Linux Clustering

Dr. Andreas Müller

Abstract. Für den Einsatz von Linux in kritischen Bereichen eines Unternehmens muss neben anderen Eigenschaften eine Garantie für höhere Verfügbarkeit abgegeben werden können. Dabei geht es nicht nur darum, Hardwareausfälle und Softwaredefekte überbrücken zu können, sondern auch darum, dass ohne Lücken für die Verfügbarkeit einzelne Teile eines Systems abgeschaltet werden können für Hardwareumbauten, Patches, Software-Upgrades und -Erweiterungen. Hochverfügbarkeitslösungen lassen solche Operationen für den Benutzer weitgehend transparent werden.

INHALT

Dieses Paper ist die Basis für einen Vortrag, der an der Linux-Conference am 27. Juni 2000 in Zürich gehalten wurde.

1.1. Ziele. Beim Aufbau eines Rechnerverbundes können grundsätzlich verschiedene Ziele angestrebt werden:

Rechenleistung: Die von den verschiedenen Prozessoren in einem Rechnerverbund aufgebrachte Leistung soll zur gemeinschaftlichen Lösung eines Problems zusammengefasst werden. Das einzelne Problem ist derart umfangreich, dass die Lösung auf einem einzelnen Rechner zu lange dauern würde.

Lastverteilung: Es ist eine grosse Zahl von kleineren Problemen zu lösen, deren Gesamtheit ein einzelnes System überfordern würde. Ein Rechnerverbund stellt sich den Klienten gegenüber als ein einziges System dar, in welchem Teil des Verbundes die Anfrage des Klienten behandelt, wird ist unwesentlich.

Verfügbarkeit: Eigentlich ist ein einzelnes System durchaus in der Lage, die gesamte Last aufzunehmen, doch soll durch ein bereitstehendes und automatisch aktiviertes System die Verfügbarkeit erhöht werden.

Selbstverständlich sind auch Mischformen denkbar: Erhöhte Verfügbarkeit kann ein Seiteneffekt eines Lastverteilungscluster sein, muss aber nicht.

1.2. Rechenleistung. Physiker und Kryptologen können jederzeit eine ganze Reihe von Problemen aufzählen, die sich mit einem einzelnen Rechner nicht innert nützlicher Frist[1] lösen lassen. Über die Art des eingesetzten Rechnerverbundes entscheidet in erster Linie das Problem selbst. Dabei sind folgende Kriterien von Bedeutung

1. In welchem Mass ist das Problem parallelisierbar? Die Berechnung der Ulam-Folge mit Hilfe der Rekursion $x_n = u(x_{n-1})$, wobei

$$u\colon \mathbb{N} \to \mathbb{N} : x \mapsto \begin{cases} \frac{x}{2} & x \text{ gerade} \\ 3x+1 & x \text{ ungerade,} \end{cases}$$

ist überhaupt nicht parallelisierbar. Kein einziger Schritt kann ausgeführt werden, bevor nicht der vorhergende abgeschlossen ist.

Am anderen Ende des Spektrums finden wir brute force Angriffe auf kryptographische Verfahren. Es geht darum, den gesamte Schlüsselraum durchzuprobieren. Jeder einzelne der 18'446'744'073'709'551'616 (18 Trillionen) möglichen Schlüssel für RC5-64[2] kann für sich allein untersucht werden. Die Untersuchung

[1] Was eine "nützliche" Frist ist hängt sehr stark vom Autor des Problems und seinem Umfeld ab. Wenn ein Physiker auf seine Resultate 10 Jahre warten muss, dürfte er beim Vorliegen der Resultate wegen mangels an Pulikationen nicht mehr in der akademischen Welt tätig sein. Muss ein Militär länger auf die Entschlüsselung der Geheimnisse seiner Feinde warten, als diese zur Konzeption neuer Pläne brauchen, sind sie ihm nicht mehr nützlich. Wenn ein Verkäufer länger auf die Entschlüsselung der Emails der Konkurrenz warten muss, als er bis zur Abgabe der Offerte Zeit hat, ist Sie ihm ebenfalls nicht mehr von Nutzen.

[2] Diese Bezeichnung ist gemäss offizieller Nomenklatur nach Rivest falsch, eigentlich müsste es RC5-32/12/8 heissen, also eine RC5 Verschlüsselung mit 32bit-Blöcken, zwölf Runden und 8 Bytes Schlüssel (64bit).

eines Schlüssels gibt auch keinen Hinweis auf eine Lösung des Problems mit einem anderen Schlüssel. Dieses Problem ist vollständig parallelisierbar, hätte man einen Verbund von 18 Trillionen Rechnern zur Verfügung, könnte man das Problem innert der kurzen Zeit lösen, die zur Bearbeitung eines einzelnen Schlüssels aufgewendet werden muss.

Ganz so viele Rechner kann das `distributed.net`-Projekt[3] nicht aufbringen, mit rund einer viertel Million beteiligter Rechner wird trotzdem eine Rate von 118 GSchlüssel/s erreicht. Ausserdem nimmt die Leistung dieses Verbundes täglich um etwa 150 MSchlüssel/s zu. Das Projekt läuft seit ungefähr 1000 Tagen, und nochmals soviel Zeit wird nötig sein, um es zum Abschluss zu bringen.

Dazwischen stehen physikalische Problem, die sich meistens mindestens teilweise parallelisieren lassen. Wegen der Tatsache, dass sich kein Signal schneller als das Licht ausbreiten kann, darf ich bei der Berechnung der Erdatmosphäre den Einfluss einer Eruption auf der Sonne während 8 Minuten ignorieren, bis das Licht auf der Erde auftrifft. Bei einer Simluation des Wasserstandes der Limmat nach einem Bruch des Sihlseedammes kann ich das Sihltal und das Limmattal ebenfalls lange Zeit als mehr oder weniger unabhängige Teilprobleme ansehen. Und wenn dann endlich etwas geschieht, gibt es eine relativ enge Schnittstelle, über die das Sihlwasser die Limmat beeinflusst.

2. Die Topologie der Teilprobleme muss in die Topologie des Cluster abgebildet werden können. Möchte alle Verbindungen zwischen je zwei Knoten realisieren, müssen bei n Knoten $\frac{n(n-1)}{2}$ Verbindungen aufgebaut werden. Die maximale Distanz zwischen zwei Knoten ist dann zwar 1, ober die Zahl der Verbindungen wächst viel zu schnell.

   Wählt man stattdessen die einfachere Anordnung eines k-dimensionalen Würfels ist die maximale Distanz k, es sind aber auch nur $2k$ Verbindungen nötig, um 2^k Knoten miteinander zu verbinden.
3. Welche Datenrate entsteht zwischen den Teilproblemen nach Parallelisierung?

   Das kryptologische Problem mit vollständiger Parallelisierung ist hier optimal: jeder Rechner testet den Schlüssel, der seiner Serien-Nummer entspricht, dazu braucht er überhaupt nicht mit den anderen Rechnern des Verbundes zu kommunizieren. Allerdings stehen dem Bau dieses Verbundes doch noch einige Hindernisse entgegen: brauchen wir für jeden Rechner nur einen Kubikzentimeter, würden sich die Rechner über die Fläche der Schweiz verteilt über vierhundert Meter hoch stapeln. Ich erspare Ihnen die Rechnungen für eine Schlüssellänge von 128bit.

1.3. Lastverteilung. Bei Lastverteilungsverbünden geht es darum, eine Resource, die ein Engpass für die Anwendung ist, mehrfach auszulegen und dafür zu sorgen, dass jede Komponente in ähnlichem Mass belastet wird. Selbstverständlich sollte die Wahl der Komponente für die Anwendung möglichst transparent sein.

[3] `www.distributed.net`

Es ist eine Ironie, dass Caching Mechanismen, die ja zur Verbesserung der Leistung eingebaut wurden, dem Load-Balancing meistens entgegenlaufen. Einige Beispiele:

1.3.1. Internet Link. Die Wahl zwischen zwei verschiedenen Netzwerklinks geschieht normalerweise durch ein Routing Protokoll. Voraussetzung ist, dass die Links entweder mit gleichen Kosten zum Internet führen (Equal Cost Multipath Routing) oder für je etwa die Hälfte der Destinationen die vorteilhaftere Metrik haben.

Der Route Cache eines Routers steuert der Lastverteilung jedoch entgegen: wurde eine Route für eine Destination einmal gewählt, fliessen alle Pakete zu dieser Destination über den gleichen Link.

1.3.2. Internet Proxy. Internet Proxies arbeiten bis auf den TCP Zustand zustandslos. Die verschiedenen Anfragen, die eine Seite ergeben, könnten also problemlos auf verschiedene Proxies verteilt werden. Doch nach welchen Kriterien soll dies geschehen, und durch welche Instanz? Verteilt man verschiedene Proxy-IP-Adressen mit Round Robin DNS an die verschiedenen Clients, die alle nach dem gleichen Proxy-Namen fragen, kann die Last trotzdem sehr asymmetrisch verteilt sein. Ausserdem verhindert das Client-seitige Caching der Adresse, dass die Anfragen eines einzelnen Browsers auf mehrere Proxies verteilt werden können.

Mit Hilfe von Proxy Autokonfiguration wäre es denkbar, bei jedem Request einen Proxy mehr oder weniger zufällig auszuwählen. Oder man könnte Anfragen mit einer externen Load Balancing Lösung auf einem Router oder einer Firewall an verschiedene Proxies leiten, in diesem Fall besteht die Hauptschwierigkeit darin, dass der Load Balancer zuverlässige Information über die Last auf den Proxies erhält.

Bei der Beurteilung der Last sind Systeme, die sich zum Beispiel auf die Laufzeit von IP abstützen völlig ungeeignet. Diese messen nämlich die nur die Länge der Paket-Queue im Kernel, nicht die Länge der Disk-Queue, über die die Seiten in den Cache geschreiben werden. Letztere Grösse ist jedoch meist der Flaschenhals bei einem Proxy.

1.3.3. Web Server. Ein Webserver kann ganz verschieden Lastengpässe haben: bei einer sehr grossen Zahl von Verbindungen kann der Netzwerkcode durch die Grösse der TCP Tabellen überfordert werden. Bei komplexen Anwendungen kann die Dauer einer einzelnen Abfrage einen CPU-Engpass verursachen. Bei einer Datenbank-Anwendung können Disk-I/Os das System and die Grenzen bringen.

Web Server können solange sehr leicht zu einem Verbund zusammengeschaltet werden, als die einzelnen Anfragen voneinander unabhängig sind. Sobald auf der Server-Seite zwischen den einzelnen Anfragen ein Zustand gespeichert bleibt, muss die nächste Abfrage wieder auf dem gleichen Server eintreffen. Nur eine gemeinsam genutzte Datenbank oder ein gemeinsamer File Store können diese Einschränkung mildern.

Das DNS Caching hilft in diesem Fall: hat ein Client einmal den Namen des Webservers in eine IP-Adrese aufgelöst, wird er es nicht nochmals tun. Und wenn, dann wird der Cache seines DNS-Servers die Daten für ihn bereithalten. Allerdings wird die Lastverteilung dadurch wieder kompromittiert.

1.3.4. File Server (read only). Diskless Clients verwenden das `/usr`-Filesystem nur le-

send, es liegt also nahe, auf verschiedenen Servern identische Kopien davon zu exportieren. Bei der Suche nach einem Server kann ein Klient dynamisch jenen wählen, der als erster Antwort gibt, also im Moment wenigstens nicht überlastet ist. Allerdings kann er später nicht mehr wechseln, die File Handles sind nur für einen Rechner gültig, er müsste also `/usr` aushängen und wieder neu einhängen, wozu ein reboot erforderlich ist.

1.4. Verfügbarkeit. Zweifellos am einfachsten zu realisieren sind Cluster, die in erster Linie der Verbesserung der Verfügbarkeit dienen. Die simpelste Version eines solchen Systems verwendet zwei identische Systeme, eines eingschaltet, das andere kalt. Der Failover erfolgt durch Betätigung des Netzschalters bei beiden Systemen. Dieser Failover ist nicht sehr schnell, die Systeme lassen sich in der Praxis nur schwer identisch konfiguriert halten, und es besteht eine Tendenz, das kalte System als Ersatzteillager zu verwenden. Im Fehlerfall fehlt dann nach Murphy eine entscheidende Komponente.

Ein weiterer Mangel ist die Tatsache, dass die Verfügbarkeit abhängt von der Verfügbarkeit des "Umschaltpersonals". Damit dauert die Umschaltung zwar immer noch weniger lang als eine Reparatur, mit Umschaltzeiten im Bereich von Stunden (statt Sekunden) ist jedoch zu rechnen.

2. Anwendungszustand

Entscheidend für den Erfolg beim Einsatz einer Clusterlösung ist das Ausmass, in dem Anwendungszustand beim Ausfall eines Knotens auf einen anderen Knoten übertragen werden kann. In diesem Kapitel werden einige Dienste mit diesem Blickwinkel auf ihre Clustertauglichkeit hin untersucht.

2.1. Zustandslose Dienste. Am einfachsten haben es Dienste, die überhaupt über keinen Zustand verfügen:

Routing — Router arbeiten zustandslos. Hat ein Paket einen Router verlassen, gibt es ausser den inkrementierten Byte- und Paketzählern keine Erinnerung mehr an das Paket. Zustand wird erst durch Address Translation (NAT) oder durch zustandsbasierte Filter (stateful firewalls) wieder eingeführt.

DNS — eine DNS Anfrage wird vom Server beantwortet, danach bleiben keine Spuren mehr. Aufgeweicht wird dieses Prinzip durch den name services cache daemon (`nscd`).

NIS — Der einzige Zustand bei einem NIS-System besteht im einmal erfolgten Bind mit einem NIS-Server. Beim Ausfall des Servers ist ein Rebind erforderlich. Dies erfolgt unter Umständen nicht automatisch. Aus Anwendungssicht ist NIS zustandslos.

LDAP — Zustand steckt in der TCP-Verbindung (sofern nicht connection less LDAP verwendet wird) und im möglicherweise erfolgten `bind`. Das LDAP API unterstützt einen automatischen Rebind.

RPC — Viele RPC Dienste sind wie NIS zustandslos konzipiert worden. Bei Diensten, die auf UDP basieren, ist nicht einmal ein neuer Verbindungsaufbau notwendig.

Secure RPCs verwenden verschlüsselte Aufrufe, der gemeinsame Schlüssel ist ein weiteres Zustands-Element, welches extrem schlecht geteilt werden kann. Aus dem gleichen Grund sind VPNs dem Clustering oft nicht gut zugänglich.

2.2. HTTP. Am Beispiel HTTP kann man sehr schön verfolgen, wie ein zustandsloses Grundkonzept immer weiter mit Zustand angereichert wird. Am Anfang war der einzige Zustand in den URLs gespeichert. Die einzelnen Seiten eines Dokumentenbestandes bildeten die Zustände, die die Zustandsmaschine "Browser" annehmen konnte. Durch Anklicken eines Links erfolgte ein Übergang in einen anderen Zustand.

Schon in Zähler auf einer Seite veränderte diese Situation grundlegend. War vorher der Zustand hinter einem URL unabhängig von der Vorgeschichte, änderte er sich mit jedem Besuch der Seite.

Einen weiteren Schritt brachte das CGI. Der Name könnte nicht treffender sein, es schafft ein Gateway zu noch viel komplexeren Zustandsmaschinen als das Web selbst ist. Jedem Programmierer von Web-Anwendungen ist das Problem bekannt, wie er einen Benutzer wiedererkennen soll. Hidden Fields oder spezielle URLs waren so lange der einzige Mechanismus, den Anwendungszustand zwischen Browser und Server zu vermitteln, bis die Cookies eingeführt wurden. Damit konnte der Zustand automatisch, ohne Zutun des Programmieres, unabhängig vom Weg des Benutzers durch die Anwendung, und sogar unabhängig vom Server, durch den Browser mitgeführt und dem Server bekanntgegeben werden.

Der letzte Schritt in der Entwicklung sind browserseitige Zustandsmaschinen: JavaScripts, Java Applets oder ActiveX controls. Bis zu einem gewissen Mass gehören auch Flash-Files dazu. Baut ein JavaApplet eine IIOP Verbindung zu einem Corba-Server auf, ist eine Anwendung entstanden in der extrem viel Zustand in einer langdauernden Verbindung auf konversationelle Art und Weise aufgebaut wird.

2.3. Authentisierung. Authentisierung beruht darauf, dass durch eine Login-Prozedur ein zwischen Klient und Server gemeinsamer Zustand aufgebaut wird, der es erlaubt, alle vom Klienten verlangten Aktionen einem bestimmten Benutzer zuzuordnen und nach bestimmten Regeln zu autorisieren.

Authentisierung und zustandsloses Design scheinen also grundsätzlich sich widersprechende Anforderungen zu sein. Allerdings nur auf den ersten Blick, die Konsequenz ist, dass bei jeder Anfrage Daten (Credentials und Verifiers) mitgeliefert werden müssen, die dem Server die zweifelsfreie Identifikation des Klienten ermöglichen. Bei HTTP ist dies zum Beispiel die Basic Authentication: bei jeder Anfrage wird in einem eigenen HTTP-Header Benutzername und Passwort mitgeschickt. Das Passwort muss jedesmal das gleiche sein, es ist auch kein Challenge Response Verfahren möglich.

Die Sicherheit eines Einmalpasswortes ist in HTTP nur dann realisierbar, wenn nach einem Authentisierungsvorgang ein Datum generiert wird, welches als Cookie oder URL-Komponente in allen nachfolgenden Schritten also "Zutrittsausweis" verwendet werden kann. Dieses Ticket muss ausserdem vor Diebstahl durch Unberechtigte geschützt werden, eine Verschlüsselung der Verbindung mit HTTPS ist also angezeigt.

Eine andere Möglichkeit sind kryptographische Verfahren. So wird zum Beispiel bei der Client-Authentisierung mit X.509-Zeritifikaten der Sitzungszustand in Form eines Sitzungsschlüssels auf beiden Seiten gespeichert. Dieser Cache kann normalerweise nicht mehrere Rechner umfassen, so dass nach einem Failover in jedem Fall eine Neuauthentisierung notwendig sein wird.

Ein sehr schönes Bespiel für zustandslose Authentisierung und Autorisierung ist Kerberos [2]. Nach einem einmaligen Austausch mit dem Kerberos Server ist der Klient im Besitz eines Ticket Granting Ticket, mit dem er für jeden Dienst, den er benutzen will, von einem Ticket Granting Server ein Ticket für den Service bekommen kann. Dank der Verschlüsselung der Tickets ist sichergestellt, dass jeder Request mit genau der gleichen Stärke authentisiert und autorisiert ist. Trotzdem können die einzelnen Requests vollständig unabhängig voneinander sein.

2.4. Netzwerkfilesysteme. Gemeinsame Nutzung von Files auf einem Server ist eine der verbreitetsten Client-Server Anwendungen überhaupt. Folgende Informationen kommen als Zustand vor:

Authentisierung: Bei einem zustandslosen Design ist es notwendig, die Identität bei jeder Anfrage neu festzustellen. Performante Lösungen sind nur mit kryptographischer Authentisierung oder durch grosses Vertrauen realisierbar.

Autorisierung: Sie entscheidet, ob ein bestimmter Benutzer auf einen Teilbereich des Filesystems zugreifen darf. Ein besonders interessantes Beispiel ist ein Unix-Prozess, der ein File mit der einen Userid öffnet, seine Userid wechselt und das File dann weiterbearbeitet. Eine zustandslose Implementation kann sich nicht an die "öffnende" Userid erinnern und muss gemäss der gegenwärtigen Userid autorisieren[4]. Dies widerspricht der üblichen Unix Filesystem Semantik.

File Offset: Soll der Server sich nicht an den gegenwärtigen Offset erinnern müssen, muss der Klient mit jeder Anfrage den File Offset mitliefern. Damit wird zuverlässiges "Anhängen" an ein File verunmöglicht.

File Locks: Wünscht ein Klient exklusive Schreibrechte auf ein File, ist dies nur durch serverseitigen Zustand realisierbar. Der Server kann Schreibanforderungen anderer Klienten nur dann ablehnen, wenn er sich an das Lock "erinnert".

Record Locks: Record Locks verlangen noch feinkörnigere Zustände auf der Serverseite.

2.4.1. Samba. Als Fileserver für PC Systeme hat Samba insbesondere mit dem Problem der plötzlich sterbenden Klienten fertig zu werden. Dabei muss es nicht immer ein Blue Screen sein, der Benutzer kann ja sein Gerät auch einfach auschalten.

Da man einem PC in keiner Weise trauen kann, genügt es nicht, wie etwa bei NFS, der in der Anfrage angegebenen Benutzeridentität einfach zu glauben. Vielmehr muss eine Authentisierung stattfinden, und deren Resultat in Form von Zustand gespeichert werden.

[4] Aus diesem Grund funktionierten zum Beispiel Applixware und plp bei Home-Verzeichnissen auf einem NFS-Server nicht zusammen.

Bei SMB geschieht das durch ein TCP-Verbindung, zu deren Beginn eine Authentisierungssequenz durchlaufen werden muss. Alle auf der gleichen Verbindung nachfolgend gestellten Anfragen werden der gleichen Beutzeridentität zugeordnet. Dieses Modell ist offensichtlich sehr schlecht als File Service für einen Multiuser-Klienten geeignet: jeder Klient muss seine eigenen Verbindungen haben[5].

Damit SMB trotzdem robust wird, muss der Klient transparent für den Benutzer eine neue Verbinungs aufbauen und eine Neuauthentisierung vornehmen. Und selbstverständlich müssen die unter Windows exzessiv verwendeten Locks wieder hergestellt werden.

2.4.2. NFS. Das Network File System NFS [1] ging ursprünglich auch von einem zustandslosen Design aus. Wichtigste Motivation war, dass der Klient einen Servercrash ohne Datenverlust überdauern können soll. Dies ist nicht möglich ohne eine Änderung des Filesystemmodelles, das dem Klienten präsentiert wird.

Bei direkt anschlossenen Platten werden alle Schreibanforderungen über den buffer pool des Betriebssystems geleitet, eine separater Prozess oder Kernel Thread sorgt dafür, dass die Daten regelmässig auf den Disk geschafft werden. Stürzt das System ab, sind alle Daten verloren, die zu diesem Zeitpunkt noch nicht auf die Platte geschrieben waren. Für NFS ist dies nicht akzeptabel. Da der Klient überhaupt nichts vom Absturz des Servers merken soll, müssen alle Schreiboperationen synchron durchgeführt werden, mit entsprechenden Folgen für die Performance.

Eine weitere Konsequenz ist die Unmöglichkeit, an ein bestehendes File zuverlässig etwas anzuhängen. Damit der Server zustandslos arbeiten kann, muss jede Schreibanforderung sowohl den Offset innerhalb des Files wie auch die Länge enthalten. Anhängen ist also nur möglich, indem zunächst die aktuelle Länge ermittelt wird, und ab dieser Position geschrieben wird. Die Race Condition ist offensichtlich.

File Locking ist eine Voraussetzung für Konsistenz von filebasierten Datenbeständen in einem Netzwerkumfeld. Ohne eine beträchtliche Menge Zustand ist es jedoch unmöglich, File Locking in NFS einzubauen. Insbesondere sind folgende Fälle zu berücksichtigen:

1. Der Server muss den Tod eines Klienten oder allgemein den Verlust des klientenseitigen Zustandes feststellen können, damit er die von diesem gehaltenen Locks freigeben kann.
2. Der Klient muss bei einem Servercrash oder allgemein bei einem Verlust des serverseitigen Zustandes die von ihm benötigten Locks wiederherstellen.

Die Schwierigkeiten zeichnen sich bereits ab: während die meisten Unix recht früh in einer NFS-Umgebung zuammenarbeiten konnten, dauerte es sehr lange, bis auch Filelocking zufriedenstellend funktionierte, und oft genug funktioniert es auch heute noch nicht.

3. Freie Failover-Projekte

[5]Ein Ausdruck dieser Schwierigkeit ist die Unmöglichkeit unter Windows als zwei verschiedene Benutzer auf zwei verschiedene Shares auf dem gleichen Server zuzugreifen. Allerdings handelt es sich dabei nicht um eine Schwäche des SMB Protokolls, sondern vielmehr der Implementation. Gemäss X/Open Spezifikation können innerhalb einer SMB Session durchaus verschiedene Uids verwendet werden.

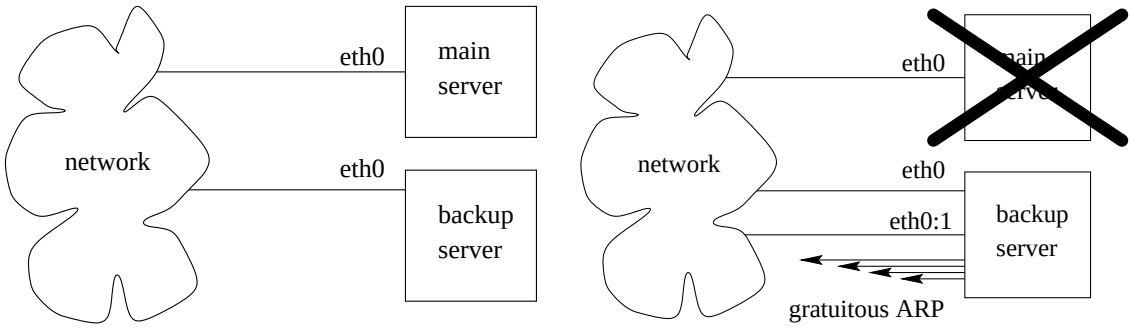

Abbildung 1. Fake Normalbetrieb (links) und Backupbetrieb (rechts).

3.1. Allgemeine Technik. Jede IP Failover-Lösung spielt mit der Hardware-Adresse des Netzwerk-Interface. Sofern nicht die Ethernet-Adresse auf das Backup-System übertragen werden kann (was viele Ethernet-Karten gar nicht unterstützen), muss mit Hilfe von ARP Paketen eine neue Zuordnung von Hardware-Adressen zu IP-Adressen signalisiert werden. Dabei werden folgende Regeln ausgenützt:

1. Die Zuordnung zwischen Ethernet-Adressen und IP-Adressen muss durch das ARP durchgeführt werden. Einmal gefundene Zuordnungen müssen in einem Cache gespeichert werden.
2. Sobald auf dem Netzwerksegment ein ARP-Paket sichtbar wird, welches eine neue Zuordnung für eine Hardware-Adresse mitteilt, muss diese neue Zuordnung verwendet werden, selbst wenn das Paket nicht an die eigene Ethernet-Adresse gerichtet ist.
3. Sobald ein Host eine ARP Anfrage nach seiner eigenen Adresse an die Hardware-Adresse `ff:ff:ff:ff:ff:ff` sendet (gratuitous ARP [3]), muss die darin angezeigte Zuordnung übernommen werden.

Leider ist die Implementation von ARP oft ziemlich verschieden von diesen Zielen:

1. Gewöhnliche ARP Replies werden normalerweise ignoriert, bis der Eintrag im eigenen Cache abgelaufen ist.
2. Eine Firewall-1 Firewall ignoriert gratuitous ARPs, andere Produkte verhalten sich ähnlich.
3. Bei vielen Switches werden die ARP Replies nicht an alle Stationen zugestellt, sie können also ihren ARP Cache gar nicht aufdatieren.

3.2. Fake. Fake[6] verwendet einen Strom von gratuitous ARP Paketen, um einem lamen Server keine Chance zu lassen, in die ARP-Tabelle der Rechner auf dem lokalen Segment zu kommen. Der Automatisierungsgrad der Lösung ist nicht sehr hoch, das Projekt scheint auch in den Folgeprojekten Heartbeat und Linux Virtual Server aufzugehen.

[6] `http://www.au.vergenet.net/linux/fake`

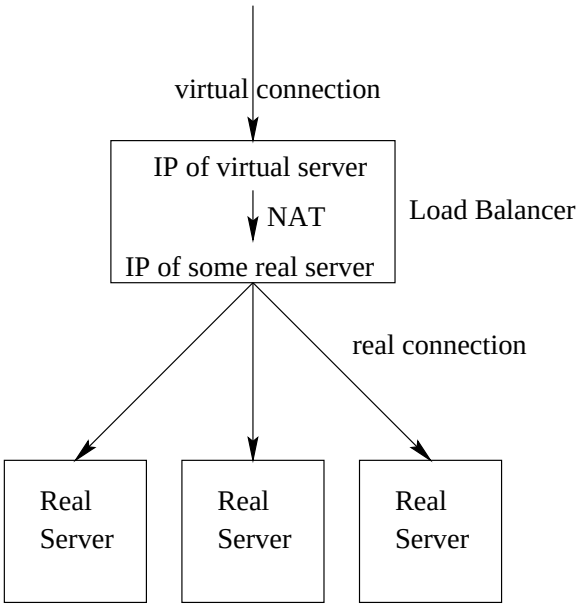

Abbildung 2. Linux Virtual Server.

3.3. failoverd. failoverd[7] versucht einen kontrollierteren Übergang zwischen verschiedenen Zuständen herbeizuführen. Statt einen lamen Server einfach mit einem Strom von gratuitous ARPs zu übertönen, wird ein vorhersehbarer Paketstrom zwischen zwei Hosts erzeugt. failoverd überwacht diesen Paketstrom und übernimmt eine IP-Adresse bei dessen Zusammenbruch (mit der Technik gemäss Abschnitt 3.1.).

3.4. Heart. Dieses Projekt[8] stellt eigentlich nur die Basistechnologie für ein weitergehendes Clusterprojekt zusammen. Hauptziel ist eine zuverlässige, portable Heartbeat-Architektur. Das Projekt wurde im September 1999 aufgelöst und die verbleibenden Teile wurden in das Linux-HA Projekt integriert.

3.5. Linux Virtual Server. Das Linux Virtual Server Projekt[9] strebt danach, mit Hilfe von Clustering einen grossen, skalierbaren virtuellen Server aus vielen einzelnen Knoten zusammenzubauen. Ein Load Sharing Gateway (Abbildung 2) verteilt mit Hilfe von Network Address Translation die Last auf mehrere reale Server. Dabei sind verschiedene Algorithmen für die Auswahl eines realen Servers verfügbar: round robin, gewichtet, nach am weitesten zurückliegender Verbindung, oder so, dass ein Client immer den gleichen Server erhält (damit Probleme mit serverseitigem Zustand minimiert werden können). Es geht nicht in erster Linie um Fehlertoleranz, denn das Konzept hat im load balancer einen single point of failure. Das Load balancing ist möglich für alle Protokolle, deren Adresstranslation in den Griff zu bekommen ist.

[7] `http://www.ps-ax.com/failoverd`
[8] `http://www.lemuria.org/Heart`
[9] `http://www.LinuxVirtualServer.org`

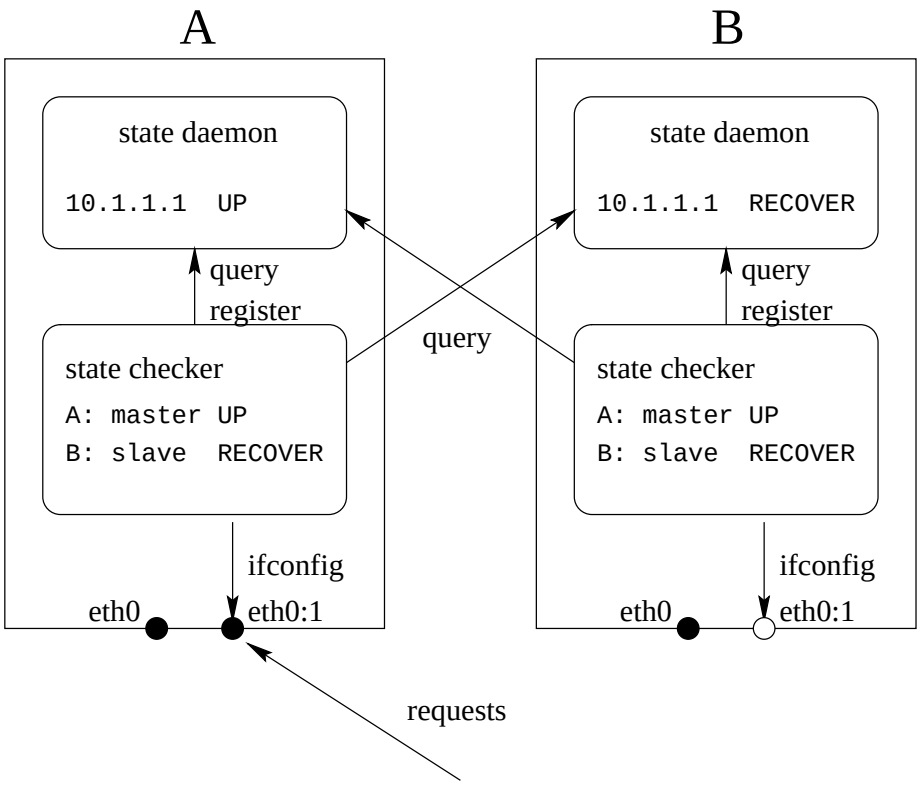

Abbildung 3. Failover Normalbetrieb.

Ähnliche Prinzipien wurden auch von Cisco, IBM und anderen in Switches oder Routers realisiert.

3.6. Das Linux-HA Projekt. Das Linux-HA Projekt hofft, bis 2001 **die** Hochverfügbarkeitslösung für Linux zu sein. Es basiert auf einem HOWTO von Harald Milz, welches die Konzepte des IBM HA Systems für AIX einer breiteren Linux Leserschaft zugänglich macht. Beim Durchlesen erhält man leider den Eindruck, Linux könne alles bereits, was dort drin steht. Dabei handelt es sich nur um Konzepte, Software dazu gibt es praktisch noch nicht.

Linux-HA verwendet Heartbeat, ein auf UDP aufbauendes Heartbeat Protokoll. Bei IANA wurde die Portnummer 684 für Heartbeat registriert. Im Moment sind Konfigurationen mit zwei Knoten unterstützt, welche in einer einfach Master/Slave-Relation stehen.

Die IP-Takeover Technik stammt aus dem Fake-Projekt. Andere Netzteilnehmer werden mit gratuitous ARPs dazu verleitet, die Zuordnung von IP-Adressen zu Ethernet-Adressen zu ändern.

3.7. `failover. failover`[10] ist mein eigener kleiner Beitrag zum Problem des automatisierten Failover. Ich erlaube mir, dieses etwas ausführlicher zu beschreiben, es ist auch

[10] `http://failover.othello.ch`

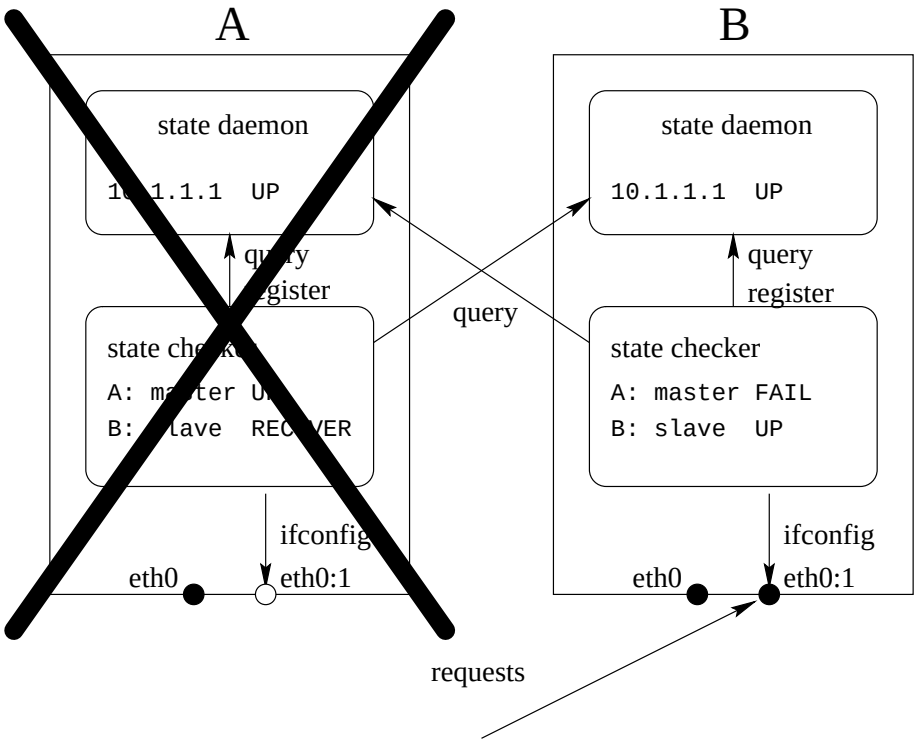

Abbildung 4. Failover Backupbetrieb.

jenes Projekt, das ich am besten verstehe.

3.7.1. Architektur. Failover verwendet einen Zustandsdämon, der auf jedem beteiligten Knoten läuft. Er enthält den Zustand jedes Dienstes, der auf dem Knoten angeboten wird. Es gibt drei Zustände:

`UP` Der Dienst ist aktiv

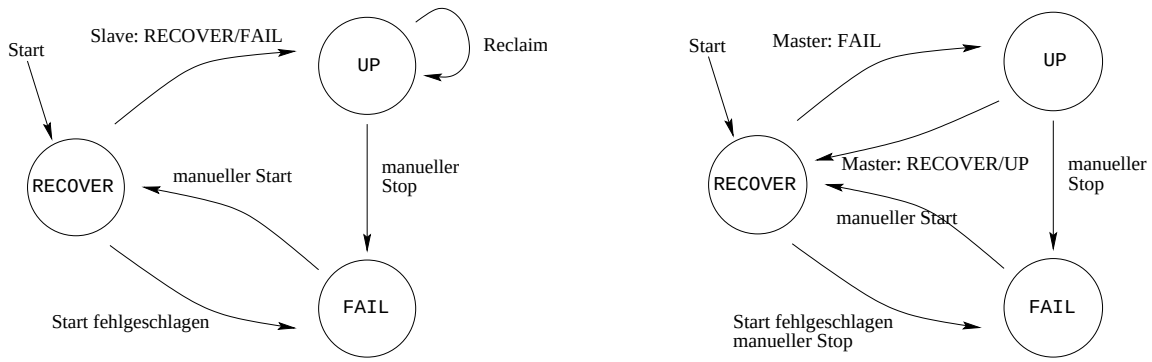

Abbildung 5. Failover Zustandsdiagramm Master (links) und Slave (rechts).

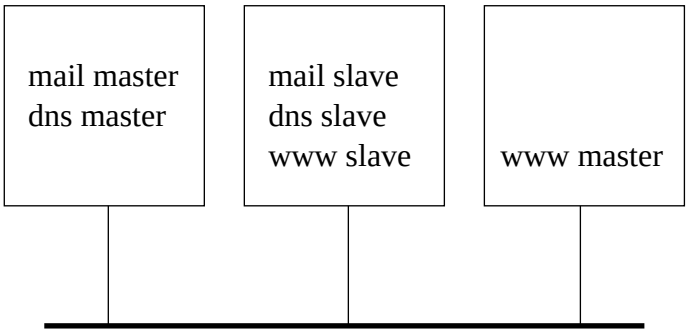

Abbildung 6. Failover Dreieckskonfiguration für DNS, Mail und Webserver.

`RECOVER` Der Dienst ist im Moment nicht aktiv, könnte aber jederzeit gestartet werden, alle Voraussetzungen dazu sind gegeben. In diesem Zustand ist ein Backup-Service im Normalfall. Beim Starten wird jeder Dienst zunächst in diesen Zustand gebracht, bevor entschieden wird, ob er aktiviert werden soll.

`FAIL` Der Dienst ist inaktiv, kann aber auch nicht gestartet werden. Dieser Knoten ist nicht in der Lage, im Failover-Fall die Funktion zu übernehmen.

Die eigentliche Arbeit des Startens und Stoppens von Diensten geschieht durch einen eigenen Prozess, der Klient des Zustandsdämons ist. In regelmässigen Abständen (meistens 5 oder 10 Sekunden, konfigurierbar) fragt er den Zustand aller Dienste ab, auf die er

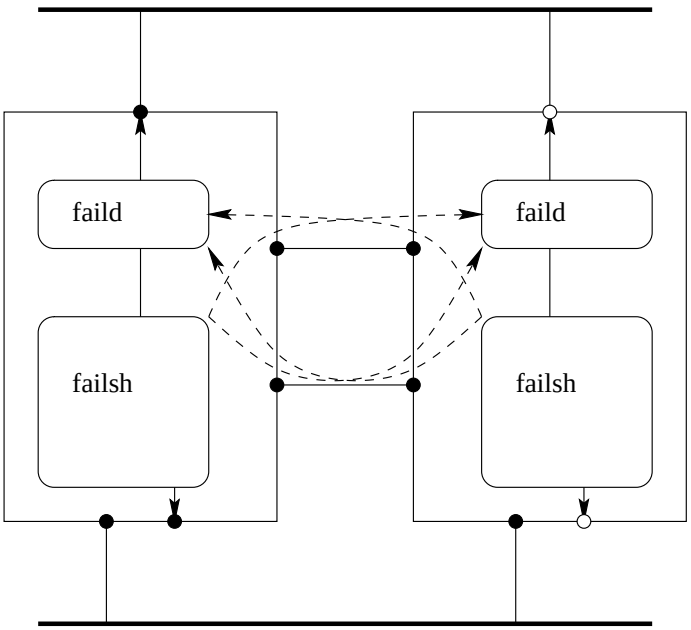

Abbildung 7. Failover für Firewall, doppelter Heartbeat.

programmiert ist. Der Klient kann anderen Knoten gegenüber zwei verschiedene Rollen einnehmen:

Master: Wenn dieser Klient den Dienst aktiviert hat, hat der andere Knoten zu schweigen.

Slave: Dieser Klient darf den Dienst erst dann aktivieren, wenn der andere sich davon zurückgezogen hat.

Der Klient kennt zwei zusätzliche Zustände:

`INDOUBT` Beim letzten Versuch hat der Zustandsdämon keine Antwort gegeben. Es besteht der Verdacht, dass der Knoten nicht mehr verfügbar ist.

`RECLAIM` Es wurde festgestellt, dass ein Knoten aktiv (`UP`) war, abwohl die oben auseindergesetzten Regeln dies eigentlich nicht gestattet hätten. Split-Brain Situation.

Der erste Zustand kann auch im normalen Betrieb eintreten, zum Beispiel bei einer Überlast des Masters. Darum entscheidet der Klient in der Slave-Rolle erst nach drei fehlgeschlagenen Verbindungs-Versuchen (also dreimaligem `INDOUBT`), dass der Master tot ist, und er den Dienst übernehmen muss.

Der zweite Zustand tritt dann ein, wenn die Netzwerkverbindung zwischen den beiden Rechnern unterbrochen worden ist. In dieser sogenannten Split-Brain Situation kommt jeder Klient zum Schluss, dass der Partner tot ist, er also den Dienst übernehmen muss. Sobald die Netzwerkverbindung wiederhergestellt ist, zum Beispiel indem der versehentlich ausgeschaltete Switch wieder eingeschaltet wird[11], stellt der Master fest, dass ein Slave aktiv ist, der nicht sein sollte, und der Slave stellt fest, dass er aktiv ist, obwohl sein Master aktiv ist. Der Slave wird sofort vom Dienst zurücktreten (in den Zustand `RECOVER`), und der Master wird sobald der Slave zurückgetreten ist, mit Hilfe einer speziellen Prozedur den Dienst wieder an sich reissen.

3.7.2. Implementation. Die Kommunikation zwischen Klient und Zustandsdämon erfolgt über normale Sun-RPCs. Diese wurden gewählt, da viele Fragen des Session-Management damit bereits gelöst schienen. Im Laufe der Zeit mussten jedoch viele Funktionen ausserhalb der RPC-Bibliotheken aufgebaut werden, da sie nicht vorgesehen oder fehlerhaft waren. Zum Beispiel ist es mit der RPC-Bibliothek allein nicht möglich, eine RPC-Verbindung aufbauen zu lassen, und bei deren Nichtzustandekommen innert fünf Sekunden einen Fehler zu generieren.

Der Klient, die `failsh`, ist eine Tcl-Anwendung, das Konfigurationsfile ist ein Tcl-Script, in welchem Prozeduren für das Starten und Stoppen von Diensten programmiert werden können. Ausserdem sind dem Tcl-Interpreter neue Befehle hinzugefügt worden, die Cluster-Dienst und -Knoten sowie deren Rollen (Master oder Slave) zu definieren gestatten.

Neben der `failsh` gibt es weitere Klienten, mit denen der Zustand abgefragt (`failstat`), verändert (`failc`) und überwacht (`failmon`, Klient zu `failsh`) werden kann. Der Monitor-Klient kann die Zustandsinformation wahlweise als ASCII-File oder HTML ausgeben, und

[11] Dieses Szenario hat sich tatsächlich einmal bei einem Kunden so abgespielt.

er kann als interaktive Curses-Anwendung betrieben werden.

Für einen einfachen Cluster baut man zwei Rechner, wobei der eine den Dienst, zum Beispiel einen Web-Server als Master anbietet, der andere als Slave. Der Web-Server selbst läuft auf einem virtuellen Interface, welches beliebig gestartet oder gestoppt werden kann, ohne die Netzwerk-Verbindung des Rechners selbst zu beeinflussen. Über die feste Adresse spricht der Klient mit dem Zustandsdämon und entscheidet, ob die virtuelle Adresse aktiv (`ifconfig up`) oder inaktiv (`ifconfig down`) sein soll.

Die Zuverlässigkeit kann gesteigert werden, indem zwischen Master und Slave ein zweites Netzwerk angelegt wird. Jeder Rechner hat jetzt zwei IP-Adressen, über die der Zustandsdämon abgefragt werden kann. Auf dem Master konfiguriert man beide Adressen als Slaves, auf dem Slave beide als Master. Für den Slave sieht es also so aus, als gäbe es zwei Rechner, die den Dienst anbieten, er wird erst dann übernehmen, wenn beide Masters tot sind. Der Master sieht ebenfalls zwei Slaves, von denen er nicht weiss, dass sie der gleiche Knoten sind, braucht er auch nicht. Sobald aber einer der Slaves aktiv wird, leitet er die Reclaim-Prozedur ein.

3.7.3. Probleme. Failover wird im Moment produktiv auf Solaris und Linux eingesetzt. Die Linux-Version ist dabei recht unproblematisch. Mit der Solaris-Version zeigten sich jedoch ein paar Probleme, die einen weitergehenden Umbau des Systems erfordern:

1. Es gibt Fälle, in denen die Last auf dem System so gross ist, dass der Zustandsdämon zu lange keine Antwort gibt. Der Klient geht dann davon aus, dass der Dienst nicht mehr verfügbar ist, und leitet die Failover-Sequenz ein. Diese Situation ist bisher erst in einem Fall eingetreten, wo das Loadaverage der Maschine bei etwa 40 lag! Das Problem kann gemildert werden, indem der Zustandsdämon mit hoher Priorität gefahren wird, wenn die Maschine jedoch zu pagen beginnt, hilft auch dies nicht mehr. Die Konsequenz ist, dass der Dienst auf zwei Systemen angeboten wird, was of zum Nichtfunktionieren führt.
2. Die Funktion `svctcp_create` der `/usr/lib/libnsl.so` funktioniert nicht. Sun empfiehlt stattdessen die gleichnamige Funktion aus `/usr/ucblib/librpcsoc.so` zu verwenden. Diese wird per Default jedoch nicht installiert.
3. Führt man einen Remote Procedure call auf einem Socket aus, den der Server inzwischen geschlossen hat, schlägt der Call natürlich fehl. Mit der Bibliothek `/usr/ucblib/librpcsoc.so` quittiert Solaris diese Situation mit einem Segmentation Fault. Sun empfiehlt, stattdessen `/usr/lib/libnsl.so` zu verwenden.
4. Mit Linux ist jedoch auch nicht alles eitel Freude: Die Implementation von Threads in Linux ist leider weit davon entfernt, Posix-kompatibel zu sein[12]. Der `clone` sy-

[12] Die Linux-Gemeinde hat leider eine Tendenz, in gewissen Dingen etwas oberflächlich zu sein. Die Marketing-Abteilung eines kommerziellen Anbieters mag vielleicht noch entschuldbar sein, wenn sie vollmundig von Posix-Compliance spricht, obwohl nur gerade die Header den Anforderungen entsprechen. Bei nicht vorhandener Funktionalität sollte die Linux-Gemeinde etwas ehrlicher sein. Was nützen mir Posix-Threads, die nicht so funktinieren wie Posix-Threads? Oder was nützt mir Large File Support, wenn die Anwendung beim Schreiben von Byte 2^{31} mit einem Fehler `ENOSPC` bestraft wird?

stem call erzeugt zwar einen eigenen Thread der mit dem Aufrufer den Adressraum teilt. Asynchrone Signale, zum Beispiel `SIGCHLD` bei Beendigung werden jedoch nicht an irgend einen Thread abgeliefert, der das Signal nicht maskiert hat. Nur der Thread, der den Kindprozess gestartet hat, kann auch das `SIGCHLD` bei seiner Beendigung erhalten. Die verbreitete Technik des Signal Handler Thread kann daher für Linux Threads nicht funktionieren.

Die in Arbeit befindliche neue Version von `failover` verwendet einen neuen Scheduler, der vollständig ohne Threads auskommt, er umgeht damit das Linux Threads Problem. Ebenfalls vorgesehen ist ein Umbau der Kommunikation, dieses Teilprojekt ist jedoch noch nicht so weit gediehen.

3.7.4. Kompliziertere Konfigurationen. Da Master und Slave nicht Attribute eines Rechners sind, und nicht einmal von einer Instanz von `failover` auf einem Rechner, sondern Rollen eines Dienstes auf einem Rechner, sind auch kompliziertere Konfigurationen ohne Änderung des Konzeptes realisierbar.

Backup für zwei verschiedene Systeme. Ein System kann Backup für zwei andere sein. Beispielsweise kann man in einer DMZ einen Web-Server und einen DNS-/Mail-Server haben. Alle diese Dienste werden auf virtuellen Interfaces angeboten. Ein dritter Rechner kann Backup für alle Dienste sein. Fällt der Webserver aus, wird seine Adresse auf dem Backup-Rechner aktiv. Fällt der Mailserver aus, geschieht das gleiche mit dessen Adresse.

Der DNS-Server auf dem Backup-System kann ein Secondary zum Primary auf dem Hauptsystem sein, für einen eventuellen weiteren offsite secondary ist dies nicht feststellbar.

Einfaches Load Sharing zwischen zwei Proxies. Zwei Proxies *A* und *B* werden mit zwei zusätzlichen virtuellen IP-Adressen *x* und *y* ausgestattet, die im DNS mit dem gleichen Namen versehen werden. Für den Dienst *x* ist *A* Master, *B* Slave, für den Dienst *y* umgekehrt. Dank Round-Robin-DNS werden die Adressen *x* und *y* ungefähr gleichmässig in der Browser-Population gestreut, beide Proxies werden also etwa gleich ausgelastet sein. Sobald einer der Proxies versagt, übernimmt der andere dessen Dienst. Das Resultat ist ein einfacher Load Sharing Cluster mit hoher Verfügbarkeit.

Da die Dienste auch manuell jederzeit auf den anderen Rechner gebracht werden können, können Wartungsarbeiten ohne totale Unterbrüche durchgeführt werden. Natürlich gehen beim Failover aktive Verbindungen verloren. Leider merken das die Browser nicht immer, und verzichten auf eine Warnung. Sie geben so dem Benutzer den Eindruck, ein grösseres File sei vollständig übertragen worden.

Redundante Firewall. Bei einer Firewall stehen zum vornherein genügend Interfaces für den Heartbeat zur Verfügung, ausserdem arbeitet sie oft völlig zustandslos, was den Failover fast trivial macht. Einziger Unterschied: es müssen beim Übernehmen des Dienstes die Adressen aller Interfaces übernommen werden.

Neu ist hingegen das Problem dass der Heartbeat durch die Firewall-Regeln einerseits zugelassen werden muss, dass aber andererseits die Firewall auch vor gefälschten Heartbeat-Paketen geschützt werden muss. Immerhin könnte damit ein Denial of Service Angriff

gefahren werden.

4. Was braucht es in Zukunft?

Es ist eine Illusion zu glauben, man könne jede beliebige Anwendung einfach so nehmen, auf einem Cluster installieren, und schon hat man ein hochverfügbares System. Damit die Anwender aus den bestehenden und in Entwicklung befindlichen Clusterfähigkeiten von Linux auch einen praktischen Nutzen ziehen können, müssen in den weitaus meisten Fällen die Anwendungen umgebaut werden. Sie müssen tolerant werden gegen Verlust eines Teils des Anwendungszustandes. In vielen Fällen wird dies nur über ein Redesign möglich sein.

Wie im Kapitel 2. ausgeführt, muss die Anwendung tolerant gegenüber dem Verlust von klienten- oder serverseitigem Zustand sein. Liegt der Zustand vor allem auf dem Server, muss man zum Beispiel durch Replikation dafür sorgen, dass er gar nicht verlorgen gehen kann. Liegt der Zustand vor allem auf dem Klienten, muss man sicherstellen, dass der Klient nach dem Failover mit seinem Zustand auf dem Server neu aufsetzen kann.

4.1. Überwachung von Anwendungen. Für einen sinnvollen Failover braucht es mehr als nur eine Antwort von einem Heartbeat-Protokoll, welches vollständig unabhängig ist von der eigentlichen Anwendung. Es müssen Mechanismen vorhanden sein, mit denen zweifelsfrei festgestellt werden kann, ob ein Dienst noch vollumfänglich erbracht werden kann.

Ein Beispiel: der iPlanet Proxy Server hat die unangenehme Eigenschaft, dass ihm gelegentlich Prozesse wegsterben. In dieser Situation funktioniert er nicht mehr richtig: die Klienten können zwar noch verbinden, und dürfen ihre Anforderungen absetzen, eine Antwort erhalten sie jedoch nie. Ein anderes Beispiel betrifft eine Oracle-Anwendung, die ein externes System über Oracle Pipes anfragt. Manchmal “verstopfen” die Pipes, und die Queries, welche die externe Resource verwenden, bleiben stecken. Für den Oracle Server ist jedoch alles in Ordnung, die Verbindungen brechen nicht ab, und warum soll eine Anfrage bei einer externen Resource nicht lange dauern dürfen. Wenn man den externen Server direkt fragt, ist kein Problem erkennbar, es handelt sich wirklich nur um ein Kommunikationsproblem zwischen Datenbankserver und Anwendung.

Voll clustertaugliche Anwendungen sollten also immer mit einem Überwachungsinterface gebaut werden, welches in der Lage ist, fehlerhafte Systemzustände zu erkennen. Oder mindestens sollte es in solchen Fällen keine Antwort mehr liefern, was für eine Failover-Lösung auch eine Antwort sein kann.

4.2. Single Master und Multi Master. Sich über mehrere Rechner erstreckende Datenspeicher müssen das Problem konkurrierender Updates lösen. Durch gleichzeitige Updates auf zwei Knoten darf die Konsistenz der Datenbank nicht verletzt werden. Die ACID-Forderungen sollen auch für das verteilte System gelten. Vergleichsweise einfach ist dieses Prinzip einzuhalten, wenn nur auf einem einzigen System Updates möglich sind, die auf

[13] `http://www.openldap.org`

die anderen Systeme weitergegeben werden. Auf diese Weise funktionieren `slapd` und `slurpd`[13], man spricht von single master replication. Bei multi master replication sind Updates auf einer grösseren Zahl von Knoten gestattet.

4.3. Clusterfähige Filesysteme. Um einen Fileserver-Cluster zu bauen, braucht man folgendes:

1. Redundante ausgelegt Disk Hardware, zum Beispiel an beide System anschliessbare Disks.
2. Spiegelung auf das redundante Diskystem, oder über das Netzwerk, zum Beispiel mit dem network block device.
3. Ein Filesstem auf dem redundaten Disksystem, welches sehr schnell von einem anderen Rechner eingehängt (gemountet) werden kann.

Nach eine Crash des primären Servers im Cluster muss der Backup-Knoten sicherstellen, dass der Master nicht mehr auf den Disk zugreift, muss den Disk übernehmen, einen file system check durchführen, das Filesystem einhängen (mounten) und exportieren. Bei einem `ext2fs` dauert der file system check für ein typisches Home-Filesystem mit einer sehr grossen Anzahl von Files viel zu lange, die meisten Benutzer dürften im Failover-Fall einen Datenverlust zu beklagen haben.

Mit einem logstrukturierten File-System wird es besser, entscheidend ist hier, dass das Log pro Filesystem angelegt wird (nicht wie in der Defaultinstallation beim `jfs` von AIX), damit es vom Backup-Rechner pro Filesystem übernommen werden kann. Ideal scheint mir der Ansatz, den Sun im File System Logging in Solaris 7 verwendet, dort ist das Log Bestandteil des Filesystems.

Auf einem sehr aktiven Fileserver würde man sich jedoch auch wünschen, dass mehrere Rechner gleichzeitig ein Filesystem aufdatieren können. Mit ein paar grössen multiported Storage Arrays, je mit grossem Cache, auf denen gespiegelt Kopien der Filesystem abgelegt werden, könnten performante ausfallsichere Fileserver gebaut werden. Das Global File System Projekt arbeitet in diese Richtung.

Ausserdem müssen die Zugriffsprotokolle ebenfalls “clustertolerant” sein, oder der Server muss darauf rücksicht nehmen können. Bei NFS bedeutet es, dass der file handle eines Files unabhängig ist von der Maschine, auf der das Filesystem gerade eingehängt. Bei SMB muss der voll qualifizierte Resourcenname ebenfalls knotenunabhängig sein. Dies bedeutet, dass sich die einzelnen Knoten des Clusters genau gleich in den Netbios Namespace einfügen wenn sie aktiv sind, so dass für den Klienten kein Unterschied bemerkbar wird.

4.4. Clusterfähige Datenbanken. Wesentlich schwieriger ist der Aufbau von Clustern von Datenbanken. Solange single master replication genügt, ist ein Setup ausreichend, bei dem der Master alle Updates in ein Log schreibt, welches der Slave möglichst zeitverzugslos ausführt. Doch auch schon in diesem einfachen Szenario stecken einige Tücken:

1. Wie wird das Log auf den Backup-Knoten geschafft? Der Mechanismus sollte einerseits möglichst zeitverzugslos sein, andererseits sollte er auch möglichst sparsam mit den Resourcen umgehen.

2. Was geschieht beim Wiederanlauf des Hauptknotens? Nach einem Failover sind möglicherweise beide Knoten in einem nicht wieder vereinbaren Zustand: einige Updates, die im Master noch durchgeführt wurden, sind nicht mehr im Slave angekommen[14]. Trotzdem sind im Slave neue Updates durchgeführt worden, die möglicherweise den verlorenen Updates im Master widersprechen. Ein klares Beispiel wären Sequenznummern für Transaktionen, die zum Beispiel von einer Webanwendung vergeben werden. Der Backupknoten wird diejenigen Sequenznummern verwenden, die der Master schon vergeben hat, aber noch nicht auf den Slave replizieren konnte. Diese Anwendung wird man umbauen müssen, wenn man die verschiedenen Sitzungen nicht durcheinander bringen will. Bei einem virtuellen Blumenladen mögen so ja vielleicht noch lustige Situationen entstehen, ein Kranz zur Hochzeit dürfte den Humor der beteiligten vielleicht doch strapazieren. Bei Anwendungen mit sensitiven Daten, zum Beispiel Lebensversicherungs- oder Krankenversicherungsofferten würde klar der Datenschutz missachtet.

Es scheint machbar, single master replication zum Beispiel mit MySQL zu realisieren. Für multi master replication scheint neben kommerziellen Datenbankanbietern dürfte das Projekt Mariposa[15] von Mike Stonebraker am weitesten fortgeschritten sein.

4.5. Zustandslose Client-Server Anwendungen. Nach dem Vorbild von NFS und HTTP sollten wir danach streben, unsere Anwendungen so zustandslos wie möglich zu bauen. Transaktionsorientierte Anwendungen sind einem Cluster-Umfeld weit eher zugänglich, als Anwendungen, in denen Zustand über eine lange Folge interaktiver Dialogschritte zwischen Client und Server aufgebaut wird.

Eine HTTP-Anwedung, die den gesamten Sitzungszustand in einem Cookie im Client ablegt, also einen zustandslosen Server verwendet, kann problemlos in einem Loadsharing-Cluster betrieben werden, bei dem jeder Request auf einem anderen Knoten landen kann.

Eine Web-Anwendung, die den Zustand in einer Datenbank auf einem separaten Server speichert, kann eine Cluster von HTTP-Servern für Loadsharing und Failover verwenden. Der Datenbankserver selbst wird jedoch ein single point of failure sein.

Während Zustand eine Frage ist, die vor allem das Design einer Anwendung betrifft, hat die Forderung nach Clusterfähigkeit von Anwendungen auch unmittelbare Konsquenzen für die Implementation:

1. Nicht mehr antwortende Verbindungen müssen viel eher abgebrochen werden, als das Betriebssystem es von sich aus tun würde. Ein TCP-Timeout von 10 Minuten ist viel zu lange.
2. Abgebrochen Verbindungen müssen für die Anwendung transparent wiederhergestellt werden.

[14]Bei der Replikation von IBM DB2 beispielsweise wird das Log zur Schonung von Netzresourcen in festen Zeitintervallen auf den Backup-Knoten gebracht, bei einem Crash des Haupt-Knoten bleibt als ein Stück Log, welches nie auf den Backup gebracht wurde.

[15]`http://mariposa.cs.berkeley.edu`

3. Wir brauchen Frameworks, die Zustand ohne Zutun des Anwendungsentwicklers wiederherstellen können.

Daher der Aufruf an alle Designer und Entwickler: Think Cluster!

LITERATUR

1. Brent Callaghan, *NFS Illustrated*, Addison Wesley, 2000.
2. Bruce Schneier, *Applied Cryptography, Second Edition*, John, Wiley & Sons, Inc., 1996.
3. W. Richard Stevens, *TCP/IP Illustrated, Volume 1, The Protocols*, Addison Wesley, 1998.
4. W. Richard Stevens, *TCP/IP Illustrated, Volume 3*, Addison Wesley, 1998.