Patricia Jung

Software aus dem Quellcode installieren

Modul IF LIN 03 der Linux-Akademie im Rahmen der Informatica Feminale 2008 in Bremen

1 Vom Quellcode zur installierten Software

Nicht zu jeder Software gibt es ein für die eigene Distribution geeignetes Binärpaket; viele Tools findet frau grundsätzlich nur im Sourcecode. Die entsprechenden, mit `gzip` oder `bzip2` komprimierten Quellarchive tragen meist Dateiendungen wie `tar.gz`, `.tgz` bzw. `.tar.bz2` oder `.tbz`. Sie gilt es zunächst auszupacken. Ein aktuelles GNU-`tar` vorausgesetzt, geht das mit `tar -xzf` bei mit `gzip` gepackten, mit `tar -xjf` bei mit `bzip2` gepackten `tar`-Archiven.

In aller Regel erhält frau ein neues Verzeichnis, in dem sich meist eine Datei namens `README` und/oder `INSTALL` befindet. Sie gibt – mehr oder weniger brauchbare – Informationen darüber, wie frau zu einem Binärprogramm kommt. Wenn frau sich den Quellcode aus dem Versionskontrollsystem des Projekts (oft Subversion oder CVS, immer öfter auch GIT oder Mercurial) direkt herunterlädt, entfällt der Schritt des Entpackens.

Einfachen Projekten liegt im Projektverzeichnis eine Datei `Makefile` oder `makefile` bei, die Regeln aufstellt, wie aus den im Archiv mitgelieferten Dateien die ausführbaren Programme, die Dokumentation etc. werden und ggf. auch, wohin die übersetzten Dateien kopiert werden. Ein einfaches `make` reicht meist aus, um den Kompiliervorgang einzuleiten. Gerade in diesem Fall ist es aber angebracht, einen Blick ins Makefile zu werfen, um ggf. den einen oder anderen Parameter anzupassen. Dieser verrät auch, welche Argumente frau `make` mit auf den Weg geben kann, um gezielt bestimmte Aufgaben (wie zum Beispiel das Kopieren der fertig übersetzten Dateien ins System) zu lösen.

Sobald das Projekt komplexer wird, nach verschiedenen externen Bibliotheken und Hilfsprogrammen verlangt und gar noch portabel für verschiedene Plattformen sein soll, bietet ein festes Makefile zu viele Unwägbarkeiten. Dann liegt den Sourcen in der Regel ein `configure`-Skript bei. Um es laufen zu lassen, wechselt frau ins Sourcenverzeichnis und ruft

```
$ ./configure
```

auf. Läuft das Skript nicht durch, ist meist Nachinstallation von Third-Party-Tools (etwa eines Compilers und Linkers) und -Bibliotheken angesagt. `configure` tut nichts anderes, als sich mit einer Reihe von Tests auf dem System umzuschauen, zu überprüfen, ob die Voraussetzungen fürs Kompilieren dieser Software gegeben sind und – unter Zuhilfenahme der gefundenen Informationen über das lokale System – die nötigen Makefiles automatisch zu erstellen.

Viele Projekte benutzen das Programm `autoconf`, um das `configure`-Skript aus Vorgaben automatisch zu erstellen. Gerade wer öfter Projekte aus Entwicklungsrepositories auschecken möchte, ist gut beraten, die sogenannten Autotools `autoconf` und `automake` sowie ggf. auch `libtool` zu installieren und die entsprechenden Installationsanweisungen zu befolgen. Leider haben die damit generierten `configure`- und Makefiles im Gegensatz zu den meisten „handgestrickten“ Lösungen den Nachteil, schlecht lesbar zu sein.

Das `configure`-Skript legt u. a. auch fest, wohin welche Dateien letzten Endes installiert werden. In der Regel ist das die Verzeichnishierarchie unterhalb von `/usr/local`. Diesen

und andere Parameter kann frau in der Regel über `configure`-Kommandozeilenoptionen beeinflussen. Welche Optionen das `configure`-Skript der jeweiligen Software kennt, erfährt frau mit `./configure --help`. Zwar gibt es einige gängige Optionen, mit denen die meisten `configure`-Skripte umgehen können, aber prinzipiell bleibt dies – wie auch die Argumente für `make` – den Entwicklern und Entwicklerinnen der Software überlassen.

Läuft `configure` problemlos durch, startet `make` den Übersetzungsvorgang. Die häufigste Ursache dafür, dass hier etwas schiefgeht, sind schlampig erstellte `configure`-Skripte, die eben doch nicht alle Voraussetzungen vernünftig geprüft haben. An zweiter Stelle kommen nachlässige Programmierung und ungenügendes Testen auf mehreren Plattformen.

Denn was sich auf einem System problemlos kompilieren lässt, kann auf einem anderen Probleme machen. Wer sich in der jeweiligen Programmiersprache (meistens C oder C++) auskennt, ist hier sicher in der Lage, die eine oder andere Sache auszubügeln, doch bei komplexer Software ist das oft nicht so einfach. Dann bleibt immer noch die Möglichkeit, eine andere Version auszuprobieren, im Netz nach Leidensgenoss(inn)en und deren Problemlösungen zu fahnden oder aber den Autor(inn)en der Software eine Mail zu schreiben.

Viele reagieren darauf ausgesprochen hilfsbereit, sofern frau soviel relevante Informationen wie möglich (welche Distribution, welche Compilerversion, welche Version benötigter Bibliotheken, welche Fehlermeldung etc.) mitliefert. Große Projekte nutzen Bug-Tracking-Systeme wie Bugzilla, in die diese Informationen eingetragen werden. Gute Bugreports sind zwar eine wichtige Säule, auf der das Prinzip Open Source basiert, dennoch scheint es, dass manche Projekte diese Art der Hilfe mittlerweile eher als lästig empfinden. Manche stellen so abschreckende Bedingungen für die Akzeptanz eines Bugreports, dass selbst gutwillige und technisch versierte Nutzerinnen davon Abstand nehmen werden, andere ignorieren speziell Usability-Bugs auch dann noch, wenn diese über lange Zeit hinweg von mehreren Leuten gemeldet werden.

Für die endgültige Installation braucht frau `root`-Rechte, sofern sie nicht in ein eigenes Verzeichnis installiert: `make install` kopiert alle relevanten Daten an Ort und Stelle. Alternativ baut das Tool `checkinstall`[URL 1] ein (intermediäres) `.rpm`- oder `.deb`-Paket und installiert es, sodass zur Deinstallation der Paketmanager zum Zuge kommen kann.

Wer erst einmal wissen möchte, was `make install` vorhat, lässt den Befehl zunächst ohne `root`-Rechte laufen: Das gibt Fehlermeldungen, wenn das Makefile Dateien in Verzeichnisse kopieren will, in denen eine unprivilegierte Nutzerin keine Schreibrechte hat. Sinnvoller ist zwar die Trockenübung `make -n install`, die nur so tut, als ob `make install` ausgeführt werden soll, aber speziell bei von den Autotools generierten Makefiles ist die Ausgabe selten allgemeinverständlich.

2 `make` und Makefiles

`make` oder auch alternative Build-Systeme wie `jam`[URL 2] führen lediglich Regeln aus, die in einer Datei in einem bestimmten Format abgelegt sind. Obwohl sie meist im Zusammenhang

mit Programmierprojekten eingesetzt werden, hängt es vollkommen von der Regelschreiberin ab, was ein `make`-Aufruf tut!

Sofern frau `make` nicht über den Parameter `-f` sagt, in welcher Datei sich die Regeln verstecken, erwartet das Tool diese in einer Datei namens `makefile` oder `Makefile`:

```
SRC=test.txt

help:
    @(echo -e "make help\t zeigt diese Hilfe an.")
    @(echo -e "make install\t installiert ein Testfile namens $(SRC).")
    @(echo -e "make deinstall\t loescht das Testfile.")

install:
    echo -e "Dies ist ein Testfile.\n" > $(SRC)

deinstall: $(SRC)
    rm $(SRC)
```

Dieses Beispiel definiert zunächst ein *Target* namens `help`. Ruft frau `make` mit diesem Argument auf (also `make help`), führt `make` die darunter angegebenen und mit einem *Tabulator* eingerückten Aktionen aus. Das erste Target in einem Makefile gilt als Default, wenn `make` ohne Argumente ausgeführt wird.

✎ Was passiert, wenn frau den `echo`-Befehl *nicht* in `@()` einschließt?

Gängige Targets sind `install` und `clean`; sehen Softwareautor(inn)en die Möglichkeit der Deinstallation vor, heißt das entsprechend Target meist `deinstall`. Es ist aber eher selten anzutreffen, was zu der Problematik führt, dass sich selbstkompilierte Software selten ohne Verrenkungen rückstandslos deinstallieren lässt.

3 Überlegungen zum Installationsziel

Generell gehört aus dem Quellcode installierte Software *nicht* in Verzeichnisse, die dem Paketmanagement der Distribution unterliegen, also etwa nicht nach `/bin` oder `/usr/bin`. Stattdessen sieht der Linux Filesystem Hierarchy Standard die `/usr/local`-Hierarchie dafür vor. Dementsprechend nutzen die meisten Projekte die darin befindlichen Verzeichnisse per Default als Ablageplatz.

Wer viel aus dem Quellcode installiert, verliert aber leicht den Überblick, welche Datei zu welcher Software gehört. Damit sind der rückstandsfreien Deinstallation Grenzen gesetzt. Wer nicht auf `checkinstall` setzt (nicht jede Software lässt sich damit problemlos installieren), sollte also Vorkehrungen treffen und etwa jedem Softwarepaket ein eigenes Projektverzeichnis wie `/usr/local/`*`extra-verzeichnis`* spendieren. Das hat allerdings den Nachteil, dass man mit jeder neuen Software auch die `PATH`-Variable erweitern muss,

sollen die entsprechenden Binaries ohne explizite Pfadangabe aufrufbar sein. Ein alternativer Weg führt hier über die Verlinkung nach `/usr/local/bin` etc., automatisierbar über das Werkzeug `stow`.[URL 3]

Wenn frau auf einem Rechner keine `root`-Rechte hat, lässt sich zwar keine Software per Paketmanagement installieren, wohl aber selbst kompilieren und etwa unterhalb des Home-Verzeichnisses ablegen. Dann muss frau das Installationsverzeichnis in jedem Fall anpassen (und ggf. `$HOME/bin` in die `PATH`-Variable aufnehmen).

4 `configure`

Weil die Anpassung des Installationsverzeichnisses ein gängiger Änderungswunsch ist, bieten fast alle `configure`-Skripte dafür eine quasi-standardisierte Kommandozeilenoption an: `--prefix` (manchmal auch `-prefix`):

```
$ ./configure --prefix=/usr/local/extra-verzeichnis
```

✎ Lade das Paket `stow` unter `http://ftp.debian.org/debian/pool/main/s/stow/stow\_1.3.3.orig.tar.gz` herunter und installiere es in Deinem Homeverzeichnis!

✎ Lade die aktuelle Version des Qt-Toolkits von `http://trolltech.com/developer/downloads/qt/x11` herunter und installiere sie in Deinem Homeverzeichnis!

✎ Welche Version des C-Compilers `gcc` käme auf dem Pool-Rechner zum Einsatz? Finde den passenden Aufruf mit der `gcc`-Option `--help` heraus!

✎ Eine Software, bei der wir ein bisschen arbeiten müssen, um sie lokal lauffähig zu installieren, ist die größtenteils in Perl geschriebene Monitoring-Software Munin. Lade dazu das Quellarchiv `http://downloads.sourceforge.net/munin/munin_1.3.4.tar.gz` herunter und bringe den Munin-Node-Daemon zum Laufen!

Lösungen

✎ Was passiert, wenn frau den `echo`-Befehl *nicht* in `@()` einschließt?

`make` gibt normalerweise die Befehle, die es ausführt, auf der Standardausgabe aus. `@()` unterdrückt dies.

✎ Lade das Paket `stow` unter `http://ftp.debian.org/debian/pool/main/s/stow/stow\_1.3.3.orig.tar.gz` herunter und installiere es in Deinem Homeverzeichnis!

```
$ ncftpget http://ftp.debian.org/debian/pool/main/s/stow/stow_1.3.3.orig.
tar.gz
$ tar -xzvf stow_1.3.3.orig.tar.gz
stow-1.3.3/
stow-1.3.3/README
[...]
$ cd stow-1.3.3
$ ./configure --prefix=$HOME
$ make
make: Nothing to be done for 'all'.
$ make install
```

Sorry – das war ein extrem unspektakuläres Beispiel: `stow` ist nur ein Perlskript, muss also nicht kompiliert werden. Aus diesem Grund bekommen wir nach dem `make`-Aufruf keinen Compiler-Output zu sehen, sondern die lapidare Meldung, dass es kein entsprechendes Make-Target gibt.

Dafür bietet dieses Beispiel die Möglichkeit, sich ein sehr einfaches `configure.in` anzuschauen, aus dem `autoconf` das `configure`-Skript erzeugt. Letzteres macht aus dem seinerseits von `automake` aus `Makefile.am` generierten `Makefile.in` das endgültige Makefile.

✎ Lade die aktuelle Version des Qt-Toolkits von `http://trolltech.com/developer/downloads/qt/x11` herunter und installiere sie in Deinem Homeverzeichnis!

Das Qt-`configure`-Skript nutzt lange Optionen, die mit einem einfachen Minus (statt einem doppelten) eingeleitet werden: `-prefix $HOME/qt-4.4.1`. Sowohl der `configure`- als auch der folgende `make`-Lauf dauern sehr lange.

Prinzipiell ist das Setzen des Installationspräfix' bei einer individuellen Qt-Installation aber gar nicht unbedingt nötig, denn diese lässt sich bereits jetzt, ohne nachfolgendes `make install`, nutzen:

```
$ export QTDIR=$(pwd)
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$QTDIR/lib
$ ./bin/assistant &
```

Allerdings misslingt den so gestarteten Tools der Zugriff auf die mitgelieferte Dokumentation. Nach einem `make install` kann man zumindest für die Qt-eigenen Tools auch auf das Setzen des `LD_LIBRARY_PATH` verzichten.

Die Umgebungsvariable `QTDIR` legt das Oberverzeichnis der derzeit benutzten Qt-Installation fest. Damit lässt sich mit einer anderen Qt-Version entwickeln, als systemweit installiert ist.

✎ Welche Version des C-Compilers `gcc` käme auf dem Pool-Rechner zum Einsatz? Finde den passenden Aufruf mit der `gcc`-Option `--help` heraus!

```
$ gcc --help
Usage: gcc [options] file...
Options:
--help                   Display this information
(Use '-v --help' to display command line options of sub-processes)
-dumpspecs               Display all of the built in spec strings
-dumpversion             Display the version of the compiler
[...]
$ gcc -dumpversion
2.95.4
```

Mit der Option `--help` lässt sich den meisten Kommandozeilenprogrammen eine Kurzhilfe entlocken: Sollten sie über diese Option nicht verfügen, sorgt diese Fehlbedienung in der Regel dafür, dass die Hilfeseite trotzdem erscheint.

✎ Eine Software, bei der wir ein bisschen arbeiten müssen, um sie lokal lauffähig zu installieren, ist die größtenteils in Perl geschriebene Monitoring-Software Munin. Lade dazu das Quellarchiv `http://downloads.sourceforge.net/munin/munin_1.3.4.tar.gz` herunter und bringe den Munin-Node-Daemon zum Laufen!

```
$ tar -xzf munin_1.3.4.tar.gz
$ cd munin-1.3.4/
$ less INSTALL
[...]
To install a node:

        - edit Makefile.config
        - create the group "munin"
        - make install-node install-node-plugins (NOTE: This installs
          all plugins in the distribution, no matter what you already
          have, if you want to save some plugins you've installed yourself
          or have customized please make a backup copy).
```

```
        - decide which plugins to use.  The quick auto-plug-and-play
          solution:
            munin-node-configure --shell --families=contrib,auto | sh -x

        - start the node agent. You probably want an init-script for
          this and you might find a good one under build/dists
[...]
$ vi Makefile.config
```

Munin bringt kein `configure`-Skript mit. Die nötigen Anpassungen müssen wir in der Datei `Makefile.config` vornehmen. Wenn wir das Installationspräfix

```
PREFIX     = $(DESTDIR)/opt/munin
```

in

```
PREFIX     = $(DESTDIR)/$(HOME)/munin
```

ändern, müssen wir uns jetzt nur noch um die Variablen kümmern, die sich nicht auf das `PREFIX` beziehen. Außerdem können wir für den Munin-Node die Variablen vernachlässigen, die nur für den „Server“ (tatsächlich: den Master) gelten.

```
CONFDIR    = $(PREFIX)/etc
DBDIR      = $(PREFIX)/var/rrd
LOGDIR     = $(PREFIX)/var/log
STATEDIR   = $(PREFIX)/var/run
```

Zeit für einen ersten „`make install-node`“-Versuch:

```
[...]
mkdir: /usr/local/lib/perl5/site_perl/5.005/Munin: Permission denied
make: *** [install-node-non-snmp] Error 1
```

In `/usr/local` haben wir natürlich keine Schreibrechte. Wir müssen also in `Makefile.config` herausfinden, wo das Makefile das Munin-eigene Perlmodul hinkopieren will:

```
# Server only - Where to install the perl libraries
PERLLIB    = $(DESTDIR)$(shell $(PERL) -V:sitelib | cut -d"'" -f2)
```

Der Kommentar lügt: Diese Verzeichnis muss auch für den Node schreibbar sein. Wir definieren kurzerhand eines, das sich unter unserer Kontrolle befindet. Das hat allerdings den Nachteil, dass wir später vermutlich ein paar Perlskripten sagen müssen, dass sie Perlmodule auch dort suchen müssen:

```
PERLLIB    = $(DESTDIR)/$(PREFIX)/perl
```

Ein weiterer Testlauf . . .

```
$ make install-node
chown nobody:munin ///home/trish/munin/var/rrd/plugin-state
chown: munin: illegal group name
make: *** [install-node-non-snmp] Error 1
```

`Makefile.config` erlaubt es auch, die Nutzerinnen, mit deren Rechten der Munin-Node und die Plugins ausgeführt werden, zu ändern. Da das alles unter unserer Regie laufen muss, kommt hier unser eigener Nutzerinnenname hin:

```
# User to run munin as
USER       = trish
GROUP      = trish

# Default user to run the plugins as
PLUGINUSER = trish
```

Damit läuft nun alles durch, und wir können auch die Plugins installieren:

```
$ make install-node
[...]
Done.
$ make install-node-plugins
[...]
./install-sh -m 0644 build/node/Plugin.pm ///home/trish/munin/perl/Munin/
```

Jetzt können wir prüfen, ob der Daemon funktioniert:

```
$ ~/munin/sbin/munin-node
$ telnet localhost 4949
Trying 127.0.0.1...
Connected to localhost.localdomain.
telnet: Unable to connect to remote host: Connection refused
$ ps -auxw|grep munin-node
trish 9574  0.0  0.0   2924   812 pts/12   R+   16:04   0:00 grep munin-node
```

Es käuft kein Daemon. Warum? Die Antwort findet sich im Logfile:

```
$ tail ~/munin/var/log/munin-node.log
2008/08/27-16:03:39 MyPackage (type Net::Server::Fork) starting! pid(9562)
Binding to TCP port 4949 on host *
2008/08/27-16:03:39 Couldn't chown "///home/trish/munin/run/munin-node.pid" [Op
eration not permitted]

  at line 488 in file /usr/share/perl5/Net/Server.pm
2008/08/27-16:03:39 Server closing!
```

Ein Rechteproblem, das sich durch Änderung der Konfigurationsdatei `munin-node.conf` lösen lässt:

```
user root
group root
```

Die Defaultkonfiguration überschreibt hier unsere beim „Kompilieren“ festgelegten Einstellungen. Aus `root` müssen wir hier unsere eigenen User- und Gruppennamen eintragen. Dann startet auch der Server:

```
$ ~/munin/sbin/munin-node
$ telnet localhost 4949
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
# munin node at localhost.localdomain
quit
```

Bevor wir Plugins per Link einbinden, prüfen wir deren Funktionalität:

```
$ ~/munin/lib/plugins/uptime
uptime.value 1.10
$ ln -s ~/munin/lib/plugins/uptime ~/munin  ~/munin/etc/plugins/uptime
$ ~/munin/lib/plugins/df
Can't locate Munin/Plugin.pm in @INC (@INC contains: /etc/perl /usr/local/lib
/perl/5.8.7 /usr/local/share/perl/5.8.7 /usr/lib/perl5 /usr/share/perl5 /usr/
lib/perl/5.8 /usr/share/perl/5.8 /usr/local/lib/site_perl .) at /home/trish/m
unin/lib/plugins/df line 30.
BEGIN failed--compilation aborted at /home/trish/munin/lib/plugins/df line 30.
```

Das Shell-Plugin `uptime` funktioniert gut, aber das Perlskript `df` beschwert sich wie erwartet darüber, dass es `Munin/Plugin.pm` nicht finden kann, das in `~/munin/perl` liegt. Jetzt sind minimale Perlkenntnisse gefragt, denn wir müssen `df` um die zweite der folgenden Zeilen erweitern:

```
use strict;
use lib '/home/trish/munin/perl';
use Munin::Plugin;
```

Dann funktioniert auch dieses Skript:

```
$ ~/munin/lib/plugins/df
_dev_hda1.value 52
varrun.value 1
varlock.value 1
udev.value 1
devshm.value 0
lrm.value 3
$ ln -s ~/munin/lib/plugins/df ~/munin/etc/plugins/df
```

Nachdem wir den laufenden Munin-Node-Daemon gekillt und neu gestartet haben, können wir diese Plugins auch darüber abfragen – ganz so, wie es der Munin-Master tun würde:

```
$ ps -auxw|grep munin-node
[...]
trish 13997  0.0  0.3  7848  4844 ?  Ss  15:28  0:00  /home/trish/munin/sbin/
munin-node
$ kill 13997
$ ~/munin/sbin/munin-node
$ telnet localhost 5050
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
# munin node at localhost.localdomain
list
df uptime
fetch df
_dev_hda1.value 53
varrun.value 1
varlock.value 1
udev.value 1
devshm.value 0
lrm.value 3
.
quit
```

Online-Ressourcen

URL 1 checkinstall: http://asic-linux.com.mx/~izto/checkinstall/

URL 2 jam: http://www.perforce.com/jam/jam.html

URL 3 stow: http://ftp.debian.org/debian/pool/main/s/stow/

Impressum

Dieses Skript basiert passagenweise auf der für die Informatica Feminale erarbeiteten Kursunterlage „Linux ist weiblich“ in deren 5. Auflage. Die Autorin bedankt sich bei Sibylle Nägle und Gabriele Pohl für die Mitarbeit an der 1. Auflage (2001) bzw. der 5. Auflage (2005).

Satz: TEX/LATEX unter Linux