Aus dem Nähkästchen geplaudert: Interprozesskommunikation jenseits einfacher Signale

Dienstgespräche

Unix trennt seine Prozesse zwar sauber, lässt ihnen aber auch vielfältige Kommunikationsmöglichkeiten offen. Wenn es im System klemmt, braucht der Admin daher einen guten Überblick, um die muntere Gesprächsrunde seiner Linux-Prozesse wieder in Gang zu setzen. Marc André Selig

Müssen sich Prozesse untereinander verständigen, dann sind Signale das einfachste Mittel [1]. Diese wirkungsvolle Technik ist aber in vielerlei Hinsicht recht eingeschränkt. Da nur wenige verschiedene Signale existieren, bleibt der Informationsgehalt einer Botschaft stark begrenzt. Zudem dürfen Nutzer – Root ausgenommen – Signale nur an eigene Prozesse versenden, die Kommunikation mit Fremden ist unmöglich.

Es gibt aber eine Menge weiterer Kanäle, über die sich Prozesse miteinander unterhalten: Pipes, Named Pipes, Sockets, Lockfiles und File-Locks bis hin zur System-V-Interprozesskommunikation und Shared Memory.

Der Klassiker: Pipes

Jeder Prozess unter Unix verfügt über eine Standardeingabe und eine Standardausgabe. Über die Standardeingabe erhält er Daten: Wenn ein User in einem Anflug von Nostalgie direkt auf der Kommandozeile seine E-Mail verfasst, dann liest der von ihm benutzte Client »mailx« die Nachricht per Standardeingabe. Umgekehrt schreibt ein Prozess, wenn er nicht explizit einen anderen Kanal öffnet und benutzt, seine Ergebnisse auf die Standardausgabe. Der Text erscheint dann gewöhnlich im Terminalfenster. Darüber hinaus verfügt jeder Prozess auch noch über einen Kanal für Standardfehler, auf dem er über Probleme berichtet.

Eine Pipe koppelt einfach die Standardausgabe eines Prozesses an die Standardeingabe eines anderen. Was das eine Programm schreibt, kann das andere lesen. Diese effiziente und direkte Methode der Kommunikation setzt aber voraus, dass beide Programmen zusammen gestartet werden.

Ein typisches Beispiel für eine Pipe ist das Heraussuchen von Daten aus Logdateien. Die folgende Befehlszeile holt mit Grep alle »GET«-Einträge aus einer Apache-Logdatei, schneidet mit Cut das erste Feld (mit der IP-Adresse des Clients) heraus, sortiert die Ergebnisse numerisch und gibt dank Uniq jeden Hostnamen nur je ein Mal aus:

```
grep GET access_log | cut -d " " -f 1 | ➥
  sort -n | uniq > hosts.txt
```

Die Pipe bezieht sich nur auf Standardein- und -ausgabe, den Standardfehlerkanal leitet sie nicht um. Sollte eines der Programme eine Fehler- oder Warnmeldung ausspucken, würde diese nicht in »hosts.txt« landen, sondern direkt auf Shellebene zum Vorschein kommen. Wer den Standardfehler mit umleiten möchte, gibt das explizit an:

```
find /etc -type f 2>/dev/null -print0 | ➥
  xargs -0 grep -i imap 2>&1 | less
```

Unix gibt jeder offenen Datei eine fortlaufende Nummer. Die Standardeingabe hat Nummer 0, die Standardausgabe Nummer 1. Die 2 steht für den Standardfehlerkanal. Die normale Umleitung »> hosts.txt« bezieht sich auf den Filedeskriptor 1, also die Standardausgabe, während »2 > /dev/null« den Deskriptor 2 umleitet, den Standardfehler. Tipp: Ein modernes Linux bietet Pseudodateien für diese Streams, nämlich »/dev/fd/0«, »/dev/fd/1« und »/dev/fd/2«.

Gesucht – gefunden

Die obige Anweisung sucht also zunächst alle regulären Dateien im Verzeichnis »/etc«. Wenn dabei Fehlermeldungen auftauchen, landen sie in der Pseudodatei »/dev/null«, werden also verworfen. Die gefundenen Files leitet die Pipe zu »xargs« weiter. Dieses Kommando ruft »grep« auf und hängt dabei die von der Standardeingabe gelesenen

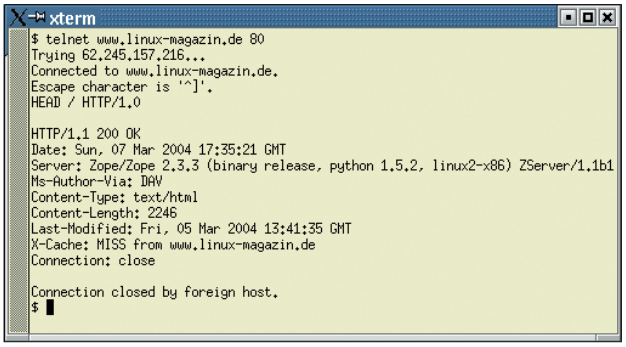

Abbildung 1: Per Telnet-Kommando sprechen Admins direkt mit ihren Servern: Die Eingabe »HEAD / HTTP/1.0« – gefolgt von einer Leerzeile – entlockt dem Apache Informationen über die Linux-Magazin-Homepage.

Daten an den Aufruf an. Grep sucht dann den String »imap« bei beliebiger Groß- und Kleinschreibung (Option »-i«). Die auftretenden Fehlermeldungen dupliziert »2 > &1« zur Standardausgabe und leitet beide an Less weiter. Die Kombination aus »find ... -print0« und »xargs -0« sorgt dafür, dass diese Aufrufkette auch mit Dateinamen umgehen kann, die Leerzeichen enthalten.

Pipes sind einfach und effizient. Dafür ist ihr Anwendungsbereich allerdings ziemlich eingeschränkt: Alle beteiligten Prozesse werden gleichzeitig auf dem gleichen Computer vom gleichen Benutzer gestartet.

Named Pipes

Eine besondere Variante der Pipes sind die Named Pipes (Fifo, first in first out). Sie koppeln die beiden Programme nicht unmittelbar aneinander, sondern verwenden zum Lesen und Schreiben spezielle Dateien. Diese Files werden mit »mkfifo« angelegt. Vorteil: Der Benutzer muss die beteiligten Programme nicht gleichzeitig starten, die Prozesse müssen nicht einmal beide dem gleichen User gehören. Eine mit »mkfifo /tmp/mas/test« angelegte Datei sieht in »ls -l /tmp/mas/test« wie folgt aus:

```
prw-------  1 mas  users  0 Mar  7 13:17 ➥
  /tmp/mas/test
```

Lese- und Schreibrechte haben dieselbe Bedeutung wie bei regulären Dateien. Auch sonst verhält sich das Fifo fast wie ein ganz normales File. Die Ausgabe beliebiger Kommandos lässt sich ins Fifo leiten:

```
ls /etc/mail/spamassassin >/tmp/mas/test
```

Zunächst blockiert der »ls«-Befehl jedoch, da die Ausgabe im Moment von niemandem gelesen wird. Der Aufruf »cat /tmp/mas/test« in einem anderen Terminal übernimmt diesen Part und gibt brav die Dateinamen aus (»local.cf« und »no-osiru.cf«).

Sehr praktisch sind Named Pipes, um Meldungen an den Nutzer weiterzuleiten, der gerade an der X11-Konsole arbeitet (Konsole meint hier die Arbeitsstation, nicht die gleichnamige KDE-Terminalemulation). Würden die Meldungen direkt auf dem Bildschirmhintergrund erscheinen, wäre das bei der Arbeit störend – ganz abgesehen davon, dass XFree die asynchrone Ausgabe von Text ohne zugehöriges Fenster gar nicht ohne weiteres unterstützt.

Verbreitet ist es, stattdessen ein Fifo namens »/dev/xconsole« anzulegen und per Konfiguration in »/etc/syslog.conf« Meldungen dorthin umzuleiten:

```
# Alle Meldungen an die X-Konsole kopieren
*.*     |/dev/xconsole
```

Die Ausgaben können Admins oder die jeweils eingeloggten Benutzer leicht mit »cat /dev/xconsole« betrachten.

Sockets

Sockets treiben die Kommunikation zwischen zwei Prozessen auf eine noch abstraktere Stufe, als dies bei den Named Pipes der Fall ist. Jetzt müssen die Prozesse nicht einmal mehr auf dem gleichen Computer laufen. Sockets dienen als Schnittstelle zu verschiedenen Kommunikationsprotokollen, vor allem den im Internet gebräuchlichen TCP (Transmission Control Protocol) und UDP (User Datagram Protocol). Andere Varianten wie die Unix-Domain-Sockets funktionieren dagegen nur lokal, also innerhalb eines Rechners.

Über Sockets laufen fast alle Formen der Kommunikation im Internet. Wer im Web surft, eine E-Mail verschickt oder eine Terminalverbindung aufbaut, benutzt Sockets. Die meisten Socket-basierten Applikationen entsprechen dem Client-Server-Konzept. Telnet oder Mozilla sind Beispiele für Clients, Inetd oder Apache (Abbildung 1) wären passende Server. Ein minimalistisches Beispiel für einen TCP-basierten Server ist in Listing 1 zu finden. In der Praxis bearbeitet der Prozess die erhaltenen Daten selten direkt, sondern gibt sie an einen eigenen Unterprozess weiter. So kann der Server in Ruhe auf die nächste Verbindung warten.

Sockets gibt es in mehreren Varianten mit vielen Optionen. Eine wichtige Unterscheidung ist die zwischen Unix-Domain- und Internet-Domain-Sockets. Letztere eignet sich für die grenzenlose Kommunikation. Wesentlich schneller sind Unix-Domain-Sockets, die zwar wieder eine Beschränkung auf den lokalen Computer einführen, dafür aber weniger Overhead tragen.

MySQL zum Beispiel verwendet Unix-Domain-Sockets automatisch, wenn der Server auf dem gleichen Computer arbeitet wie der Client. Ein einfaches »ls -l /var/lib/mysql/mysql.sock« zeigt:

```
srwxrwxrwx 1 mysql mysql  0 Feb 13 14:24 ➥
  /var/lib/mysql/mysql.sock
```

Lockfiles

Die bisher gezeigten Varianten der Interprozesskommunikation boten eine zunehmende Generalisierung. Ein Internet-

Listing 1: Einfacher TCP-Server in Perl

```
01 #!/usr/bin/perl -w
02 use strict;
03 use IO::Socket;
04
05 my $socket = IO::Socket::INET->new(
06   Listen => 5,
07   Proto => "tcp",
08   LocalPort => 2345,
09   ReuseAddr => 1,
10 )
11 or die "Problem: $!";
12
13 while (my $client = $socket->accept) {
14   my $line = <$client>;
15   print "Verbindung von " . $client->peerhost . ": $line";
16   print $client "Demo\r\n";
17 }
```

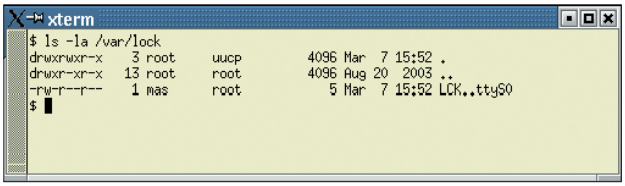

Abbildung 2: Für Lockfiles steht unter Unix das Verzeichnis »/var/lock« zur Verfügung. Es ist traditionell für die Gruppe »uucp« beschreibbar.

Domain-Socket kann praktisch alle Daten überallhin übertragen. Oft verlangt die Praxis aber nach Lösungen, die zwar mehr bieten als ein einfaches Signal, aber dennoch schlanker und mit weniger Overhead belegt sind als ein aufwändiger Socket.

Zu diesen Techniken gehören Lockfiles oder klassische Semaphore. Sie zeigen an, dass eine bestimmte Ressource derzeit belegt ist. Wenn beispielsweise ein Programm die serielle Schnittstelle verwendet, schreibt es seine Prozess-ID in eine bestimmte Datei und verkündet damit: Diese Schnittstelle ist jetzt für mich reserviert. Derartige Locks stehen typischerweise in »/var/lock«.

Das Lock-Verzeichnis

Wie Abbildung 2 bestätigt, ist das Verzeichnis »/var/lock« traditionsgemäß für die Gruppe »uucp« beschreibbar. Das UUCP-Programm (Unix to Unix copy) überträgt asynchron Dateien zwischen Rechnern, es hatte zur Zeit der Modemverbindungen Hochkonjunktur.

Wer eine über Semaphore verwaltete Ressource benutzen möchte, wird von Root in die »uucp«-Gruppe aufgenommen und kann fortan Lockfiles erstellen. Im Beispiel existiert eine Datei namens »LCK..ttyS0«. Sie zeigt an, dass der Nutzer »mas« das Gerät »/dev/ttyS0« exklusiv in Gebrauch hat. Ein anderer Prozess kann prüfen, ob diese Information noch aktuell ist, indem er die Lockdatei ausliest. Sie enthält die PID des zugehörigen Prozesses. Existiert dieser Prozess noch, gilt die Ressource als belegt. Ist der Prozess mit der dort abgelegten Nummer längst verschwunden, darf ein neuer Prozess die Lockdatei entfernen und neu anlegen.

Ein böswilliger Prozess könnte die Lockdatei sofort entfernen, ein ignoranter Prozess wird sie gar nicht erst überprüfen. Das ist eine wichtige Eigenschaft vieler Locking-Techniken: Es handelt sich oft um so genannte Advisory Locks, also bloße Ratschläge. Ein sauber programmiertes Programm hält sich daran. Das System hindert aber niemanden (mit den entsprechenden Zugriffsrechten), ohne Rücksicht auf die Locks die zugehörige Ressource zu verwenden.

Listing 2: Mail-Lock als Semaphor

```
01 #!/bin/sh
02 lockfile -ml
03 echo Jetzt kann ich eine Mail löschen.
04 lockfile -mu
```

Listing 3: Mail-Lock mit »flock()«

```
01 #!/usr/bin/perl
02 use Fcntl ':flock';
03 open (MAILBOX, ">>/var/mail/$ENV{'USER'}")
04     or die "Kann Mailbox nicht schreiben: $!";
05 flock(MAILBOX, LOCK_EX);
06 print "Jetzt kann ich eine Mail löschen.\n";
07 flock(MAILBOX, LOCK_UN);
08 close MAILBOX;
```

File-Locks

Wenn es sich bei der reservierten Ressource um eine normale Datei handelt, ist auf einem modernen Unix-System im lokalen Dateisystem kein Lockfile nötig. Stattdessen kann der Prozess einen Lock auf die Datei setzen. Er entscheidet auch, ob andere Prozesse noch lesen dürfen oder ob die Datei komplett gesperrt ist. Derartige File-Locks kommen häufig zusammen mit E-Mail-Systemen zum Einsatz, für die Mailspools in »/var/mail« oder »/var/spool/mail«.

Hier ist der sorgfältige Umgang mit Locks besonders wichtig, weil E-Mail ihrer Natur nach asynchron abgewickelt wird. Wenn ein Anwender eine Mail löscht, schreibt sein Client die Mailspool-Datei neu und lässt die gelöschte Mitteilung weg. Kommt nun gleichzeitig eine neue Mail an, wäre das Ergebnis ohne Locking undefiniert. Im günstigsten Fall wäre die vermeintlich gelöschte Mail hinterher noch vorhanden, im schlimmsten fände sich die neu angekommene Mail mitten in den übrigen Mails wieder und hätte einen Teil davon überschrieben.

Das beste Locking

Korrekterweise wird ein Mail User Agent (Mail-Client) den Mailspool durch einen Lock für sich reservieren, bevor er irgendwelche Änderungen daran vornimmt. Der Mail Delivery Agent befolgt diese Locks und liefert seinerseits Mail nur aus, wenn er zuvor erfolgreich einen Lock anlegen konnte.

Gerade für E-Mail existieren mehrere Locking-Varianten. Die Listings 2 und 3 zeigen wichtige Alternativen, die oft parallel zum Einsatz kommen. Ein Semaphor, ähnlich wie oben bei »/var/lock«, kann einfach in »/var/mail« abgelegt werden und hat den Vorteil, dass er auch bei verteilten Dateisystemen wie NFS oder AFS problemlos funktioniert.

Bei Netzwerk-Filesystemen funktionieren die File-Locks mit »flock()« oder »fcntl()« dagegen häufig nicht oder nicht zuverlässig. Wo sie funktionieren, haben sie aber einige Vorteile: File-Locks sind schneller und effizienter als Lockfiles; auch benötigen die beteiligten Prozess keine Schreibrechte für ein gemeinsames Verzeichnis. *(fjl)* ■

System-V-IPC

Moderne System-V-Unix-Systeme, also auch Linux, bieten neben den im Artikel genannten noch eine ganze Reihe von Möglichkeiten für IPC (Inter-Process Communication). Über die Library-Funktion »ipc()« stehen Semaphore und Warteschlangen für Nachrichten (Message Queues) zur Verfügung. Viele der vorgestellten Mechanismen führen zu ähnlichen Ergebnissen, entstammen aber der BSD-Tradition.

Eine recht interessante Technik ist Shared Memory. Dabei teilen sich zwei Prozesse einen gemeinsamen Bereich im Hauptspeicher. Das erlaubt einen besonders einfachen und effizienten Datenaustausch [2].

Infos

[1] Marc André Selig, „Handzeichen – Interprozesskommunikation mit Signalen“: Linux-Magazin 04/04, S. 76

[2] Dirk Henrici, „Gemeinsame Sache – Standard Template Library für Objekte im Shared Memory verwenden“: Linux-Magazin 12/03, S. 102