleanup** oder auch andere, einfache Ressourcen-Änderungen per crm-Shell einen laufenden **crm_mon** zum Absturz. Insofern zeichnen sich hier bereits die ersten, hoffentlich bald wieder behobenen, Inkompatibilitäten ab.

Die Clusterlabs-Entwickler wären – mit allem gebotenen Respekt vor einer sehr gut entwickelten, ausgereiften und bewährten Cluster-Software – gut beraten gewesen, nicht wieder zu versuchen, das Rad neu zu erfinden, sondern sich auf eine sorgfältige Modellpflege (und gegebenenfalls Erweiterung) bestehender und bereits bestens funktionierender Tools – wie unserer crm-Shell – zu fokussieren. Nun denn, ein kompletter, pcs-spezifischer Guide findet sich unter *http://clusterlabs.org/doc/en-US/Pacemaker/1.1-pcs/*.

Kapitel 16
... und noch einmal Tool-Time: Eigene OCF-Ressource-Agenten erstellen

Tim: »Tag Wilson. Was machen Sie da?«
Wilson: »Ich schnitze mir ein Kanu.«
Tim: »Schwere Arbeit, was?«
Wilson: »Nein, eigentlich nicht, Tim. Man nimmt nur einen dicken Stamm und schnitzt alles weg, was nicht zu einem Kanu gehört.«
– Tool Time

16.1 Vorbetrachtungen

Zunächst: Die Entwicklung eigener OCF-Agenten ist kein Muss – sondern immer nur eine erforderliche Notwendigkeit. Wie bereits mehrfach erwähnt, ist bei der Wahl zwischen OCF und LSB immer dem OCF-RA – aufgrund seiner in der Regel erweiterten Monitoring-Kapabilitäten – der Vorzug zu geben. Steht kein entsprechender OCF-RA zur Verfügung, und das zur Verfügung stehende LSB-Pendant kann seiner Aufgabe konzeptbedingt nur unzureichend nachkommen, müssen wir den Werkzeugkoffer aufklappen und loslegen. Das Gleiche gilt für den Fall, dass ein bestehender OCF-Agent nur als Wrapper für ein LSB-Script fungiert und dadurch ebenfalls keine »höheren« und/oder selektiven Funktionen abbilden kann. Ein Beispiel hierfür wäre u. a. der Ceph-RA, dem wir im Abschnitt über Storage Clustering noch mehrmals begegnen werden.

Kernpunkte bei der (Weiter-)Entwicklung eines OCF-RA müssen vor allem immer die beiden folgenden sein:

1. Einhaltung der Standards, d. h. die Rückgabecodes des RA müssen entsprechend der Vorgabe/Standards normiert sein, damit Pacemaker die vom RA übermittelten Returncodes korrekt interpretieren und darauf reagieren kann.
2. Wenn ein RA eine dringend benötigte Überwachungs-Funktionalität nicht bietet – rüste sie nach. Der neue RA sollte jedoch in jedem Fall eine Kopie sein (z. B. *apache_custom*), damit der RA bei Paket-Upgrades nicht in der binären Mülltonne verschwindet.

16.1.1 Zum ersten Punkt – den zu verwendenden Standards und Returncodes

Existiert noch kein RA für die Aufgabe, kann die Ausgangsbasis für einen neuen RA z. B. entweder ein gegebenenfalls vorhandenes LSB-Script für den entsprechenden Service sein, oder der OCF-Dummy-RA, oder ein funktional ähnlich aufgebauter OCF-RA, wie z. B. im Fall von OpenLDAP und Samba 4: beide überwachen einen Verzeichnisdienst. Jeder RA sollte in einem entsprechenden Unterordner unterhalb des Pfades */usr/lib/ocf/resource.d/* liegen. Eigene Scripts bzw. RAs können wir sowohl in einem der vorhandenen Ordner (wie z. B. *heartbeat*) ablegen, oder einen neuen Ordner erstellen. Der Scriptname des Agenten gibt auch den Namen vor, über den er als *Primitive* konfiguriert werden kann. Jeder Agent muss die üblichen Operationen wie `start/stop/monitor` usw. interpretieren können. Je nach weiteren Kapabilitäten des RA (soll er z. B. auch als Clone- oder Multistate- Ressource arbeiten können) muss er gegebenenfalls noch andere Operationen, wie beispielweise `interleave`, `notify`, `promote` und `demote` verstehen und verarbeiten können. Wichtig ist in jedem Fall die Einhaltung der Standards, was die `OCF_`-Variablen und die von ihnen gelieferten Returncodes angeht. Hier eine Tabelle der vorhandenen Returncodes:

OCF-Return Code	OCF-Alias	Beschreibung	Recovery-Art
0	OCF_SUCCESS	Erfolgreiche Kommando-ausführung – der Cluster erwartet dieses Ergebnis z. B. für alle start/stop/monitor/ promote/demote-Operationen.	soft
1	OCF_ERR_GENERIC	Generische »Da ist ein unspezifiziertes Problem«-Fehlermeldung.	soft
2	OCF_ERR_ARGS	Ungültige Ressourcen-Konfiguration (z. B. ein benötigtes Tool wird nicht gefunden).	hard
3	OCF_ERR_UNIMPLEMENTED	Die gewünschte Aktion kann nicht implementiert werden (wird z. B. von der Binary nicht unterstützt).	hard
4	OCF_ERR_PERM	Ungenügende Privilegien zum Ausführen des Kommandos.	hard

Tabelle 16.1 OCF-Returncodes

OCF-Return Code	OCF-Alias	Beschreibung	Recovery-Art
5	OCF_ERR_ INSTALLED	Die von der Ressource benötigten Tools sind auf dem Node nicht installiert oder im entsprechenden Suchpfad.	hard
6	OCF_ERR_ CONFIGURED	Ungültige Ressourcen-Konfiguration (z. B. fehlende Parameter im RA).	fatal
7	OCF_NOT_ RUNNING	Die Ressource ist nicht aktiv, der Cluster versucht nicht, eine Ressource die diesen Returncode liefert, zu stoppen. Je nach Art des RA kann der Returncode ein Recovery erforderlich machen. Ist es ein erwarteter Status, passiert nichts, ist er unerwartet: Soft-Recovery (s. u.).	N/A
8	OCF_RUNNING_ MASTER	Die (Multistate-)Ressource läuft im Master-Mode.	soft
9	OCF_FAILED_ MASTER	Die Ressource ist im Master Node, gibt aber einen Fehler zurück. Die Ressource wird demoted, gestoppt, und dann ggf. neu gestartet und promoted.	soft
other	N/A	Custom error code.	soft

Tabelle 16.1 OCF-Returncodes (Forts.)

Die Returncodes finden sich auch in der Datei *ocf-returncodes*, meist unter */usr/lib/ocf/lib/heartbeat/ocf-returncodes*. Ergänzend müssen wir nun natürlich noch die Recovery-Arten betrachten, die der Cluster entsprechend des zurückgelieferten Returncodes startet, als da wären:

Recovery-Art	Beschreibung	Aktion des Clusters
soft	Ein vorübergehender Fehler ist aufgetreten.	Versuche die Ressource neu zu starten oder verschiebe sie an eine andere Lokation.
hard	Ein permanenter Fehler ist aufgetreten, der sich auf einen spezifischen Node beziehen kann.	Verschiebe die Ressource vom aktuellen Node und stelle mit einem entsprechenden Score sicher, dass sie auf dem vakanten Node nicht mehr gestartet wird.
fatal	Ein permanenter Fehler ist aufgetreten, der sich auf alle Nodes beziehen kann. In der Praxis bedeutet dies oft, dass eine fehlerhafte Konfiguration angegeben wurde.	Ressource stoppen und dafür sorgen, dass sie auf keinem der Nodes gestartet wird.

Tabelle 16.2 OCF-Recovery-Arten

Die Funktionalität des erstellten RAs sollte vor seiner Integration in den Cluster selbstverständlich ordentlich durchleuchtet werden. Hier steht uns der `ocf-tester(8)` hilfreich zur Seite, er testet den RA »trocken« durch und validiert, ob der der RA auf bestimmte Events (anhand der Returncodes) wie von Pacemaker vorgesehen, reagiert. Ein Beispiel für seine Arbeitsweise findet sich in Kapitel 25, »Debugging im Cluster«.

16.1.2 Zum zweiten Punkt – den Überwachungsfunktionalitäten

Zunächst – die üblichen Verdächtigen bei der Erstellung eines neuen OCF-RA sind natürlich Parameter wie z. B. die Pfade zur Binary, zum Config-File oder dem Config-Ordner, zum PID-File, usw. Um nun sicherzustellen, dass unser RA die ihm anvertraute Ressource auch sorgfältig überwacht, müssen wir uns vorstellen, wie wir selbst die Funktion dieser Ressource ohne Cluster-Integration validieren würden: Nehmen wir als Beispiel den Samba-4-RA, der den Beispieldaten zu diesem Buch beiliegt. Der RA entstammt ursprünglich dem slapd-OCF, also dem Ressource Agenten für OpenLDAP. Da Samba 4 jedoch (unter anderem) auch Verzeichnisdienst-Funktionalitäten in grob ähnlicher Form bietet, sind die Werkzeuge und Verfahren zur Überprüfung ähnlich gelagert: per `ldbsearch` wird in diesem Fall validiert, ob ein LDAP-»Tree« vorhanden ist; auch alle vorhandenen Kontexte können damit auf ihr Vorhandensein überprüft werden. Die ganze Chose könnte sich auch noch dahingehend ergänzen

lassen, ob der entsprechende Port 389 (z. B. via `nmap` oder `netstat` abgefragt) geöffnet ist. Wir sehen also – die Monitoring Funktionalitäten können wir in fast beliebiger Weise realisieren. Worauf jedoch strengstens zu achten ist, wäre u. a.: das nur Standard-Tools bzw. Tools, die nicht erst eine separate Kompilation erfordern, und die in der Regel auf jedem Linux-System vorhanden sind, verwendet werden. Bevor jetzt jemand auf die Idee kommt: »Ja nee, Liebel, is' klar … und was ist mit dem *ldbsearch* vom S4 aus Abschnitt 20.4? *Der* ist doch kompiliert.« Antwort: »Ja nee, is' tatsächlich klar – weil der S4 sehr bald zur Standardausstattung aller Linux-Distros gehören wird und sich die Tools damit auch in den Standard-Suchpfaden befinden.«

So viel an dieser Stelle zum Thema »eigene RAs«. Weitere Hinweise zur Erstellung eigener OCF-RAs (und damit auch eingebundener Sub-Funktionen) finden sich z. B. unter *http://www.linux-ha.org/doc/dev-guides/ra-dev-guide.html*, sowie *http://doc.opensuse.org/products/draft/SLE-HA/SLE-ha-guide_sd_draft/cha.ha.agents.html*, sowie *http://clusterlabs.org/doc/en-US/Pacemaker/1.1-pcs/html-single/Pacemaker_Explained/index.html#s-ocf-return-codes*. Und nun wird es Zeit für ein kleines Zwischen-Fazit.

16.1.3 Fazit

Womit wir unsere Cluster zukünftig auch managen werden: Die dann verwendeten Tools zum Management unserer Cluster-Ressourcen müssen einen vollen Werkzeugkoffer mitbringen, der zudem vom verwendeten Cluster-Brain voll und ganz unterstützt werden muss.

Neben intelligenten RAs (die die echte Verfügbarkeit eines Dienstes in möglichst allen möglichen und unmöglichen Situationen erfassen und sicherstellen sollen) für alle benötigten Ressourcen und für eine zuverlässige, effiziente Überwachung der Systemgesundheit sind Werkzeuge zur möglichst realen Simulation (siehe Shadow-Copy) von geplanten Szenarien absolut zwingend notwendig. Die Cluster-Management-Werkzeuge selbst müssen intuitiv sein – die crm-Shell und ihre sehr intuitive Bedienung zeigen nach wie vor den richtigen und besten Weg auf. Eine weitere Optimierung der Auto-Vervollständigung als Hilfestellung bei der Erzeugung, dem Management und der Überwachung von Ressourcen, ist abseits der Klicki-Bunti-Administration die richtige und direkteste Lösung für den Praxiseinsatz.

Zum Thema *pcs*: Auch wenn einige Ansätze in die richtige Richtung gehen, das pcs-Tool ist noch ein gutes Stück von dem Begriff »Praxistauglichkeit« entfernt, wie unsere eben erfolgte Betrachtung deutlich gezeigt haben sollte. Weder ist die Handhabung intuitiv, noch lassen sich in der getesteten Version einfache, aber in der täglichen Praxis durchweg erforderliche Operationen wie ein Ressourcen-Restart oder -Cleanup durchführen. Die selektive Modifikation von Ressourcen bzw. einzelnen Attributen/Parametern im laufenden Betrieb ist derzeit ebenfalls weder besonders

transparent noch simpel, und eine halbwegs praxistaugliche Corosync-Konfiguration lässt sich Out-of-the-Box nur mit Einschränkungen generieren. Die fehlende Auto-Completion rundet das unzureichende Bild ab. Klingt hart? Das soll es. Nicht jede Neuerung muss zwangsläufig eine Verbesserung sein. Das pcs-Tool hat – zumindest nach derzeitigem Stand, und mal sehr vorsichtig ausgedrückt – noch ein gutes Stück Weg vor sich.

Was GUI-Implementierungen angeht: Jede GUI sollte auch immer unter dem Aspekt ihrer Nähe zur eigentlichen Cluster-Applikation bewertet werden, wie z. B. die CRM in Verbindung mit der Pacemaker-GUI bzw. pcs mit der kommenden pcs-GUI. Ein Drittanbieter kann wesentlich schneller die Lust daran verlieren, eine GUI weiterzuentwickeln, insbesondere wenn sich z. B. durch Übernahmen, Fusionen, Entwicklerabwanderungen etc. wirtschaftliche Notwendigkeiten und/oder andere Interessen/Einnahmequellen ergeben. Zudem kommen weitere Faktoren hinzu, die sowohl die Management-Tools selbst als auch die Cluster-Software und ihre RAs betreffen, wie z. B. die Möglichkeiten eines möglichst reibungslosen Upgrades, optimalerweise mit Rollback-Funktionalitäten. Nun aber zu unserem nächsten, großen Thema: Storage im Cluster.

Kapitel 17

Ausfallsichere Shared-Nothing-Cluster mit DRBD

»Replicants are like any other machine, they're either a benefit or a hazard ... If they're a benefit, it's not my problem.«
– Blade Runner, USA 1982

Yessir! Es geht – unter anderem – um Replikation. Und was schon der gute alte Rick Deckard pragmatisch schlussfolgerte, gilt auch für uns: Wenn unsere DRBD-, GlusterFS- und Ceph-Storage-Cluster als sogenannter *Shared-Nothing*-Verbund die Hochverfügbarkeit unserer Nutzdaten auf dem replizierten/verteilten Storage Cluster anstandslos zur Verfügung stellen, haben wir kein Problem. Aber – *Shared Nothing?* Klingt ja erst einmal nach herzlich wenig bis gar nichts, und »gar nichts« kann für uns im HA-Bereich (und auch darüber hinaus) wohl kaum hilfreich sein, oder? Falsch gedacht, zumindest in diesem Fall. Denn: Irgendein schlaues Köpfchen hat mal – wie wir schon wissen – ein einzelnes SAN (*Storage Area Network*, ein sogenanntes »Shared-All«-System) als »*Very Expensive Single Point of Failure*« bezeichnet. Dem ist eigentlich auch nichts hinzuzufügen, denn jeder, der sich mit einem einzigen »Standalone«-SAN – auch wenn es intern mit redundanten Raid-Leveln ausgestattet und mit zwei HBAs (*Host Bus Adapters*) via Multipathing an den Host angebunden ist – in Sicherheit wiegt, unterliegt einem mächtig großen Irrtum.

Denn auch auf dieser Ebene gilt – echte Ausfallsicherheit erreichen wir nur durch mindestens zwei physikalisch voneinander unabhängige Storage-Einheiten. Aber wie bereits eingangs erläutert: Wir werden hier keine speziellen, teuren Hardwarelösungen à la EMC betrachten, sondern eine optimale Storage-Verfügbarkeit mit Open-Source-Tools und völlig normalen Plattensubsystemen erreichen.

Bevor wir loslegen, analysieren wir zunächst, was wir bereits haben und wohin wir möchten. In unserem Einkaufswagen befinden sich bereits lokal redundante Raid-Systeme, die wir hervorragend mit LVM um größtmögliche Flexibilität erweitern können (oder per LV-Raid beides über LVM erschlagen können), das Ganze in aktuellen und kommenden Kernel-Generationen alternativ auch per BTRFS, um die serielle Kette der lokalen HA bei Bedarf drastisch einzudampfen.

Betrachten wir zunächst die unterschiedlichen Ansätze des Storage-Clusterings. Auf der einen Seite haben wir, zum Zeitpunkt der Erstellung dieses Buches, die »normalen« DRBD-8.x Shared Nothing Cluster, die üblicherweise mit zwei Nodes in einer Art Netzwerk-Raid-1-Verbund ihre Daten synchron halten. Auf der anderen Seite ist mit Ceph in den letzten Jahren ein völlig neues Konzept im Bereich der (Distributed-)Storage Cluster aufgetaucht, das weder eine strikte 2-Node-Grenze besitzt, sondern – ganz im Gegenteil – völlig flexibel skalierbar in der Anzahl der Nodes und der Bereitstellung des Storages ist, und dabei, wie auch DRBD, keine allzu besonderen Anforderungen an den Storage auf den lokalen Nodes stellt. Die DRBD-Truppe hat 2012 auch die Notwendigkeit erkannt, hier in – sehr grob verwandter Form, zumindest was die Anzahl der Nodes angeht – nachzuarbeiten, und mit DRBD 9.0 das konzeptionell veraltete 2-Node-Limit hinter sich zu lassen (Anmerkung: Die alten 3-Node-DRBD-»Stacked Ressources« konnte man auch bei sehr wohlwollender Betrachtung nicht wirklich als Raid 1 plus Spare im Netz betrachten).

Mit Ceph, GlusterFS und DRBD schauen wir uns – abseits von kommerziellen/proprietären SAN-Lösungen, die in diesem Buch, wie bereits ausdrücklich erklärt, *nicht* behandelt werden – die drei wichtigsten Stellvertreter der Gattung an.

Wir werden ihre konzeptionellen Vor- und Nachteile erörtern und sie bis in die letzte Schraube beleuchten – so wie die Integration der Storage-Ressourcen in unserem Cluster. Das Ganze hinterlegen wir natürlich mit jeder Menge Beispiel-Setups aus der Praxis, z. B. als Storage-Backend für File-Server und/oder Storage-Backends für vSphere/ESXi-Umgebungen, die in den hier gezeigten oder ähnlichen Varianten bereits in etlichen großen Produktiv-Umgebungen zuverlässig ihren Dienst verrichten. Zunächst werfen wir einen Blick auf das »normale« DRBD 8.x, seine Konzepte, Funktionsweisen und deren praktische Umsetzung, bevor wir uns die Neuerungen wie Multinode-/Mesh-Replication in DRBD 9.x anschauen. Danach geht die Reise dann ins Land der vielseitigen Cephalopoden und seiner verteilten Kollegen. Ans Werk ...

17.1 DRBD – Vorbetrachtungen

Die erste Frage an dieser Stelle lautet schlicht und einfach: Wie können wir zwei physikalisch voneinander getrennte Storage-Einheiten dazu bringen, sich wie ein lokaler Raid-1-Mirror zu verhalten? Die Antwort auf diese Frage ist DRBD, das sogenannte *Distributed Replicated Blockdevice*. Wie bereits vorab kurz erläutert, verhält sich ein DRBD wie ein Raid-1-Mirror, nur dass die Replikation der Daten in diesem Fall nicht lokal über den Device-Mapper erfolgt (und es logischerweise auch nicht kann), sondern protokollbasiert über das Netz.

Das grundlegende Konzept ist vergleichbar mit einem üblichen Raid 1, nur dass hierbei jeder Teil des Mirrors auf einem eigenständigen Server im Netz liegt. Jede DRBD-Spiegelhälfte kann – je nach Setup-Typ – den Status *Primär (Primary)* oder *Sekundär (Secondary)* annehmen, wobei nur auf eine primäre Spiegelhälfte schreibend zugegriffen werden kann, auf eine sekundäre gar nicht – diese Spiegelhälfte lässt sich auch nicht Readonly mounten und wird erst dann mount- und schreibbar, wenn wir (oder z. B. unser Cluster im Fehlerfall auf dem ursprünglichen Primary) sie zu dem neuen Primary heraufstufen (s. u.). In älteren DRBD-Versionen (≤ 0.7x) war nur ein Master/Slave- bzw. Primary/Secondary-Setup möglich, bei dem – im Fehlerfall auf dem Primary – der Secondary zum neuen Primary *promoted* (also »befördert«) wurde, auf den dann schreibend zugegriffen werden kann. Die Schwenkzeiten lagen, aufgrund der konzeptionellen Systematik, natürlich jenseits eines echten Hot-Failovers.

Mit DRBD 8.x (kein Tippfehler, der etwas krude Versionssprung wurde tatsächlich so durchgeführt) wurden auch Dual-Primary-Setups möglich, wodurch auf beide Nodes gleichzeitig schreibend zugegriffen werden kann. Zwingende Voraussetzung für ein derartiges Setup ist natürlich ein clusterfähiges Dateisystem, wie z. B. OCFS2 oder GFS2, mit einem sogenannten *Distributed Lock Manager* (DLM), bzw. ein Dateisystem, das entsprechende systemübergreifende Locking-Mechanismen kennt. Eine weitere Neuerung, die in Version 8.3 hinzugekommen war, war das sogenannte »Stacked«-Device (»gestapeltes« Device): vereinfacht ausgedrückt ein drittes DRBD-Device, das dem normalen 2-Node-DRBD als »Offline-Backup« zur weiteren Erhöhung der Verfügbarkeit hinzugefügt werden konnte. Das clustertechnische Setup und die Failover-Funktionalität für das dritte »Stacked«-Device sind bzw. waren, vorsichtig ausgedrückt, komplex und mehr als hakelig und erfahrungsgemäß für Produktivumgebungen eher weniger geeignet. Mit der neuen Multinode-/Mesh-Replication in Version 9.x sollte dieses Manko behoben sein. DRBD funktioniert zwar auch Standalone, aber die entsprechenden automatischen Failover-Mechanismen können nur in Verbindung mit einem Cluster wirklich (sinn)voll zum Einsatz gebracht werden.

Funktionell betrachtet, kümmert sich auf Kernel-Ebene das DRBD-Modul darum, dass alle Zugriffe, die auf das virtuelle DRBD-Device (z. B. */dev/drbd0*) erfolgen, auf das darunterliegende »echte« Blockdevice gemappt werden – egal ob es sich dabei um eine Disk, eine Partition, ein (Soft-)Raid und/oder ein Logical Volume handelt. Genauso gut kann das DRBD-Device natürlich auch als Unterlage für ein LV(-Raid) verwendet werden. Ab Kernel 2.6.33 wurde DRBD in die Kernel-Mainline aufgenommen, d. h. DRBD steht seitdem als Modul Out-of-the-Box auf Systemen ab dieser Kernelversion zur Verfügung. Die Liste der benötigten Pakete zu diesem Abschnitt findet sich wie üblich im Anhang.

Die zur Verwaltung des DRBDs erforderlichen Tools sind als Userspace-Werkzeuge implementiert, die alle gängigen Distributionen mitbringen. Die Sourcen finden sich

unter *www.drbd.org*. Bevor wir uns die DRBD-Funktionalität genauer anschauen und zu den Setups kommen, werfen wir noch einen kurzen, auszugsweisen Blick auf die konzeptionell wichtigsten Features:

- Unterstützung von verschiedenen Replikationsprotokollvarianten (online umstellbar) mit verschiedenen Geschwindigkeits- und Sicherheitsstufen: *Fully Synchronous (C)*, *Memory Synchronous (B)* oder *Asynchronous Mode (A)*
- einstellbare Bandbreite für die (Re-)Synchronisation
- Auto-Recover nach Node-, Netzwerk- oder Disk-Fehlern
- geringe Deltas im Fall eines Resync, es werden nur die während der Ausfallzeit geänderten Blöcke übertragen
- semi-automatische Split-Brain-Handler
- herstellereigener OCF-RA zur Cluster-Integration
- *Dual-Primary Support* in Verbindung mit Cluster-Dateisystemen
- *Online Data Verification*
- kann auf ein bestehendes LV(-Raid) aufgesetzt werden, ebenso kann DRBD auch als PV für ein LV fungieren
- Scripte für LVM, um automatisch einen Snapshot zu erstellen, bevor der betreffende Node das Ziel einer Resync-Operation wird
- seit Version 8.3 dritter Offline-Node als zusätzliches Backup konfigurierbar, ab Version 9.x Mesh-/Multinode-Replication
- seit Version 8.3.2 Unterstützung von Resource-Level-Fencing-Scripten auf der Basis von Pacemaker-Constraints
- Datentransfer zwischen den Nodes kann mithilfe von Standard-Mechanismen wie IPSec oder OpenVPN verschlüsselt werden.
- DRBD-Devices können über die Standard-Verschlüsselungstools des OS für Blockdevices verschlüsselt werden.

17.1.1 Wozu – und wozu nicht: Einsatzmöglichkeiten von DRBD

Wozu: Eines der häufigsten DRBD-Einsatz-Szenarien sind in der Regel Fileserver, auf denen viele Daten redundant gehostet werden sollen, wie z. B. Samba-Shares. Für Fileservices unter Samba bietet sich beispielsweise auch das Active/Active-Setup im Dual-Primary Mode an. In Verbindung mit CTDB (siehe hierzu auch *http://ctdb.samba.org*) kann Samba ab Version 3.3 z. B. auch als geclusterter Fileserver arbeiten: Es bietet sich dabei ein verteiltes Dateisystem über mehrere (Samba-technisch identisch konfigurierte) Cluster-Nodes nach außen wie ein einziger SMB/CIFS-Server an. Stark vereinfacht zusammengefasst kann man das CTDB-Konzept wie einen Loadbalancing-(Active/Active-)Cluster mit HA-Funktionalität betrachten. Allerdings ist in

Verbindung mit Samba 3 zu beachten, dass das verwendete Cluster-Dateisystem extended Posix-ACLs unterstützt, andernfalls können die Benutzerrechte auf Dateien und Ordner nicht Windows-konform umgesetzt werden. »Relativ« statischer Content – wie z. B. der von Webservern – kann im Rahmen eines Loadbalancer-Setups (Apache mit `mod_proxy(_balancer)`) ebenfalls recht einfach konsistent gehalten werden. Für derartige, in der Regel relativ zeitunkritische Änderungen auf den Websites können unter entsprechenden Voraussetzungen jedoch auch konventionelle Sync-Maßnahmen, wie z. B. **`rsync`**, verwendet werden.

Wozu nicht: Wenn sehr schnelle Failover-Zeiten gefragt sind, z. B. bei Datenbanken oder Verzeichnisdiensten. Zudem kümmern sich Services, wie z. B. MySQL oder OpenLDAP, ohnehin (wie bereits ausführlich erläutert) eigenständig um die Replikation ihrer Daten, und der Einsatz eines DRBD kann – je nach Applikation und Einsatztyp – sogar kontraproduktiv sein. Eine Vergleichsbetrachtung dieser Thematik werden wir genau deshalb in diesem Teil (Storage Cluster) in Abschnitt 17.5 am Beispiel vom MySQL durchführen.

Achtung

In diesen Fällen – und auch in allen anderen, bei denen sich vakante Dienste selbst um die Replikation ihrer Daten kümmern – sollte *immer* den Built-In-Funktionalitäten der jeweiligen Services/Applikationen zur Daten-Synchronisation der Vorzug gegenüber DRBD gegeben werden, denn nur sie sind komplett in der Lage, alle gegebenenfalls vorhandenen und notwendigen Lockings zu berücksichtigen und eine entsprechende Transaktions-Sicherheit zu bieten.

17.1.2 Die DRBD-Funktionalität im Detail

Auf den beteiligten Systemen bzw. Nodes erzeugt DRBD bei einem »konventionellen« Setup zunächst eine Verbindung von dem lokalen Blockdevice (Single Disk/Partition, Raid und/oder LVM/LV-Raid) zu einem virtuellen DRBD-Blockdevice, das z. B. über */dev/drbd<Zahl>*, */dev/drbd/by-disk/<Disk/Partition>* oder */dev/drbd/by-res/<Ressourcen-Name>* angesprochen werden kann. Letztendlich sind die beiden letzteren aber nur Links auf */dev/drbd<Zahl>*. Die Kommunikation zwischen den Cluster-Nodes zur Synchronisation der lokalen DRBD-Devices erfolgt protokollbasiert via TCP, sowohl für IP V4 als auch für IP V6. Ebenfalls möglich, aber in der Praxis wohl eher exotischer, sind die Replikationsvarianten per SDP (RDMA/Infiniband) oder SuperSockets (Dolphin Interconnect Solutions).

Ab Version DRBD 9.x (wahrscheinlich noch nicht in 9.0) soll zusätzlich *UDP Multicast Transport* (anstelle von *TCP Connection Mesh*) unterstützt werden.

In einem typischen Primary/Secondary-Setup werden Schreibzugriffe auf den Primary-DRBD-Node sowohl an das unterliegende Blockdevice, als auch an den Secondary-DRBD-Node propagiert. Alle Lesezugriffe werden stets lokal (auf dem primären System) durchgeführt. Fällt das primäre System aus, versetzt der entsprechende RA das ehemals sekundäre System in den primären Systemzustand. Zusätzlich regeln entsprechende `handlers`-Direktiven innerhalb der DRBD-Konfiguration bei Bedarf die Prozeduren / Verfahrensweisen in bestimmten Fehlersituationen (I/O-Error, Split-Brain usw.).

Das DRBD-Modul arbeitet innerhalb des Linux-Kernels auf Blockebene und ist damit für alle darauf aufsetzenden Schichten transparent. DRBD kann somit u. a. als Grundlage verwendet werden für:

- konventionelle, lokale Dateisysteme, wie z. B. ext3/4, XFS oder BTRFS
- gemeinsam genutzte Cluster-Dateisysteme, wie z. B. GFS2, OCFS2
- ein weiteres logisches Blockdevice, wie z. B. LVM*
- jede Applikation, die den direkten Zugriff auf ein Blockdevice unterstützt

Achtung

Wenn DRBD mit einem unterliegenden LV eingesetzt wird, muss LVM so eingerichtet werden, dass die */dev/drbd**-Devices von LVM während der Bootphase gegebenenfalls mit gescannt und eingebunden werden.

17.1.3 Rollenspielchen – DRBD-Modi

Single-Primary-Mode

Im »klassischen« Single-Primary- bzw. Primary/Secondary-(oder auch Master/Slave-) Mode kann immer nur genau ein Leg des DRBDs den primären, schreibbaren Status auf genau einem Cluster-Node haben. Der sekundäre Node kann den Datenbestand nicht mounten, nicht einmal Readonly. Da in dieser Konfiguration garantiert ist, dass immer nur auf einem Node die Daten manipuliert werden können, kann der Single-Primary-Mode mit jedem konventionellen (nicht clusterfähigen) FS verwendet werden, wie z. B. ext3/4, BTRFS oder XFS. Das typische Einsatzgebiet für diesen Mode wären u. a. Active/Passive- bzw. Failover-/Hot-Standby-Cluster.

Dual-Primary-Mode (ab DRBD 8.x)

Im Dual-Primary-Mode kann *jede* DRBD-Ressource zu jeder Zeit den Primary-(Master-)Mode auf beiden Cluster-Nodes annehmen. Da hierdurch ein zeitgleicher, schreibender Zugriff auf Daten möglich ist, muss zwingend ein Cluster-Filesystem mit entsprechenden Locking-Mechanismen eingesetzt werden, wie z. B. GFS2 oder

OCFS2. DRBD im *Dual-Primary-Mode* ist als Default deaktiviert und muss erst separat in der DRBD-Konfiguration aktiviert werden.

Multinode-/Mesh-Replication (ab DRBD 9.x)

In DRBD 9.x kommt mit dem Feature der Multinode-/Mesh-Replication die Fähigkeit hinzu, multiple Devices als Primary zu definieren und bei Bedarf als solche (Primaries) auch automatisch beim Mountvorgang zu promoten. Einen Blick auf DRBD 9.x, das sich zum Zeitpunkt der Erstellung dieses Buches noch in der Entwicklung befindet, werden wir in Kapitel 18 werfen.

17.1.4 Replikations-Varianten

DRBD unterstützt drei (inoffiziell: vier) verschiedene Replikations-Protokolle, die hinsichtlich Transaktionssicherheit und Geschwindigkeit stark differieren. Wir betrachten sie im Folgenden am Beispiel eines typischen Primary/Secondary-Setups. Natürlich gelten die Vor- und Nachteile der jeweiligen Protokollvariante auch im Rahmen eines Dual-Primary-Setups, nur läuft hier die Synchro eben in beide Richtungen.

Protocol A – Asynchronous replication protocol

Lokale Schreiboperationen auf dem Primary-Node gelten genau dann als abgeschlossen, wenn die eigentliche Schreiboperation auf der Disk abgeschlossen ist und das Replikations-Datenpaket im lokalen TCP-Sendbuffer dieses Nodes platziert ist *(noch nicht gesendet!)*. Im Fehlerfall (Failover) können daraus logischerweise Dateninkonsistenzen resultieren, denn: Die Daten auf dem Secondary-Node sind in der Regel zwar für sich allein genommen konsistent, aber im Gegensatz zu denen auf dem Primary-Node gegebenenfalls veraltet! Protokoll-Variante (A) bietet logischerweise den höchsten Speed, aber auch das größte Potenzial an Daten-Inkonsistenzen. Sie könnte z. B. in einem 3-Node-DRBD für den dritten »Offline«-Node verwendet werden, der als Backup dient. Dieses Setup ist allerdings, wie bereits erwähnt, sowohl von der Cluster- als auch von der DRBD-Konfiguration her für Produktivumgebungen eher ungelenk und wird in kommenden DRBD-Versionen ohnehin durch die weitaus praktikablere Multinode-/Mesh-Replication abgelöst.

Protocol B – Memory synchronous (semi-synchronous) replication protocol

Lokale Schreiboperationen auf dem Primary-Node gelten dann als abgeschlossen, wenn die eigentliche Schreiboperation auf der Disk abgeschlossen ist und das Replikations-Datenpaket den anderen Node *erreicht* hat. Diese Protokoll-Variante (B) ist etwas langsamer als Protokollvariante A, dafür entsteht im Fall eines Failovers in der Regel kein Datenverlust. Die einzige Möglichkeit wäre dann gegeben, wenn beide Nodes auf einmal ausfallen, und die Daten auf der Partition des Primar-Nodes

geschreddert sind. Aber selbst dann würden auf dem Secondary nur die allerletzten Schreiboperationen des Primary fehlen; die Inkonsistenz im Vergleich zum Primary-Node wäre relativ gering. Trotzdem: nichts für Produktivumgebungen.

Protocol C – Synchronous replication protocol

Safety First: Lokale Schreiboperationen auf dem Primary-Node gelten genau dann als abgeschlossen, wenn die eigentliche Schreiboperation auf der lokalen Disk *und* – nach erfolgtem Transfer – auf der Remote Disk ebenfalls abgeschlossen *und bestätigt* ist. Diese Variante bietet die höchste Ausfallsicherheit, die ohne Inkonsistenzen und bei voller Datenintegrität jederzeit den Verlust eines Nodes verkraften kann. Natürlich sind die Latenzzeiten in dieser Variante etwas größer als in A und B; aufgrund der weitaus höheren Datensicherheit sollte im Praxisbetrieb jedoch immer dieser Protokoll-Variante (C), dem synchronen Replikationsprotokoll, der Vorzug gegeben werden.

Protocol D (Remus, inoffiziell)

Bei dieser inoffiziellen Replikationsvariante handelte es sich um eine für Remus (Xen-HA-Lösung, siehe Kapitel 26, »Virtualisierung im Cluster«) gepatchte Protokoll-A-Variante (Funktionslevel: DRBD 8.3), die in diesem Zusammenhang verwendet wird/wurde, um mithilfe von DRBD die Shadow-Copy einer Xen-VM synchron zu halten.

(Re-)Synchronisation der Nodes

Nicht nur im F-Fall: Die Synchro ist gegebenenfalls auch dann erforderlich, wenn ein Node längere Zeit z. B. zu Wartungszwecken offline war und nun – wie ein konventionelles Raid – seine Daten wieder mit dem intakten bzw. aktiven Node abgleichen muss. Während dieser Phase finden natürlich jede Menge Leseoperationen auf dem Primary-Node statt, was die Performance für die anderen, darauf zugreifenden Applikationen herunterbremst. Der Secondary-Node befindet sich bis zum vollständigen Abgleich mit dem Primary in einem inkonsistenten Zustand.

Um das Netz und den Primary-Node während dieser Phase nicht zu sehr zu belasten, kann die DRBD-Synchronisationsrate über die Konfiguration angepasst werden. Eine Änderung der Resync-Rate kann auch im laufenden Betrieb durch Neu-Einlesen der Konfiguration per `drbdadm adjust <Ressource>` erfolgen, oder direkt per `drbdadm disk-options --resync-rate=<Rate> <DRBD-Ressource>`.

Ein weiterer Punkt, den wir berücksichtigen sollten, ist der initiale Datenabgleich: Sollen unsere Storages ein recht beachtliches Datenvolumen beherbergen, könnte die normale Initial-Synchro unser DRBD – je nach Anbindung und Bandbreite – recht lange ausbremsen. Eine Möglichkeit wäre in diesem Fall der kalte Offline-Sync, heißt: Wir befüllen beide Storages auf den Nodes mit einem identischen Datenbestand und nehmen unser DRBD erst anschließend in Betrieb. Dabei muss natürlich sicherge-

stellt sein, dass während dieser Zeit kein Zugriff auf den/die DRBD-Nodes erfolgen kann und dass der übliche Initial-Sync des DRBDs übersprungen wird, z. B. per **drbdsetup --assume-clean** oder:

```
#> drbdadm -- --clear-bitmap \
   new-current-uuid  <Ressource-Name>
```

17.2 DRBD-Standalone-Setup

Um die Funktionsweise des DRBDs zu verstehen, müssen wir uns zunächst anschauen, wie unser *Distributed Replicated Blockdevice* Standalone seinen Dienst auf weiter Server-Flur verrichtet, heißt also: ohne Cluster-Einbindung. Nachdem wir verstanden haben, wie der manuelle Failover bzw. das Demote/Promote der Ressource funktioniert, können wir auch detailliert nachvollziehen, was der OCF-RA im später folgenden DRBD-Cluster-Setup für uns vollautomatisiert bewerkstelligt. Im folgenden Setup verwenden wir die zum Zeitpunkt der Erstellung des Buches aktuelle stabile DRBD-Version 8.4.2 sowohl für das korrespondierende Kernelmodul (seit 2.6.33 in der Kernel-Mainline vorhanden), als auch für die Userspace-Tools. Die aktuellen DRBD-Sourcen können z. B. über *http://www.drbd.org/download/mainline* heruntergeladen werden. Dort finden sich auch Verweise auf distributionsspezifische Paketierungen. Alle im Folgenden verwendeten Pakete sind wie üblich im Anhang gelistet.

DRBD-Versions-Hints

Aktuell ist die bereits benannte DRBD-Version 8.4.x, die auch als Default im SLES11 SP2 anzutreffen ist. Ubuntu 12.04 LTS verwendet Out-of-the-Box beispielsweise Version 8.3.x, Updates sind z. B. über das Repo *ppa:icamargo/drbd* möglich:

```
#> apt-add-repository ppa:icamargo/drbd
#> aptitude update
#> aptitude upgrade drbd8-utils
#> aptitude install drbd8-module-source
#> module-assistant auto-install drbd8
#> modprobe drbd
#> modinfo drbd
...
version:        8.4.3
```

Für DRBD Version 8.3 liegt auch eine Beispieldatei (*drbd83.conf*) den Daten zu diesem Buch bei. Für alle, die ein Up- oder Downgrade ihrer jeweiligen DRBD-Versionen planen, ist folgendes unbedingt zu beachten: Die Metadatenformate zwischen Version 8.3 und 8.4 sind weitestgehend identisch, mit Ausnahme z. B. des Activity-Logs.

Weitere Infos zu den (In-)Kompatibilitäten und Änderungen zwischen den Versionen finden sich unter

http://www.drbd.org/users-guide/s-recent-changes-config.html.

Zunächst müssen wir sicherstellen, dass das DRBD-Modul geladen ist. Die permanente Einbindung kann unter Ubuntu z. B. via */etc/modules* erfolgen, bei SUSE über */etc/sysconfig/kernel* und den dort zu setzenden Parameter `MODULES_LOADED_ON_BOOT`. Ein **`lsmod | grep drbd`** sollte uns zeigen, ob das Modul geladen ist, falls dies noch nicht geschehen ist, hilft ein schnelles **`modprobe drbd`**; per **`modinfo`** können wir einen Blick auf seine Version werfen.

Hinweis

Das Laden des DRBD-Moduls erfolgt in der Regel – sowohl über die distributionsspezifischen Start-Scripte, als auch über den OCF-Agenten – automatisch.

17.2.1 drbd.conf – die Schaltzentrale

Dreh- und Angelpunkt der DRBD-Konfiguration ist die Datei *drbd.conf(5)*, die auf beiden Nodes existieren und vor allem identisch sein muss, denn das Userspace-Tool **`drbdadm`**, mit dem wir unser DRBD primär verwalten, holt sich dort alle relevanten Informationen. In der *drbd.conf* existieren verschiedene Haupt-Sektionen, die wir im Folgenden der Reihe nach anhand der wichtigsten Direktiven durchgehen und später auf unsere Bedürfnisse anpassen werden. Die Konfigurations-Dateien finden sich natürlich wie üblich in den Beispieldaten zu diesem Abschnitt. Nun zu den Abschnitten im Einzelnen:

- `global` – wie das Keyword unschwer vermuten lässt: *globale* Konfigurationsdirektiven. Die globale Sektion kann nur einmal gesetzt werden, und dies sollte direkt am Beginn der *drbd.conf* geschehen. Über `global` werden z. B. der `minor-count` (Anzahl der managebaren Ressourcen, ohne das Modul neu laden zu müssen – Default: definierte Ressourcen plus 11, maximal 32), `dialog-refresh` (user-Dialog-Refresh-Time in Sekunden, Default: 1) und `disable-ip-verification` (Überprüfung der Peer-IP per **`ip`**/**`ifconfig`**) gesetzt. Wer keine Lust auf Linbits etwas nervige Phone-Home-Aktivitäten bei der Initialisierung eines DRBD-Devices hat, sollte dies per `usage-count no`; abschalten. Die `global`-Sektion kann auch über die Datei */etc/drbd.d/global_common.conf* ausgelagert werden. Gleiches gilt für die Direktiven der nachfolgend beschriebenen `common`-Sektion, auch sie können in der *global_common.conf* angegeben werden. Diese Datei kann – wie auch separate DRBD-Ressourcen-Dateien, z. B. *.res – über entsprechende `Include`-Statements in die *drbd.conf* inkludiert werden.

- `common` – Alle definierten Ressourcen erben die in dieser Sektion getroffenen Einstellungen. `common`-Settings sind nicht zwingend vorgeschrieben, jedoch wird das Setup – insbesondere bei der Verwaltung multipler DRBD-Ressourcen – erheblich übersichtlicher. Unter `common` können Settings für die `startup`, `options`, `handlers`, `net` und `disk`-Sektion getroffen werden, betrachten wir daher also zunächst die gerade aufgeführten die Common-Subsektionen:
- `startup` – In dieser Subsektion findet das »Feintuning« der DRBD-Settings statt. So können hier unter anderem Timeouts für verschiedene Situationen festgelegt werden, z. B. `wfc-timeout` (wfc = *wait for connection:* gilt, wenn der Peer-Node noch online und konsistent war, bevor der aktuelle Node rebootet wurde), `degr-wfc-timeout` (wenn der Peer-Node beim Reboot des aktuellen bereits offline war), `outdated-wfc-timeout` (s. o., jedoch mit einen Peer, der »outdated« war), und wie groß der maximale Timeout nach einer Split-Brain-Situation sein darf (`wait-after-sb`). Als Default ist für alle Direktiven kein Zeitlimit gesetzt: warten bis zum Jüngsten Tag ist also die Vorgabe. In der `startup`-Sektion wird ebenso festgelegt, welche Nodes beim Startup Primary werden dürfen (`become-primary-on`). Weitere detaillierte Infos zu den oben angegebenen Subdirektiven liefert *drbd.conf(5)*.
- `disk` – Über diese Subsektion können unter anderem die Parameter für die Synchronisation definiert werden. Wichtige Parameter an dieser Stelle wären z. B. `resnyc-rate <Speed>` (ehemals `syncer`-Subsektion, Direktive: `rate`), denn hierüber wird festgelegt, welche maximale Bandbreite (Default: 250 KB/sec – *Achtung: Byte, nicht Bit*) DRBD auf der zugewiesenen Verbindung nutzen darf. Dies spielt insbesondere dann eine wichtige Rolle, wenn über die der vakanten IP zugeordneten Leitung andere Prozesse ebenfalls Daten austauschen, damit diese nicht völlig ausgebremst werden. Mit `resync-after <ressource-name>` kann festgelegt werden, dass ein Resync auf diesem Device z. B. erst dann initiiert wird, wenn sich die betreffende Minor-Ressource bereits in einem *connected*-State befindet. Andernfalls wartet das Device im Status *SyncPause*. Per `cpu-mask` kann in hexadezimaler Notation festgelegt werden, ob die DRBD-Kernel-Threads über alle CPUs (Default: 0, wird auf alle verteilt) verteilt wird. Hinzu kommen Direktiven wie z. B. `on-io-error <handler>`, über die festgelegt werden kann, wie sich das DRBD verhalten soll (z. B. `pass_on` oder `detach`), wenn das Low-Level-Device einen Fehler an die höheren DRBD-Layer meldet. Weitere Parameter wären z. B. in Bezug auf das Activity-Log die Direktiven `al-extents` und `al-updates`. Über `fencing` kann die Policy festgelegt werden, wie sich das DRBD im Split-Brain-Fall verhalten soll. Möglich sind hier Handler-Werte wie `dont-care` (ignorieren), `resource` (Ressource-Fencing per Handler `fence-peer`) und das selbsterklärende `resource-and-stonith`. Load-Balancing-relevante Settings können z. B. per `read-balancing <method>` getroffen werden.

- `net` – Ebenfalls eine Subsektion zum Feintuning der Netzwerk-spezifischen Parameter. Hierüber können z. B. Buffer(-Size)-spezifische Einstellungen festgelegt werden (`protocol`, `sndbuf-size`, `rcvbuf-size`, `max-buffers`), verschlüsselungsspezifische Settings (`cram-hmac-alg`, `shared-secret`) sowie der Startup von zwei Primarys (`allow-two-primaries`) erlaubt werden. Daneben können u. a. Policies für das Split-Brain-Handling festgelegt werden (`after-sb-0pri`, `after-sb-1pri`, `after-sb-2pri`), die wir uns im Rahmen eines konkreten Dual-Primary-Setups noch genau anschauen werden.
- `handlers` – Über die *handlers*-Subsektion können sogenannte »Handler« definiert werden, die im Fall eines bestimmten Ereignisses – wie der Name schon vermuten lässt – eventbasiert getriggert werden. Typische Handler sind z. B. Prozeduren, die auf Split-Brain-Events (`pri-lost-after-sb`) oder inkonsistente (`pri-on-incon-degr`) DRBD-Nodes reagieren können.
- `resource <name>` – Die `resource`-Sektion legt die Eigenschaften einer namentlich explizit definierten DRBD-Ressource fest. Wie bereits erwähnt, können hier ebenfalls alle Ressourcen-spezifischen Einstellungen getroffen werden, die unter `common` möglich sind. Werden hier keine Einstellungen getroffen, z. B. zu `handlers`, `options`, `disk`, `net` und `startup`, gelten – insofern vorhanden – die Einstellungen der `common`-Sektion.

  Zwingend notwendig sind hier in jedem Fall die Spezifikation des Übertragungs-Protokolls (A, B oder C) sowie die Einrichtung von zwei sogenannten Host-Sektionen (`on <hostname>`). Sie legen die eigentlichen DRBD-Legs unseres Replicated Blockdevice fest. Minimal erforderlich sind dort der jeweilige Hostname des Nodes, seine IP-Adresse und der Port, über den die DRBD-Kommunikation abgewickelt wird, der Bezeichner des Blockdevices (z. B. `device /dev/drbd0`) auf dem jeweiligen Node sowie das dem DRBD-Device unterliegende Blockdevice (für eine Single Disk Partition z. B. `device /dev/sdb1`, für Softraids z. B. `device /dev/md0` oder für LVs z. B. `device /dev/VG1/LV1`). Der Parameter `meta-disk <internal|external>` legt fest, *wie*, bzw. genauer, *wo* die Meta-Daten (die unser DRBD definieren/beschreiben) gespeichert werden. Im oben angegebenen Beispiel also direkt auf unserem DRBD-Device. Die Meta-Daten des DRBD enthalten dabei u. a. die Größe des DRBD und das Activity-Log und werden bei der (empfohlenen, aber etwas langsameren) Setup-Variante `meta-disk internal` in bestimmten, reservierten Bereichen am Ende des Devices abgelegt. Im Fall von `external` können die Meta-Daten auf externen Disks untergebracht werden.

Die genaue Berechnung der Größe des für die Metadaten benötigten Bereiches ist z. B. hier beschrieben *http://www.drbd.org/users-guide/ch-internals.html#s-meta-data-size*. Weitere Infos zu allen vorgenannten und weiteren Konfigurationsoptionen liefert natürlich *drbd.conf(5)* sowie **`drbdsetup(8)`**. Schauen wir uns nun eine sehr

einfache Master/Slave- bzw. Primary/Secondary-Konfiguration für die Ressource `r0` an, die konkret auf unsere Belange angepasst ist:

```
global {
   dialog-refresh  1;
   minor-count  5;
   usage-count no;
}
common {
}
resource r0 {
   protocol  C;
   disk {
        on-io-error pass_on;
        resync-rate 100M;
   }
   net {
   }
   startup {
   }
   on jake {
      device    /dev/drbd0;
      address   10.0.199.12:7788;
      meta-disk internal;
      disk      /dev/sdb1;
   }
   on elwood {
      device    /dev/drbd0;
      address   10.0.199.13:7788;
      meta-disk internal;
      disk      /dev/sdb1;
   }
}
```

Wie wir unschwer erkennen können, liegt unserem DRDB auf jedem Leg die jeweils lokale Partition */dev/sdb1* zugrunde. Die Kommunikation bzw. Synchro der beiden DRBD-Devices läuft über unser (optimalerweise gebündeltes) 10er-Netz via Port 7788.

17.2.2 Die DRBD-Userspace-Tools

Werfen wir an dieser Stelle einen kurzen Blick auf die DRBD-Tools, die uns zur Administration unserer DRBD-Devices zur Verfügung stehen – **`drbdadm`** und **`drbdsetup`**. Wie die namentliche Kennzeichnung schon vermuten lässt, dient das Tool **`drbdadm(8)`**

primär zur Administration des bereits eingerichteten DRBDs, sein Kollege **drbdsetup(8)** zur Einrichtung und Änderung von Konfigurationsparametern desselben. Dabei stellt **drbdadm** das High-Level-Tool dar, **drbdsetup** sein Low-Level-Pendant: **drbdadm** liest die Konfigurationsdatei aus und führt die notwendigen Operationen aus, indem es **drbdsetup** und/oder **drbdmeta** aufruft – aus diesem Grund existieren auch viele ähnliche Optionen in beiden Tools, wie z. B.:

```
#> drbdsetup /dev/drbd0 role
#> drbdadm role r0
```

Betrachten wir kurz einige der wichtigsten Befehlsparameter anhand von **drbdadm**, mit dem wir sowohl eine gezielte Ressource (in unserem Fall `r0`) als auch alle definierten Ressourcen ansprechen können:

```
#> drbdadm < befehl >  < lokale drbd-resource(n) | all >
```

Die wichtigsten **drbdadm**-Befehle schlüsseln sich dabei wie folgt auf:

- **create-md** – das DRBD mit frischen Metadaten initialisieren
- **wipe-md** – Metadaten löschen
- **attach** – das unterliegende Blockdevice mit DRBD-Device verbinden
- **detach** – das unterliegende Blockdevice trennen
- **connect** – zwei konfigurierte DRBD-Nodes verbinden
- **disconnect** – die DRBD-Nodes voneinander trennen
- **up** – Kurzform für **attach** und **connect**
- **down** – Kurzform für **disconnect** und **detach**
- **primary** – den aktuellen Node in primären Zustand versetzen
- **secondary** – den aktuellen Node in sekundären Zustand versetzen
- **invalidate(-remote)** – Resync auf aktuellen oder entfernten Node erzwingen
- **syncer** – Resynchronisations-Parameter neu laden (bis Version 8.3)
- **resize** – Größe der DRBD-Ressource anpassen, wenn das unterliegende Device (z. B. ein Logical Volume) verändert (größer/kleiner gemacht) wurde
- Zudem stehen noch die (zum Teil) undokumentierten **hidden-commands** zur Verfügung.

Ausführlichere Infos zu **drbdsetup** und **drbdadm** liefern die jeweiligen Manpages – aber darauf ruhen wir uns natürlich nicht aus, sondern arbeiten mit den Tools im Folgenden an konkreten Setup-Szenarien.

17.2.3 Setup der DRBD-Devices

Im ersten Schritt kümmern wir uns logischerweise um die Einrichtung der lokalen Disks. Wie bereits erwähnt: Wir können nicht nur normale Disks verwenden, sondern auch die lokale Redundanz und Flexibilität erhöhen, indem wir das jeweils lokale DRBD-Device als Softraid abbilden und es wahlweise mit einem LV unter – oder auf – dem DRBD kombinieren. Ebenso sind natürlich auch Setup-Varianten mit LV-Raids möglich. In den folgenden Abschnitten werden wir auch diese Varianten mit ihren Vor- und Nachteilen erörtern. Für dieses erste, einfache Setup wählen wir jedoch eine normale Disk, die nur aus einer Partition besteht (ID 0x83 Linux). Stellt sich an dieser Stelle natürlich die Frage: Funktioniert auch ein partitionierbares DRBD? Klare Antwort: Jein! Daher:

Exkurs: To part or not to part... partitionierbare DRBDs?

Zunächst sollten wir uns – wie bei allen Problematiken – immer zuerst damit beschäftigen, was uns ein bestimmtes Setup und/oder ein Feature konkret an Nutzen im Verhältnis zum Aufwand und unserem immer präsenten *KIS(S)*-Motto bringt. Fahren wir vorab also im ersten Schritt eine kurze Machbarkeits-/Aufwands-/Nutzen-Studie. Zunächst die Machbarkeit: Ja, es geht. Mit geringem Aufwand realisierbar? Eher weniger. Was würde uns das partitionierbare DRBD bringen? Prinzipiell wenig, denn ein *LV on top of DRBD* bringt uns wesentlich mehr Vorteile mit geringerem Aufwand. Wer sich dennoch daran versuchen will: Im DRBD-Setup komplette Disks verwenden (z. B. */dev/sdb*), nach dem Initial-Sync des DRBDs das Device */dev/drbd0* per **fdisk** »normal« partitionieren. Danach per

```
#> kpartx -a -v /dev/drbd0
```

das Drive-Mapping konfigurieren. Die Partitionen können bei korrekt initialisiertem DRDB (Primary/Secondary) nur auf dem aktiven, bzw. bei einem Dual-Primary-Setup auf beiden DRBD-Legs per */dev/mapper/drbd0p1|2|3* usw. angesprochen werden. Natürlich dürfen bei dieser Art des Setups entsprechende Prozeduren nicht vergessen werden, die sich um die Reboot-Persistenz von den per **kpartx** generierten Subdevices kümmern. Weiter im Text: Beginnen wir nun mit dem eigentlichen DRBD-Setup. Hierbei werden die DRBD-Legs auf unseren beiden Nodes entsprechend den in der *drbd.conf* hinterlegten Konfigurationsparametern erstmalig initialisiert. Im ersten Schritt erzeugen wir per **drbdadm** unsere eigentliche DRBD-Ressource `r0`, und zwar auf beiden Legs:

```
#> drbdadm create-md r0
...
Writing meta data...
initialising activity log
NOT initialized bitmap
New drbd meta data block sucessfully created.
```

Falls es bei diesem Schritt zu Problemen mit »alten/störrischen« Metadaten kommen sollte (z. B. durch alte (c)LVM- und/oder Raid-Metadaten oder eine vormalige DRBD-Konfiguration), können wir uns dieser lästigen Überbleibsel z. B. per **drbdmeta(8)**, dem DRBD-Tool zum Management der DRBD-Metadaten, entledigen:

```
#> drbdmeta 0 v08 /dev/sdb1 internal wipe-md  --force
Do you really want to wipe out the DRBD meta data?
*** confirmation forced via --force option ***
Wiping meta data...
DRBD meta data block successfully wiped out.
```

oder alternativ auch per **drbdadm wipe-md <[resource]|all>**. Hat bis zu diesem Punkt alles geklappt, sollte dem manuellen Start unseres DRBDs nichts mehr im Wege stehen. Da nahezu alle Distributionen ein entsprechendes init-/systemd-/upstart-Script für DRBD besitzen, können wir z. B. per:

```
#> /etc/init.d/drbd start
```

oder:

```
#> service drbd start
```

den DRBD-Service auf jedem Node abfeuern. Alternativ können wir das DRBD – pro Leg – auch direkt per:

```
#> drbdadm up r0
```

dazu nötigen, sich mit seinem jeweiligen Peer zu syncen und zu konnektieren. Hierzu muss allerdings sichergestellt sein, dass das DRBD-Modul zuvor geladen wurde. Das Subkommando **up** beinhaltet dabei die Subkommandos **attach** und **connect**. Nach erfolgreichem Connect sollte uns der DRBD-Status unter */proc/drbd* in etwa folgenden Output zeigen:

```
#> cat /proc/drbd
version: 8.4.2 (api:1/proto:86-101)
GIT-hash: 7ad5f850d711223713d6dcadc3dd48860321070c build by root@lixia, 2012-09-19 16:40:30
 0: cs:Connected ro:Secondary/Secondary ds:Inconsistent/Inconsistent C r-----
    ns:0 nr:0 dw:0 dr:0 al:0 bm:0 lo:0 pe:0 ua:0 ap:0 ep:1 wo:f oos:20969820
```

Die beiden Paare des Outputs, die für uns zunächst am wichtigsten sind, springen uns direkt ins Auge – `ro:Secondary/Secondary` und `ds:Inconsistent/Inconsistent`. Was bedeutet das en détail? Ganz einfach: `ro` steht für die *Role* des jeweiligen Legs (das links stehende ist dabei immer das Leg, auf dem der Befehl ausgeführt wurde, also der lokale DRBD-Node), in unserem konkreten Fall *Secondary/Secondary*. Na prima – hatten wir nicht ein Master/Slave- bzw. Primary/Secondary-Setup auf unserer Wunschliste? Sicher, aber die beiden Schlüsselwörter *Inconsistent/Inconsistent*, die den Sync-Status unseres DRBDs anzeigen, sprechen ebenfalls eine klare Sprache. Der Grund dafür liegt auf der Hand: Wir haben unser DRBD initial bzw.

frisch erstellt – was ihm fehlt, ist unser Kommando, das ihm sagt, wer der Boss im nagelneuen DRBD-Stall ist. Die Ausgangssituation ist in diesem Fall noch undefiniert, und in dem von uns gewählten Setup-Typ kann es schließlich nur einen Master geben. Also tun wir ihm den Gefallen und können das DRBD-Leg von Node *jake* z. B. per **drbdsetup**-Kommando zum primären DRBD ernennen (Achtung: für ein leeres, neues DRBD gibt es eine deutlich effizientere und schnellere Variante, siehe nächster Kasten *Skip Initial Sync*):

```
jake:~# drbdsetup /dev/drbd0 primary --overwrite-data-of-peer
```

Wie wir unschwer erkennen können, werden die Daten auf dem DRBD-Device des Peers *(elwood)* mit denen des Nodes, auf dem das Kommando abgesetzt wurde, überschrieben. Per (**watch**) **cat /proc/drbd** können wir den Sync-Vorgang gut beobachten (zweite Zeile umbrochen):

```
0: cs:SyncSource ro:Primary/Secondary ds:UpToDate/Inconsistent C r---n-
    ns:4031140 nr:0 dw:0 dr:4039320 al:0 bm:245 lo:0 pe:2 ua:8 ap:0 ep:1 wo:f oos:16940380
        [==>.................] sync'ed: 19.3% (16540/20476)M
        finish: 0:03:20 speed: 84,536 (60,140) K/sec
```

Node *jake* fungiert nun dank unserer manuellen Promotion als Primary, ist *UpToDate* und *SyncSource* für *elwood*s DRBD-Leg.

Hinweis: Skip initial Sync

Erstellen wir unser DRBD initial, d. h. mit leeren Disks, können wir uns die ganze Mimik und das Kaffetrinken bis zum vollständigen Resync sparen, indem wir beide DRBD-Legs wie bereits zuvor beschrieben als »clean« und »consistent« markern, und das natürlich nur auf *einem* der beiden Nodes, und diesen (hier: *jake*) dann manuell zum neuen *Primary* promoten:

```
jake #> drbdadm -- --clear-bitmap  new-current-uuid  r0
jake #> drbdadm primary r0
```

Ist die Sync-Rate sehr niedrig und haben wir auf unseren Netzdevices noch bandbreitenmäßig Luft nach oben, können wir die `resync-rate` in der *drbd.conf* auf beiden Nodes innerhalb eines zulässigen und sinnvollen Rahmens nach oben schrauben und unser DRBD mit Hilfe eines

```
#> drbdadm adjust r0
```

auf beiden Nodes umgehend über die persistenten Änderungen in Kenntnis setzen. Wollen wir den Syncspeed nur temporär ändern, ist dies ebenfalls möglich, z. B. per:

```
#> drbdadm disk-options --resync-rate=<rate> r0
```

Über

```
#> drbdsetup /dev/drbd0 show | grep rate
    resync-rate   102400k; # bytes/second
```

können wir uns den Erfolg der Aktion sofort anzeigen lassen. Der **show**-Sub-Parameter liefert uns – bei gestartetem DRBD – einen Überblick über die aktuell gesetzten Einstellungen, inklusive der nicht von uns explizit definierten Werte mit ihren Default-Einstellungen. Nach abgeschlossenem Sync sollten sich unsere DRBD-Legs in */proc/drbd* wie im nun folgenden Exkurs präsentieren.

Exkurs: drbd-overview und /proc/drbd

Ein nützliches Tool, mit dem wir uns schnell einen Überblick über den DRBD-Status verschaffen können, ist **drbd-overview** (auch kein Voodoo-Tool – es bereitet lediglich Informationen aus */proc/drbd* etwas übersichtlicher auf). Hier die Status-Ausgabe:

```
#> drbd-overview
  0:r0  Connected Primary/Secondary UpToDate/UpToDate C r----
```

Die ausführlichere Variante liefert uns nach wie vor */proc/drbd*.

```
#> cat /proc/drbd
0: cs:Connected ro:Primary/Secondary ds:UpToDate/UpToDate C r----
    ns:4 nr:0 dw:0 dr:532 al:0 bm:1 lo:0 pe:0 ua:0 ap:0 ep:1 wo:b oos:0
```

Werfen wir nun einen kurzen Blick auf die wichtigsten Felder von */proc/drbd*, alle Informationen hierzu liefert u. a. *http://www.drbd.org/users-guide/ch-admin.html#s-proc-drbd*. Legende (Auszüge):

- `cs:`

  liefert den *Connection-State*. Dieser kann ebenfalls per **drbdadm cstate <Ressource>** für den jeweiligen Node abgefragt werden. Mögliche Werte für `cs`:

  - `StandAlone` – keine Netzverbindung zwischen den beiden DRBD-Legs; entweder durch Verbindungsabbruch, administrativen Disconnect (`drbdadm disconnect <resource>`), Split-Brain oder fehlerhafte Authentifizierung
  - `Disconnecting` – nur während der Diskonnektierungsphase, danach sofort: `StandAlone`
  - `Unconnected` – wird gegebenenfalls vor der Verbindungsaufnahme temporär kurz angezeigt; danach meistens: `WFConnection` / `WFReportParams`
  - `Timeout` – wird während eines Timeouts temporär angezeigt; danach folgt in der Regel `Unconnected`
  - `BrokenPipe` | `NetworkFailure` | `ProtocolError` – temporäre Anzeige, nachdem die Verbindung zum Peer unterbrochen ist; danach folgt in der Regel ebenfalls: `Unconnected`

- `TearDown` – temporäre Anzeige, während der Peer die Verbindung unterbricht; danach ebenfalls: `Unconnected`
- `WFConnection` – Wait for Connection (bis der Peer im Netz sichtbar ist)
- `WFReportParams` – TCP-Connection etabliert, der Node wartet auf das erste Paket vom Peer
- `Connected` – Normalzustand: Verbindung etabliert, Spiegelung aktiv
- `StartingSyncS/StartingSyncT` – Full Sync, vom Admin initiiert, wird gestartet (S = Source, T = Target). Danach folgt i. d. R.: `SyncSource/PausedSyncS/PausedSyncT`.
- `WFBitMapS/WFBitMapT` – partieller Sync startet gerade (S = Source, T = Target); danach: `SyncSource/PausedSyncS/PausedSyncT`
- `WFSyncUUID` – auf Syncbeginn warten; danach folgt in der Regel: `SyncTarget`
- `SyncSource` – Sync läuft, der lokale Node ist die Quelle
- `SyncTarget` – Sync läuft, der lokale Node ist das Ziel
- `PausedSyncS` – der lokale Node ist die Quelle des laufenden Sync, die jedoch gerade pausiert (was z. B. an der Wartestellung hinter einer anderen laufenden Sync liegen kann, oder an der manuellen Initiierung durch das Kommando **`drbdadm pause-sync <resource>`**)
- `PausedSyncT` – s. o., lokaler Node ist jedoch das Ziel
- `VerifyS` – die *Online Device Verification* läuft gerade, lokaler Node ist die Verifikations-Quelle
- `VerifyT` – s. o., lokaler Node ist das Verifikations-Ziel

▶ `ro`:

liefert die *Roles* (Rollen) der korrespondierenden Devices. Diese können ebenfalls per `drbdadm role <Ressource>` abgefragt werden.

Hinweis

An dieser Stelle noch einmal der Hinweis: Die *lokale* Ressource wird immer als Erstes angezeigt, als Zweites die Remote-Ressource des Peers.

Mögliche Werte für `ro`:

- `Primary` – die Ressource befindet sich im Read/Write-Mode. Kann nur im Dual-Primary Mode auf beiden Nodes gleichzeitig aktiv sein.
- `Secondary` – die Ressource befindet sich im Readonly-Mode. Kann je nach Zustand des DRBDs auf beiden Nodes gleichzeitig vorliegen.
- `Disconnecting` – nur während der Diskonnektierungsphase, danach folgt in der Regel sofort: `StandAlone`

- `Unconnected` – wird gegebenenfalls vor der Verbindungsaufnahme temporär kurz angezeigt. Danach folgt meistens: `WFConnection` / `WFReportParams`.
- `Timeout` – wird während eines Timeouts temporär angezeigt. Danach: `Unconnected`.
- `Unknown` – die Rolle der Ressource ist unbekannt. Dies trifft immer nur auf den jeweiligen Peer (im Disconnected Mode) zu, nie auf den lokalen Node.

▶ `ds`:

liefert die *Disk-States*. Kann ebenfalls per **`drbdadm dstate <Ressource>`** abgefragt werden.

- `Diskless` – dem DRBD Modul wurde aktuell kein lokales Blockdevice zugewiesen (Grund: die Ressource war noch nie »attached«, manuell erfolgter »Detach« oder Auto-Detach durch I/O-Error)
- `Attaching` – Übergangstatus, während die Meta-Daten gelesen werden
- `Failed` – Übergangsphase nach einem I/O-Error des lokalen Blockdevices; nächster Status: `Diskless`
- `Negotiating` – Übergangsphase während der Verbindung
- `Inconsistent` – inkonsistente Daten. Gründe: z. B. neue Ressource auf beiden Nodes vor einem Full Sync. Entspricht ebenfalls der Anzeige während des Sync-Vorgangs.
- `Outdated` – Daten konsistent, aber nicht mehr aktuell
- `DUnknown` – Status des Peers unbekannt (z. B. durch Netzwerkfehler)
- `Consistent` – konsistente Daten auf dem DRBD-Node, jedoch ohne Connection zum Peer. Sobald die Connection etabliert ist, erfolgt die Rückmeldung, ob die Daten *UpToDate* oder *OutDated* sind.
- `UpToDate` – konsistent, alle Daten sind Up-To-Date: der normale Status, der auf beiden Nodes vorliegen sollte.
- Die Bedeutung der restlichen Felder:
- `ns` (*network send*) – Menge der Daten in Kilobytes, die bisher über das Netz zum Peer gesendet wurden
- `nr` (*network receive*) – Menge der Daten in Kilobytes, die bisher über das Netz vom Peer empfangen wurden
- `dw` (*disk write*) – über das Netz empfangene Daten in Kilobyte, die auf die lokale Disk geschrieben wurden
- `dr` (*disk read*) – über das Netz gesendete Daten in Kilobyte, die von der lokalen Disk gelesen wurden
- `al` (*activity log*) – Anzahl der Updates des Aktivitätslogs im Meta-Datenbereich
- `bm` (*bit map*) – Anzahl der Updates der Bitmap-Area des Meta-Datenbreiches

- `lo` (*local count*) – Anzahl der offenen Requests zum lokalen I/O-Subsystem, die durch DRBD initiiert wurden
- `pe` (*pending*) – Anzahl der zum Peer verschickten Requests, die bisher nicht beantwortet wurden (ausstehende Empfangsbestätigung)
- `ua` (*unacknowledged*) – Anzahl der Requests, die der Peer über das Netz empfangen, jedoch noch nicht bestätigt/beantwortet hat
- `ap` (*application pending*) – Anzahl der Block-I/O-Requests, die zum DRBD gesendet, jedoch noch nicht beantwortet wurden
- `ep` (*epochs*) – Anzahl der *epoch*-Objekte (Seit DRBD 8.2.7) – üblicherweise nur 1 (eines). Kann bei I/O-Last u. U. höher liegen, wenn entweder `barrier` oder `none write ordering` aktiv sind (siehe nächster Punkt).
- `wo` (*write order*) – die aktuell eingesetzte *write ordering method*: b (*barrier*), f (*flush*), d (*drain*) oder n (*none*) (seit DRBD 8.2.7)
- `oos` (*out of sync*) – Speichermenge auf dem Storage, die zum Zeitpunkt der Abfrage Out of Sync ist (seit DRBD 8.2.6)

Weitere Details siehe *http://www.drbd.org/users-guide/ch-admin.html#s-proc-drbd*.

17.2.4 Manueller DRBD-Funktionstest (Master/Slave)

Um unser DRBD vor der Integration als Cluster-Ressource zu testen und vor allem um seine Funktionsweise zu verstehen, werden wir nun einen manuellen Failover simulieren. Nachdem wir sichergestellt haben, dass unser DRBD aktiv ist und sich im korrekten Zustand befindet *(Primary/Secondary* und *UpToDate/UpToDate)*, legen wir auf beiden Nodes den Mountpunkt */daten* an.

Auf dem Node, der den DRBD-Primary hält, erzeugen wir ein Dateisystem (z. B.: **`mkfs.btrfs /dev/drbd0`**) und hängen das DRBD-Leg anschließend unter */daten* ein. An dieser Stelle natürlich noch einmal der Hinweis, dass BTRFS derzeit (zum Zeitpunkt der Erstellung dieses Buches, Ausnahme wie bereits angesprochen: Ubuntu) noch nicht »ganz« offiziell für Produktivumgebungen empfohlen wird. Alternativ können wir als lokales FS natürlich auch z. B. ext4 oder XFS verwenden. Nun erzeugen wir eine Datei unter */daten*, z. B. *test.txt*, und befüllen sie optional mit etwas Inhalt. Dann simulieren wir den Failover-Fall, den später in unserem Cluster der DRBD-OCF-RA erkennt und entsprechend darauf reagiert. Die Schritte sind dabei im Einzelnen: Zuerst hängen wir den Mountpunkt */daten* auf dem Primary wieder aus, dann kommt der entscheidende Part: die Demotion (Herabstufung) des Primary zum Secondary – auf dem Primary:

```
#> drbdadm secondary r0
```

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z