9 Datenaustausch im Netz

9.1 Datenaustausch via NFS

von Dr. Lex Wennmacher

Mit Hilfe von NFS ist es möglich, ein gesamtes Dateisystem oder einen Unterverzeichnisbaum daraus (auf dem sogenannten NFS-Server) über das Netzwerk einem anderen Rechner (dem sogenannten NFS-Client) zur Verfügung zu stellen. Der Client kann dabei (anders als zum Beispiel beim Datenaustausch über ftp) mit den Standard-Datei-Systemaufrufen *open*(2), *read*(2) und *write*(2) auf die Dateien zugreifen, also ganz so, als ob die Daten lokal auf dem Rechner vorhanden wären.

NFS wurde wie viele andere heute benutzten Netzwerk-Standards im Jahr 1985 von der Firma Sun Microsystems entwickelt. Sun entwarf dabei nicht nur eine NFS-Implementation, sondern publizierte auch den dazugehörigen Standard. Aus einer Reihe von Gründen – unter anderem lizenzrechtlicher Natur – konnte Berkeley die Original-Sun-Implementation nicht übernehmen. Man entschloß sich stattdessen, NFS zu reimplementieren. Dabei hielt man sich nicht in allen Aspekten an den von Sun vorgegebenen NFS-Standard, sondern verbesserte ihn in einigen Punkten (zum Beispiel bei den sogenannten »Leases«). Die daraus resultierende Implementation erhielt daher den Namen *nqnfs* (not quite NFS, nicht ganz NFS). Das in NetBSD verwendete nqnfs ist völlig abwärtskompatibel zu NFS Version 2, viele der Verbesserungen des nqnfs-Protokolls sind in NFS Version 3 eingeflossen.

Wie sieht nun das Grundprinzip der Funktionsweise von NFS aus? Wichtig zu wissen ist zunächst, daß es sich bei NFS um ein zustandsloses Protokoll handelt. Das bedeutet, daß sich der NFS-Server nicht den Zustand eines NFS-Clients merkt. Alle NFS-Operationen werden als Fern-Prozeduraufrufe (Remote Procedure Calls, RPC) abgewikkelt. Der Server prüft jeweils nur, ob der RPC-Aufruf von einer berechtigten Maschine stammt und führt ihn aus.

Fern-Prozeduraufrufe sind vergleichbar mit lokalen Prozeduraufrufen. Zunächst werden die Argumente für den RPC gemarshalled, das heißt, zusammengebündelt in einen Puffer gelegt, in ein bestimmtes Transportformat umgewandelt und über das Netzwerk einem entfernten Rechner übermittelt. Dieser führt nun den umgekehrten Prozeß durch (Zurückwandeln aus dem Netzwerk-Format in das Binärformat entsprechend der eigenen Maschinenarchitektur und Zerlegen in die einzelnen Argumentbausteine). Dann führt der entfernte Rechner mit diesen Argumenten einen Prozeduraufruf durch. Dessen Ergebnisse werden auf gleiche Weise wieder zurückübertragen. Bis zum Eintreffen des Ergebnisses wartet der aufrufende Rechner (beziehungsweise arbeitet an einem anderen Prozeß). Die zurückgelieferten Daten werden wieder in das eigene Maschinenformat umgewandelt und als Ergebnisse an den Aufrufer zurückgegeben. Programmierer finden die notwendigen Details zu RPC in der Manual-Seite *rpc*(3).

Damit Rechner mit unterschiedlichen Maschinenformaten (Little Endian und Big Endian) über das Netzwerk miteinander Daten austauschen können, hat Sun Microsystems ebenfalls ein spezielles Datenformat hierfür definiert: XDR (eXternal Data Representa-

tion). Grob gesagt legt XDR für jeden Maschinentyp exakt fest, in welcher Reihenfolge jedes einzelne Bit über die Leitung geht. NetBSD stellt, wie jedes andere gute Unix auch, als Teil seiner RPC-Implementation XDR-Routinen in der *libc* zur Verfügung. Wer mehr über XDR wissen will, sei auf die Manual-Seite *xdr*(3) hingewiesen.

Um größtmögliche Flexibilität zu gewährleisten, werden für RPCs keine festen Ports oder Adressen vergeben, sondern es wird eine Art Übersetzer verwendet. Wenn ein Client also einen RPC durchführen will, kontaktiert er zuvor diesen Übersetzer (auf dem entfernten Rechner) mit dem Namen *rpcbind*, um sich die Adresse geben zu lassen, an die der nachfolgende RPC dann geschickt wird. Damit *rpcbind* diese Übersetzungen alle durchführen kann, muß *rpcbind* auf jeder Maschine, die RPC-Anfragen annehmen soll, als erster RPC-Dienst gestartet werden. Alle später gestarteten RPC-Dienste registrieren ihre Adressen und die Dienste, die sie bedienen, beim *rpcbind*, so daß dieser dann die Anfragen der Netzwerk-Clients beantworten kann. Man erkennt übrigens, daß wegen dieses Registrierungsmechanismus auch alle RPC-Dienste neu gestartet werden müssen, falls einmal der Neustart des *rpcbind* erforderlich werden sollte.

RPC als solches kann sowohl über TCP als auch über UDP betrieben werden. Die NetBSD-Implementation des NFS-Servers unterstützt – anders als viele andere Unix-Systeme (beispielsweise Linux) – sowohl TCP/IP als auch UDP/IP als zugrunde liegendes Transportprotokoll. Die Unterstützung von TCP ist ein enormer Vorteil, wenn NFS-Mounts über Netze vorgenommen werden sollen, bei dem man in höherem Maße mit Paketverlusten rechnen muß.

9.1.1 Vorbereitende Arbeiten

Zunächst muß natürlich auf beiden Rechnern, dem Server und dem Client, das Netzwerk korrekt aufgesetzt sein und die Netzwerk-Verbindung zwischen beiden Rechnern gewährleistet sein.

Sollten die Rechner beispielsweise durch eine Firewall getrennt sein, ist dafür Sorge zu tragen, daß Port 111 (für den RPC-Service) und Port 2049 (für den *nfsd*) geöffnet sind.

Je nach Anwendung kann auch die Synchronisation der Uhren von Server und Client erforderlich werden. Zu der Frage, wann die Uhrensynchronisation wirklich erforderlich wird, findet man in der Literatur oft nur unzureichende und zum Teil auch falsche Informationen. Hier soll deswegen dieser Themenkomplex einmal genauer besprochen werden.

Bei einem Schreibzugriff über NFS ruft letztlich der NFS-Server den Systemaufruf *write*(2) (oder einen ähnlichen Aufruf) auf. Als Teil dieses Aufrufs werden auch die entsprechenden Zeitstempel der Datei modifiziert, und zwar mit von der Uhr des NFS-Servers abgeleiteten Zeiten. Die Systemuhr des Client spielt für die Zeitstempel also keine Rolle. Selbst wenn viele Clients mit falsch gehenden Uhren quasi-parallel auf ein NFS-Dateisystem zugreifen, passiert nichts schlimmes, da die Zugriffe durch die Systemuhr des NFS-Servers serialisiert werden.

Ungemach droht lediglich dann, wenn ein NFS-Client den Zeitstempel einer NFS-Datei mit seiner eigenen Uhr vergleicht. Dann kann es sogar passieren, daß diese zur Verwirrung des Benutzers und des Systems aus der Zukunft zu stammen scheinen.

Ein Paradebeispiel für ein Unix-Kommando, bei dem Probleme mit Dateien aus der Zukunft zu erwarten sind, ist *make*. *make* wird benutzt, um eine Zieldatei nach bestimmten Regeln automatisch aus Quelldateien und möglichen Zwischendateien zu erstellen. Um seine Arbeit mit möglichst geringem Ressourcenverbrauch durchführen zu können, liest *make* ein Makefile, um herauszufinden, welche Datei von einer anderen abhängt und welche Schritte daraufhin überhaupt noch durchzuführen sind. Hierbei werden die Zeitstempel von Quell- und Zieldatei verglichen. Ist die Quelldatei neuer als die Zieldatei, muß die Zieldatei nach den im Makefile spezifizierten Regeln neu erstellt werden.

Dieses Vergleichen von Dateizeitstempeln ist über NFS völlig unkritisch, auch in dem Fall, wo eine oder beide Dateien in der Zukunft zu liegen scheinen. Der Haken beim Verwenden von *make* in einem NFS-Dateisystem liegt aber darin, daß es zu dateilosen Zwischenschritten kommen kann. In diesen Fällen muß *make* den Zeitstempel für den Zwischenschritt von der Systemuhr ableiten, womit ein mögliches Scheitern bereits vorprogrammiert ist.

Grundsätzlich ist also zur Vermeidung von Inkonsistenzen bei *make* eine Uhrensynchronisation zwischen NFS-Server und NFS-Client erstrebenswert. Ist das nicht möglich, da beispielsweise Server und Client in unterschiedlichen administrativen Domänen liegen, kann man versuchen, die Make-Datei so umzuschreiben, daß dateilose Zwischenschritte vermieden werden. Dies erreicht man dadurch, daß man alle Zwischenschritte grundsätzlich von (Dummy-) Dateien abhängig macht, deren Zeitstempel dann mit einem Aufruf von *touch* aktualisiert werden.

Zur Synchronisation von Uhren verweise ich auf Kapitel 7.6.3.

Die Rechte bei Zugriff über NFS werden genau wie beim Zugriff auf ein lokales FFS aus UID und GID bestimmt. NFS nimmt einfach an, daß die UIDs und GIDs auf Server und Client dieselben Benutzer beschreiben. Wer als Administrator also sicherstellen will, daß die Konsistenz der Zugriffsrechte auch über NFS gewahrt bleiben, muß sicherstellen, daß die Benutzer auf Server- und Client-Seite die gleichen UIDs und GIDs besitzen. Das kann natürlich manuell beim Einrichten der Accounts erfolgen oder besser durch Verwenden von netzwerkweiten Passwortsystemen, wie beispielsweise Yellow Pages (NIS). Die Konfiguration von Yellow Pages wird in Kapitel 7.6.5 beschrieben.

Ist es aus administrativen Gründen (zum Beispiel weil NFS-Server und -Client sich in unterschiedlichen administrativen Domänen befinden) nicht möglich, benutzereinheitliche UIDs und GIDs zu verwenden, kann man zwei verschiedene Tricks anwenden. Zum einen läßt NFS beim Mounten die Angabe eines General-Accounts zu, auf den alle UIDs abgebildet werden, die Details hierzu werden in 9.1.2 beschrieben. Zum anderen kann man eine UID-/GID-Abbildung durch eine Zwischenschicht des Dateisystems erwirken. Dieses Verfahren ist flexibler als die erste Methode, funktioniert zur Zeit allerdings auch nur bei bis zu sechzehn verschiedenen Benutzern und muß noch als »experimentell« eingestuft werden. Die Details werden in 9.1.4 beschrieben.

Eine Besonderheit stellt der NFS-Zugriff durch den Administrator *root* dar. Obwohl »Sicherheit« kein Designziel von NFS war, werden standardmäßig alle Zugriffe durch *root* auf der Client-Seite vom NFS-Server in Zugriffe durch den Benutzer *nobody* umgewandelt. Damit soll verhindert werden, daß die root-Privilegien auf dem NFS-Client unkontrolliert auf den NFS-Server übertragen werden.

9.1.2 NFS-Server einrichten

Das Einrichten eines NFS-Servers stellt den größten Teil der Arbeit dar, ist aber nicht wirklich kompliziert. Zuerst muß sichergestellt werden, daß der Kernel auf der NFS-Server-Maschine Unterstützung für die NFS-Server-Funktion enthält. Dies kann man zum Beispiel mit dem folgenden Befehl prüfen:

```
% nm -o /netbsd | grep nfssvc_addsock | wc -l
```

Wird eine 1 angezeigt, ist das Symbol *nfssvc_addsock* im Kernel »/netbsd« enthalten. Somit ist die NFS-Serverfunktion vorhanden. Falls eine 0 zurückgegeben wird, muß zuerst ein geeigneter Kernel gebaut werden. Das erreicht man dadurch, daß man in der Kernelkonfigurationsdatei der NFS-Servermaschine die Zeile

```
options         NFSSERVER          # Network File System server
```

aktiviert (indem man die Zeilen zur Konfigurationsdatei hinzufügt oder die vorhandenen führenden Kommentarzeichen entfernt) und den Kernel neu kompiliert. Die Details der Prozedur sind in Kapitel 4 beschrieben. Nach dem Kopieren des neuen Kernels nach »/netbsd« und einem Neustart ist die Maschine dann als Server einsatzbereit.

Ein NFS-Server braucht selbst kein NFS-Dateissystem im Kernel, also kann folgende Zeile der Kernelkonfiguration ruhig auskommentiert sein, es sei denn, man will den NFS-Server gleichzeitig auch als NFS-Client (dann bei einem dritten NFS-Server) einsetzen):

```
file-system     NFS                # Network File System client
```

Das Bedienen von NFS-Requests (im wesentlichen das Weiterleiten von NFS-Requests, die via RPC über das Netzwerk empfangen werden, an den Kernel) wird – bis auf das später zu beschreibende Locking – von zwei Dämonen übernommen: *mountd* und *nfsd*. *mountd* prozessiert lediglich Mount-Requests, wohingegen der *nfsd* für das eigentliche Bearbeiten der NFS-Requests zuständig ist.

Der Daemon *mountd* erfüllt im wesentlichen zwei Funktionen: er hört auf *mount*-Anforderungen von NFS-Clients und bedient diese; außerdem importiert er die Erlaubnisspezifikationen aus einer Konfigurationsdatei (»/etc/exports«) in den Kernel. Wichtig ist zu wissen, daß *mountd* den Inhalt von der im folgenden beschriebenen Datei »/etc/exports« in der Regel nur einmal, nämlich beim Start, liest und in den Kernel importiert. Ändert man den Inhalt dieser Datei, während der *mountd* bereits läuft, muß man den Daemon dazu bewegen, »/etc/exports« erneut zu lesen, indem man ihm ein *SIGHUP*-Signal schickt. Das erreicht man mit folgendem Kommando:

```
# kill -HUP `cat /var/run/mountd.pid`
```

In der Datei »/etc/exports« wird angegeben, welche lokalen Dateisysteme vom NFS-Server über das Netz zur Verfügung gestellt (exportiert) werden sollen. Außerdem können hier Optionen angegeben und der Kreis der Clients begrenzt werden.

Jede Zeile in »/etc/exports« (mit Ausnahme der Kommentarzeilen) spezifiziert die Bedingungen, unter denen ein Directory an einen oder mehrere andere Rechner exportiert wird. Im einfachsten Fall wird in einer Zeile nur ein Verzeichnis angegeben, was dann dazu führt, daß es an die ganze Welt exportiert wird. Wegen der fehlenden Sicherheitsmechanismen von NFS wird hiervon strikt abgeraten.

Die Exportbedingungen können nun durch Modifizierer eingeschränkt oder abgeändert werden. So wird beispielsweise mit der Angabe *-network 192.168.48/24* beziehungsweise *-network 192.168.48 -mask 255.255.255.0* der Zugriff auf ein Subnetz beschränkt.

Normalerweise ist nur das am Zeilenanfang angegebene Verzeichnis zum Export freigegeben. Die optionale Angabe von *-alldirs* erweitert die Freigabe auf alle Unterverzeichnisse.

Einen Schreibzugriff auf exportierte Dateisysteme verwehrt man durch die Angabe der Option *-ro* (read only).

Wie in Kapitel 9.1.1 beschrieben, werden NFS-Zugriffe durch *root* aus Sicherheitsgründen standardmäßig auf den unprivilegierten Benutzer *nobody* abgebildet. Durch die Angabe von *-maproot=user* läßt sich diese Abbildung ändern. Insbesondere läßt *-maproot=root* zu, daß *root* auf dem Client auf *root* auf dem Server abgebildet wird. Das ist zwar gefährlich, manchmal (insbesondere beim Bauen und Installieren von Software in einem NFS-Dateissystem) nützlich. *user* kann hier wahlweise mit dem Accountnamen oder über die UID angegeben werden.

Mit Hilfe der Option *-mapall=user* ist es möglich, NFS-Requests aller Benutzer (einschließlich *root*) auf einen Generalaccount abzubilden. Die allgemeinere Form *-mapall=user:group1:group2:...* erlaubt es außerdem, eine oder mehrere Gruppen anzugeben, als deren Mitglied *user* angesehen wird.

Hier nun ein paar Beispiele:

```
/usr -ro -network 131.104.48 -mask 255.255.255.0
```

Exportiert »/usr« (nur Lesezugriffe erlaubt) an das Subnetz 131.104.48.*. Mounten von »/usr/local« ist aber auf den Client wegen des fehlenden *-alldirs* nicht erlaubt.

```
/home -alldirs -maproot=root -network 131.104.48/24
```

Liberaler Export von »/home« an das Subnetz 131.104.48.*.

```
/projekt/src -ro -mapall=nobody
```

Die ganze Welt darf lesend mit den Rechten von *nobody* »/project/src« mounten.

Dies sind die wichtigsten Exportoptionen. Wer weitere eher selten verwendete Optionen benötigt (beispielsweise *netgroups*) sei auf die Manualseite *exports*(5) verwiesen.

Schließlich und endlich kommen wir zu dem Daemon, der im Betrieb die meiste Arbeit zu leisten hat: dem *nfsd*, der für das Abarbeiten aller NFS-Anforderungen zuständig ist. Seine Implementation ist interessant: nachdem *nfsd* die üblichen Arbeiten eines Unix-Daemons erledigt hat (Interpretieren der Kommandozeile und so weiter), führt er einen Systemaufruf (Aufruf von *nfssvc*(2)) durch, aus dem er (mit ganz wenigen Ausnahmen) nie zurückkehrt. Auf diese Weise war es den BSD-Implementatoren auch ohne das Vorhandensein von Kernel-Threads möglich, über den Umweg von Prozessen, die ihren Kontext bereitstellen, ein gut funktionierendes Multithreading zu erreichen. Dabei übernimmt die Routine *nfsrv_rcv()* im Kernel über einen Upcall die Aufgabe, die Last an die *nfsd* zu verteilen.

Üblicherweise werden vier bis sechs Instanzen des *nfsd* gestartet, meistens ist für normale Anwendungen der Standardwert von vier ausreichend. In Kapitel 9.1.6 wird erklärt, wie man erkennen kann, ob eine ausreichende Zahl von *nfsd*-Instanzen gestartet wurde.

Damit alle auf dem NFS-Server benötigten Daemons beim Booten automatisch gestartet werden, sollte man folgendes in die »/etc/rc.conf« eintragen:

```
rpcbind=YES
rpcbind_flags="-l"
nfs_server=YES
nfsd_flags="-6 -t -u -n 4"
```

Die Optionen von *nfsd_flags* bedeuten folgendes:

Höre auf IPv6 Anforderungen (*-6*) (falls IPv6 nicht vorhanden ist, wird einfach IPv4 benutzt); bediene NFS-Clients über TCP (*-t*); bediene NFS-Clients über UDP (*-u*); Starte vier *nfsd*-Instanzen (*-n 4*). Diese *nfsd_flags* sind übrigens die Voreinstellung von NetBSD 1.5.3 und 1.6, man findet sie in kompakterer Form in der »/etc/defaults/rc.conf« wieder als:

```
nfsd_flags="-6tun 4"
```

9.1.3 NFS-Client einrichten

Auch beim Einrichten eines NFS-Clients prüfen wir zuerst, ob der aktuelle Kernel die erforderliche NFS-Unterstützung (diesmal als Client) bereitstellt:

```
% nm -o /netbsd | grep nqnfs_clientd | wc -l
```

Wird eine 1 angezeigt, ist die NFS-Client-Funktionalität im Kernel vorhanden, andernfalls muß sie in der Kernel-Konfigurationsdatei hinzugefügt werden:

```
file-system       NFS               # Network File System client
```

Nach dem Übersetzen und dem Installieren des neuen Kernels steht dann nach dem ersten Reboot die erforderliche NFS-Client-Funktionalität zur Verfügung.

Es muß nun noch geprüft werden, ob der Mountpunkt (im folgenden Beispiel »/mntpkt«), auf den das entfernte Dateisystem montiert werden soll, vorhanden ist; andernfalls muß der Mountpunkt noch angelegt werden:

```
# ls -ald /mntpkt
ls: /mntpkt: No such file or directory
# mkdir /mntpkt
```

Nun kann das entfernte Dateisystem montiert werden:

```
# mount -t nfs fern1:/home/fritz /mntpkt
```

Ab nun steht der Dateibaum »/home/fritz« des entfernten Rechners »fern1« unterhalb des Mountpunkts »/mntpkt« zur Verfügung.

Beim obigen *mount*-Befehl kann die Option *-t nfs* auch weggelassen werden, weil *mount* so intelligent ist, daß es an der Syntax *host:directory* erkennt, daß ein NFS-Mount gemeint ist. Wenn dieser Dateibaum nicht mehr benötigt wird, kann er mit Hilfe von

```
# umount /mntpkt
```

wieder entfernt werden. Das funktioniert allerdings nur, wenn in dem noch eingebundenen Dateisystem keine Datei mehr offen ist. Es darf also keinen Prozeß mehr geben, der eine Datei unterhalb von »/mntpkt« geöffnet hat. Auch reicht es aus, daß ein Prozeß sein aktuelles Arbeitsverzeichnis unterhalb von »/mntpkt« hat, damit *umount* den Dienst versagt. Wenn Ihnen das passiert, lesen Sie weiter unten unter »Probleme finden und beseitigen« nach, was zu tun ist.

Der *mount*-Befehl ist ein sogenanntes Frontend zu einer Vielzahl spezialisierter Mount-Befehle. Im Fall von NFS-Mounts interpretiert *mount* die Kommandozeile, liest die Datei »/etc/fstab« und ruft letztendlich den Befehl *mount_nfs* auf. Im Prinzip könnte der Benutzer *mount_nfs* auch direkt aufrufen, allerdings überläßt man besser dem generischen *mount* das Interpretieren der Argumente (damit diese über alle Mount-Befehle hinweg einheitlich gehandhabt werden).

Zum NFS-*mount*-Kommando gibt es eine Reihe von Optionen, von denen insbesondere die Attribute *hard*, *soft* und *interruptible* von Interesse sind. Anders als bei lokalen Platten sind Störungen von NFS-gemounteten Dateisystemen leider häufig (wegen Ausfall des NFS-Servers oder einer zwischenliegenden Netzwerkkomponente). Je nachdem, für welches der erwähnten Attribute man sich entschieden hat, reagiert das Betriebssystem unterschiedlich auf NFS-Anforderungen an hängende NFS-Server.

Da gibt es zunächst einmal das »harte« Standardverhalten: Datei-Systemaufrufe blokkieren (und können nicht unterbrochen werden), falls der entfernte NFS-Server hängt. Viele Programme sind auf transiente Fehler nicht vorbereitet und würden unnötigerweise abbrechen, falls eine Dateioperation mit einem transienten Fehler zurückkehren würde. Dies macht sich natürlich insbesondere bei lange laufenden Programmen negativ bemerkbar. Das Standardverhalten sorgt also dafür, daß Programme bei temporären Netzwerkstörungen nicht abbrechen, sondern blockieren, bis die Normalität wieder hergestellt ist.

Das andere Extrem sind die sogenannten »Soft-Mounts«. Sie versuchen, eine bestimmte Anzahl von Wiederholungen der NFS-Anforderungen (standardmäßig 10000) auszuführen, bevor sie dann aber endgültig aufgeben. Der sogenannte Retrycount kann mit Hilfe der Option *-R* von *mount_nfs* eingestellt werden. Wird er zu niedrig gewählt, brechen NFS-Operationen zu früh (zum Beispiel bei überlastetem NFS-Server) ab. Bei zu hohen Retrycount-Werten kann dagegen ein Prozeß zu lange hängen. Mounts werden auf diese »weiche« Weise durchgeführt, indem man die Option *-s* bei *mount_nfs* (beziehungsweise *-o -s* bei *mount*) angibt.

Als von vielen Systemadministratoren bevorzugten Kompromiß bieten sich »interruptible« (abbrechbare) Mounts an. Ihr Verhalten ähnelt denen des »Hard«-Mounts mit dem Unterschied, daß bei unterbrechbaren Mounts geprüft wird, ob dem auf NFS-Operationen wartenden Prozeß ein Interrupt-Signal geschickt wurde, worauf hin dieser die NFS-Operation unterbricht. Unterbrechbare Mounts werden mit Hilfe der Option *-i* bei *mount_nfs* (beziehungsweise *-o -i* bei *mount*) spezifiziert.

Genauso wie lokale Dateisysteme in »/etc/fstab« eingetragen werden, können und sollten über NFS einzubindende Dateisysteme dort eingetragen sein. In der ersten Spalte sind Rechnername und zu mountendes Verzeichnis in der Form »Rechnername:Verzeichnis« anzugeben. In der zweiten Spalte wird der lokale Mount-Punkt eingetragen, in der dritten natürlich *nfs* als Dateisystem. Schließlich nimmt die vierte Spalte die Optionen auf, zum Beispiel *rw,-i* für unterbrechbare Schreib-/Lese-Mounts. Die fünfte und sechste Spalte sind für NFS-Mounts ohne Bedeutung.

Vielleicht ist Ihnen von anderen unix-ähnlichen Betriebssystemen her der Daemon *nfsiod* bekannt. Aufgabe dieses Daemons ist es, auf der Seite des NFS-Clients NFS-Operationen im Hauptspeicher zwischenzuspeichern. Dafür schaltet sich der *nfsiod* zwischen Systemaufrufebene und NFS-Ebene. So gehen beispielsweise alle *read*(2)-Aufrufe erst an den *nfsiod*, bevor sie über das Netz an den NFS-Server weitergeleitet werden. Der *nfsiod* prüft in einem solchen Fall, ob er die Leseanfrage aus seinem eigenen Cache befriedigen kann. Falls nein, formuliert *nfsiod* die Anfrage gegebenenfalls so um, daß sie eine hinreichend große Datenmenge umfaßt. Auf diese Weise wird das leistungssteigernde »Read ahead caching« erreicht. Die Leistungssteigerung ist dabei darauf zurückzuführen, daß für mehrere kurze Leseanforderungen nur ein (aufwendiger) RPC-Request ausgeführt werden muß, alle nachfolgenden Leseanforderungen können danach aus dem Cache bedient werden.

Genau wie bei den *nfsd*-Daemons werden mehrere *nfsiod* gestartet (üblicherweise vier bis sechs), um quasi-gleichzeitige Leseanforderungen (von verschiedenen Prozessen) besser bedienen zu können.

Seit Betriebssystemversion 1.5 gibt es in NetBSD allerdings keinen *nfsiod* mehr. Dessen Funktionalität wurde, nachdem Kernel-Threads eingeführt worden waren, vollständig in den NFS-Teil des Kernels übernommen. Angaben wie

```
nfsiod=YES
```

oder

```
nfsiod_flags=
```

sind in »/etc/rc.conf« nicht mehr erforderlich oder wünschenswert und sollten daher entfernt werden.

9.1.4 Selten benötigt

mount_umap

Möchte man Dateisysteme aus unterschiedlichen administrativen Domänen via NFS miteinander verbinden, steht man vor dem Problem, daß Benutzer, die Accounts in beiden Domänen besitzen, in der Regel unterschiedliche UIDs haben. Sie können also normalerweise nicht als Eigentümer auf ihre via NFS gemounteten Dateien zugreifen. Zur Lösung dieses Problems stellt NetBSD das Kommando *mount_umap* zur Verfügung.

Der Administrator gibt (über die Optionen *-u* beziehungsweise *-g*) *mount_umap* zwei Dateien an, in denen die Abbildungen der UIDs und der GIDS von der einen auf die andere Domäne enthalten sind. *mount_umap* montiert dann das angegebene Zielsystem unterhalb eines ebenfalls angegebenen Mountpunkts. Wichtig: Damit der Befehl *mount_umap* funktioniert, muß der aktuelle Kernel mit der Option

```
file-system    UMAPFS          # NULLFS + uid and gid remapping
```

übersetzt worden sein!

Die Benutzung von *mount_umap* sei hier anhand eines Beispiels erläutert. Nehmen wir an, auf dem lokalen NetBSD-System gäbe es einen Account *fritz* mit der UID 777 und der GID 15. Fritz hat auch einen Account *xy123* (mit der UID 123, GID 123) auf der Maschine *most.secret.com*, von der wir das Dateisystem »/home« lokal unter »/mnt« einbinden wollen. Zunächst legen wir als *root* die Dateien »uidfile«

```
1
777 123
```

und »gidfile«

```
1
15 123
```

an. Die erste Zeile gibt die Zahl der Einträge an, gefolgt von paarweisen Einträgen der Form *<lokale ID> <Original-ID>*. »uidfile« und »gidfile« müssen *root* gehören und dürfen nur von *root* beschreibbar sein.

Zunächst wird »/home« normal via NFS eingebunden (dazu muß dieses Verzeichnis natürlich auf *most.secret.com* exportiert werden):

```
# mount most.secret.com:/home /mnt2
```

Dann wird »/mnt2« erneut gemountet, diesmal mit *mount_umap*:

```
# mount_umap -u uidfile -g gidfile /mnt2 /mnt
```

Nun kann *fritz* unter »/mnt« auf seine Dateien als Eigentümer zugreifen.

lockd

NFS bietet standardmäßig die meisten Dateioperationen (*open*, *close*, *read*, *write*, *stat* und so weiter) für ein entferntes Dateisystem an, aber eben nicht alle. Stiefkind der Implementation ist das File Locking.

NetBSD bietet auf Systemaufrufebene zwei Methoden an, eine Datei gegen Zugriffe von anderen Prozessen zu sperren: die Systemaufrufe *flock*(2) und *fcntl*(2). Dabei bietet *flock*(2) sogenannte »Advisory Locks« an, das heißt, Prozesse müssen kooperieren, um einen geregelten Dateizugriff zu gewährleisten (beide müssen *flock*(2) aufrufen). Im Gegenteil dazu steht *fcntl*(2), mit dem man eine ganze Datei oder Teile von ihr gegen den Zugriff anderer Prozesse sperren kann. Die mit *fcntl*(2) erhaltenen Locks sind absolut, das heißt, jeder andere Prozeß, der versucht, auf die gesperrte Datei zuzugreifen, blockiert solange, bis der Lock wieder freigegeben wird. Es gibt noch einen weiteren wichtigen Unterschied zwischen den mit *flock*(2) und *fcntl*(2) erwirkten Locks, der deren Freigabe betrifft: Es kann sein, daß ein Lock auf eine Datei von mehreren Prozessen angefordert wird. Bei *flock*(2) wird der Lock erst dann wieder aufgehoben, wenn der letzte Lock wieder freigegeben wird. Anders bei *fcntl*(2): Hier reicht es aus, daß irgendein Prozeß den Lock freigibt, zum Beispiel dadurch, daß er den Dateideskriptor mit *close*(2) schließt, um den Lock aufzuheben. Dieses unsinnige Verhalten von *fcntl*(2) wird leider vom POSIX-Standard erzwungen. Beide Locking-Methoden (*flock*(2) und *fcntl*(2)) können aber völlig unabhängig voneinander eingesetzt werden und beeinflussen sich nicht gegenseitig.

Damit das im NFS-Protokoll nicht implementierte Locking dennoch eingesetzt werden kann, wurde diese Aufgabe einem separaten Daemon, dem *rpc.lockd* (oft auch nur als *lockd* bekannt) übertragen. NetBSD stellt ihn zur Verfügung. Die Sache hat nur einen

kleinen Haken: NetBSD stellt mit dem *rpc.lockd* nur die Implementation der Serverseite zur Verfügung, nicht jedoch die Client-Seite. Das heißt, es gibt unter NetBSD zur Zeit keine Möglichkeit, einen Lock über NFS zu erwirken. Ein NetBSD NFS-Server kann aber NFS-Locking-Anforderungen, die von Maschinen eines anderen Betriebssystem (zum Beispiel Solaris) aus gestellt werden, honorieren.

Wenn man den *rpc.lockd* einsetzt, sollte man ebenfalls den *rpc.statd* benutzen. Dieser Dämon stellt einen Status-Monitoring-Dienst im Verbund mit anderen *rpc.statd* auf anderen Maschinen zur Verfügung. Stürzt eine von *rpc.statd* überwachte Maschine ab und bootet anschließend, meldet sich deren *rpc.statd* beim lokalen *rpc.statd* und dieser kann dann die Programme benachrichtigen, die den Monitoring-Dienst vom *rpc.statd* angefordert haben. Da *rpc.lockd* von diesem Dienst Gebrauch macht, werden also *rpc.lockd* und *rpc.statd* in der Regel immer gemeinsam betrieben.

9.1.5 Probleme finden und beseitigen

Irgend etwas funktioniert mit NFS nicht. Dann gilt es zuerst herauszufinden, bei welcher der drei wesentlichen Komponenten NFS-Server, NFS-Client oder dem Netzwerk dazwischen das Problem liegt. Zur Diagnose stellt NetBSD (wie die meisten anderen Unix-Varianten auch) eine Reihe von Programmen zur Verfügung:

rpcinfo zeigt die im RPC-System registrierten RPC-Dienste an. In der Form

```
# rpcinfo <hostname>
```

werden die auf *hostname* laufenden RPC-Dienste ermittelt. Schlägt zum Beispiel das NFS-Mounten eines Verzeichnisses auf dem entfernten Rechner *flippy* fehl, kann man mittels

```
# rpcinfo flippy
```

überprüfen, ob die benötigten Dienste *portmapper*, *mountd* und *nfsd* auf *flippy* registriert sind.

Als nächstes ist es denkbar, daß alle NFS-Dienste laufen, aber das einzubindende Verzeichnis nicht exportiert ist. Hier hilft der Befehl *showmount* weiter.

```
# showmount -e flippy
```

zeigt an, welche Verzeichnisse *flippy* (an wen) exportiert. Beispiel:

```
# mount flippy:/usr /mnt
mount_nfs: can't access /usr: Permission denied
# showmount -e flippy
Exports list on flippy:
/home                              137.85.061.0
```

Aha, »/usr« ist gar nicht exportiert!

Wichtig ist, Exportlisten grundsätzlich mit *showmount* (und nicht durch visuelles Inspizieren von »/etc/exports«) zu überprüfen. Beispielsweise kann man nach einem Eintrag in »/etc/exports« vergessen haben, dem *mountd* dies mitzuteilen, oder die exports-Datei enthält einen Tippfehler. Benutzt man *showmount,* geht die Information den gleichen Weg wie beim Mounten (über RPC) und diese offensichtlichen Fehler werden sofort gefunden.

Auch das Unmounten kann fehlschlagen, was sogar ein häufiges Problem ist. Nehmen wir beispielsweise an, »flippy:/home« sei auf dem lokalen System als »/mnt« eingebunden und soll nun ausgebunden werden:

```
# umount /mnt
umount: /mnt: Device busy
```

Diese Fehlermeldung besagt, daß es auf dem lokalen System noch mindestens einen Prozeß gibt, der (eine) Datei(en) geöffnet hat. Damit der Prozeß seine Modifikationen zurückschreiben kann, erlaubt es das Betriebssystem dann nicht, das fremde Dateisystem zu entfernen. Wie kann man nun feststellen, welcher Prozeß den Unmount-Vorgang blockiert?

Im Paketsystem »pkgsrc« ist ein populäres Kommando, *lsof* (LiSt Open Files) enthalten (unter »pkgsrc/sysutils/lsof«), das auch von Administratoren unter Linux oder Solaris gerne in solchen oder ähnlichen Fällen benutzt wird. *lsof* kann natürlich hierzu benutzt werden, jedoch bietet die BSD-Unix-Familie bereits direkt als Bestandteil des Betriebssystems ein geeignetes Werkzeug an: *fstat*.

In diesem typischen Fall würde man als NetBSD-Administrator *fstat* folgendermaßen verwenden:

```
# fstat -f /mnt
USER     CMD          PID   FD MOUNT      INUM MODE         SZ|DV R/W
fritz    vi           420   wd /mnt       1984 drwxr-xr-x    1536 r
fritz    vi           420    3 /mnt       1999 -rw-r--r--     292 r
```

Damit ist der schuldige Prozeß identifiziert. Fritz hat noch einen *vi*-Prozeß laufen, der sowohl sein »Working Directory« unterhalb von »/mnt« besitzt, als auch eine Datei geöffnet hat (Deskriptor 3). Nun kann der Administrator Fritz bitten, *vi* zu beenden oder im Notfall den *vi*-Prozeß killen.

9.1.6 Leistungssteigerung von NFS

Meistens ist die Leistung von NFS mit den Standardeinstellungen adäquat. Jedoch kann beispielsweise beim Vorliegen bestimmter Netzwerk- oder Hardwaregegebenheiten mangelnde Leistung zum Problem werden. NetBSD bietet eine Vielzahl von Stellschräubchen, mit denen man das Feintuning von NFS vornehmen kann, um die Leistung zu steigern. Allerdings ist das Tuning eine außerordentlich komplexe Sache, so

daß im Rahmen dieses Buchs nur die groben Züge des Tunings besprochen werden können. Für Details sei auf Fachliteratur, zum Beispiel »Optimizing NFS Performance: Tuning and Troubleshooting NFS on HP-UX Systems« von David Olker (ISBN: 0130428167) verwiesen.

Die wichtigsten Tuningparameter sind:

- TCP oder UDP als Protokoll.
- Zahl der *nfsd* auf dem NFS-Server.
- Größe der Puffer.
- Synchrones/asynchrones Schreiben.

Vor der Therapie steht die Diagnose und das wichtigste Diagnosewerkzeug heißt *nfsstat*. Bei jeder NFS-Operation wird im Kernel ein entsprechender Zähler inkrementiert (sowohl auf dem Client als auch auf dem Server). *nfsstat* dient dazu, diese Zähler auszulesen. Es ist allerdings nicht ganz einfach, aus den von *nfsstat* gelieferten Zahlen eindeutige Empfehlungen für Tuning-Maßnahmen abzuleiten. Bei den folgenden (wichtigsten) Tuning-Maßnahmen kommen wir ohne *nfsstat* aus:

TCP oder UDP als Protokoll

Da UDP den weitaus geringeren Protokoll-Overhead besitzt, ist die NFS-Performance unter optimalen Bedingungen (keine Paketverluste, geringe Netzwerk-Latenz) mit UDP am besten. Sobald jedoch Paketverluste auftreten, kehrt sich dies ins Gegenteil um. Bei der Verwendung von UDP muß die Applikation die korrekte Übertragung der Netzwerkpakete sicherstellen. Dazu startet sie beim Versenden eines RPC-Requests einen Timer. Wenn dann ein Timeout auftritt, wird der gleiche RPC-Request erneut versendet (RPC Retransmit). Standardmäßig werden acht Kbyte große Puffer verwendet, was mit dem für den Headern benötigten Platz dazu führt, daß die UDP-Pakete auf sechs IP-Pakete heruntergebrochen werden müssen. Bei einem Retransmit müssen also wieder sechs neue IP-Pakete verschickt werden. Anders ist es bei der Verwendung von TCP, hier wird die Kommunikationssicherheit vom Protokoll garantiert (und bei Paketverlusten braucht in der Regel nur ein einziges IP-Paket erneut angefordert zu werden).

Aus diesen Gründen ist NFS in der Regel mit TCP ebenso schnell wie mit UDP, es sei denn, die Host-CPU ist besonders langsam, so daß der wesentlich höhere Protokoll-Aufwand von der langsamen CPU nicht verkraftet wird. Dies dürfte bei modernen Rechnern (zum Beispiel bei heute typischen PCs) keine Rolle spielen, ist aber bei einigen von NetBSD unterstützten Architekturen (zum Beispiel der VAX) oder im Embedded-Bereich zu beachten. Aus Gründen der Übertragungssicherheit kommt der Einsatz von UDP sowieso nur in LANs in Betracht, über unzuverlässige Leitungen und in WANs ist die Verwendung von TCP unabdingbar. Leider gibt es dabei ein Problem insoweit als NFS über TCP von den meisten Nicht-BSD-Betriebssystemen nicht unterstützt wird.

Zahl der nfsd

Je größer die Last auf einen NFS-Server ist, desto mehr *nfsd* sollten laufen. Wie in Kapitel 9.1.2 erläutert, sind die *nfsd* quasi Userland-Frontends zum Systemaufruf *nfssvc*(2), und *nfsrv_rcv* im Kernel kann nun die Last auf die *nfsd* aufteilen. Dabei wird die Last nicht gleichmäßig auf alle *nfsds* verteilt, sondern die *nfsds* sind in einer Queue arrangiert und ein Request wird an den ersten nicht beschäftigten *nfsd* durchgereicht. Ein bestimmter *nfsd* übernimmt also die Hauptlast, die anderen werden nur aktiv, wenn die in der Queue vorher kommenden beschäftigt sind.

Wegen dieses Schedulingverfahrens ist es ein Leichtes festzustellen, ob eine ausreichende Zahl von *nfsd* laufen. Zum Beispiel kann man mit Hilfe von *top* kontrollieren, wieviele *nfs*-Daemons gerade CPU-Zeit bekommen. Sind alle Daemons gleich stark beschäftigt, sollte die Zahl der *nfsds* erhöht werden. Genauer wird diese Methode, wenn man nach einer gewissen Laufzeit die erhaltene CPU-Zeit der *nfs*-Daemons auf einem NFS-Server mit *ps* ermittelt:

```
# ps auxww | grep nfsd
root   155  0.0  0.1     68    76 ?? Is   23Aug02   0:00.05 nfsd: master
root   157  0.9  0.1     48   160 ?? SL   23Aug02   1:29.73 nfsd: server
root   159  0.0  0.1     48   160 ?? SL   23Aug02   0:03.99 nfsd: server
root   160  0.0  0.1     48   160 ?? SL   23Aug02   0:01.19 nfsd: server
root   158  0.0  0.1     48   160 ?? SL   23Aug02   0:00.21 nfsd: server
```

In der obigen Liste sind die nfs-Daemons bereits nach erhaltener CPU-Zeit sortiert. Man erkennt, daß der *nfsd* mit der PID 157 bei weitem die meiste CPU-Zeit erhalten hat. Das entscheidende ist aber, daß der letzte Eintrag in der Liste, der *nfsd* mit der PID 158, so gut wie nie gebraucht wurde. Die Zahl der *nfsds* ist also ausreichend.

Größe der Puffer

Standardmäßig benutzt NFS über UDP acht Kbyte große Puffer, bei NFS über TCP sind diese 64 Kbyte groß. Falls es die Netzwerkhardware unterstützt, kann das Vergrößern der Puffergröße vor allem bei UDP bei einen Leistungsgewinn ergeben, da man im Mittel mit weniger RPCs auskommt.

Beim Vorhandensein alter Hardware hilft das Gegenteil: Durch die Verringerung der Größe der Puffer erreicht man, daß die NFS-Puffer vollständig in die Hardwarepuffer auf der Netzwerkkarte passen. Beispielsweise bei alten Western Digital/SMC Elite Ultra-Karten kann es sinnvoll sein, die NFS-Puffergröße auf vier Kyte zu halbieren (Details erfährt man mit *man we*).

Die Puffergröße kann mit Hilfe der Optionen *-r* und *-w* für *mount_nfs* beziehungsweise in der »/etc/fstab« im Options-Feld angegeben werden. Ein Beispiel aus der »/etc/fstab«:

```
flippy:/home    /mnt     nfs    rw,noauto,-r=4096,-w=4096    0    0
```