Packaging and RPM

Matt Behrens

DevOps West Michigan

~~December 16~~ January 13, 2013

Software Distribution Options

Source Archive

- Takes the least effort
- Highly portable
- No installation record
- Builds can vary based on available dependencies
- Everyone does things a little differently

Packages

- Use existing OS tools to build, install, upgrade
- Administrators already know how to deal with them
- Installation records available and tool-able
- Emphasis on describing installed software, rather than scripting its installation

Packages

- Metadata allows tools to discover and install dependencies
- Distinct effort required for each tool set
- Need to research and adhere to OS rules
- But a good package blends in well with the OS

Package Contents

- Packing lists or manifests
- Build artifacts (e.g. binaries, data)
- OS integration components (e.g. init scripts)
- Install-time and other-time scripts

A Brief Walk Through History

Why I Love Packages

- Always been a bit of a organizational perfectionist
- Records of installed software
- Reduction in system artifacts I’m not aware of
- Builds used to take an awfully long time

Blast From the Past: OpenBSD

- OpenBSD ports collection contains Makefile instructions and patches to install software
- Packing lists generally generated dynamically
- Installations happen into special “fake” directories which are then bundled up into packages
- Packages installed to satisfy dependencies for further builds

My First OpenBSD Port

```
# $OpenBSD: Makefile,v 1.1.1.1 2000/06/02 03:01:17 kevlo Exp $

DISTNAME=        pdmenu_1.2.61
PKGNAME=         pdmenu-1.2.61
CATEGORIES=      misc
NEED_VERSION=    1.273

HOMEPAGE=        http://kitenet.net/programs/pdmenu/

MAINTAINER=      matt@zigg.com

PERMIT_PACKAGE_CDROM=       Yes
PERMIT_PACKAGE_FTP=         Yes
PERMIT_DISTFILES_CDROM=     Yes
PERMIT_DISTFILES_FTP=       Yes

MASTER_SITES=    http://kitenet.net/programs/code/pdmenu/

LIB_DEPENDS=     slang.14::devel/libslang

GNU_CONFIGURE=   Yes
CONFIGURE_ENV=   SLANG_LIB_LOC=${LOCALBASE}/lib \
                 SLANG_H_LOC=${LOCALBASE}/include/slang.h
FAKE=            Yes
FAKE_FLAGS=      INSTALL_PREFIX=${WRKINST}
WRKDIST=         ${WRKDIR}/pdmenu

.include <bsd.port.mk>
```

Blast From the Past: Solaris

- Sunfreeware offered nice packaged builds but no way to change them
- At this point I was really enamored with OpenBSD ports, so I implemented my own clone for Solaris
- Solaris Package System used Solaris make and Solaris tools (pkgproto, pkgtrans) to build Solaris packages
- Still available at solpkg.berlios.de, probably bitrotted

Today: RPMs for RHEL/ CentOS

- Inherited a system based on an existing custom CentOS distro that extracted a bunch of tarballs in postinstall and ran scripts
- Challenge: make it upgradable!
- Logical tool: RPM

What’s in an RPM?

The .spec File

- To make an RPM, you need a source archive and a .spec file
- Contains metadata about the package
- Contains scriptlets for all phases of the project's lifecycle
- Introductory material: Maximum RPM www.rpm.org/max-rpm
- Modern material: Fedora, www.rpm.org, others

Metadata

```
Summary: Entropy-gathering daemon using the system timer
Vendor: Code Blue Corporation
Name: timer-entropyd
Version: 0.1_codeblue.1
Release: 2
License: GPL
Group: Utilities/System
URL: http://www.codeblue.com/
BuildRoot: %(mktemp -ud %{_tmppath}/%{name}-%{version}-%{release}-XXXXXX)
Source0: timer_entropyd-%{version}.tgz
Source1: timer_entropyd

%description
Uses the system timer to keep the Linux entropy pool full, increasing the
quality of cryptographic operations as well as keeping tasks like key
generation which demand high-quality randomness from starving.
```

Version Diversion

- 0.1_codeblue.1-2 breaks up into
  - 0
  - 1
  - codeblue
  - 1
  - -2 (build number)
- Everyone does things differently! Pay attention!

Scriptlets

```
%prep
%setup -c

%build
make

%install
rm -rf %{buildroot}
mkdir -p %{buildroot}/usr/sbin
make DESTDIR=%{buildroot} install
mkdir -p %{buildroot}/etc/init.d
cp %{SOURCE1} %{buildroot}/etc/init.d/

%post
chkconfig timer_entropyd on
service timer_entropyd start

%clean
rm -rf %{buildroot}

%files
%attr(755, root, root) /etc/init.d/timer_entropyd
%attr(755, root, root) /usr/sbin/timer_entropyd
```

Target System Scriptlet Tasks

- %pre and %post run on the target system
  - Database schema upgrades
  - Migration of existing configurations
  - Creation of accounts to run services under
  - Service enablement and startup

Scriptlet Arguments

- %pre runs before install, %preun before uninstall
- %post, %postun, same thing
- On first install, %pre and %post are passed 1
- On last uninstall, %preun and %postun are passed 0
- Can use this property to distinguish new installs, upgrades (and last gasps before upgrades), and final uninstalls

Source RPMs

- Has the extension .src.rpm
- Contains everything needed to build a binary RPM
- .spec file, source archives, auxiliary files…
- Can be busted open with rpm2cpio:

```
rpm2cpio foo.src.rpm | cpio -idmv
```

From .spec to .src.rpm

- Set up an rpmbuild tree, e.g.
  `mkdir -p rpmbuild/{SOURCES,BUILD,RPMS,SRPMS}`
- Link or copy your source tar ball et al. into rpmbuild/SOURCES
- Use rpmbuild to generate a .src.rpm, e.g.
  `rpmbuild --define '_topdir ``pwd``/rpmbuild' -bs .spec`
- The source RPM has everything necessary to build a binary RPM

From .src.rpm to .rpm

- rpmbuild --rebuild can be used to build directly on your system, but…
- Your build platform must be identical to your target platform, including dependencies
- Unpredictable builds, undeclared dependencies
- (That said, I still use it for small, build-free noarch packages)

Mock

- Dynamically builds a chroot environment with specified build dependencies in a source RPM
- Ensures that your source RPM is specified completely and correctly
- Only reliable way I've found to build i386 binaries on x86_64 (with help from setarch)
- My 0.6 magic incantation:
  `mock --configdir=. --debug -r mock.cfg rpmbuild/SRPMS/*.src.rpm`
  Doesn't work with later versions… someday I'll upgrade 😉

Alternative RPM Tools

- Some systems can make their own RPMs
- e.g. Python's bdist_rpm target to setup.py
- Requires some additional configuration, too…
  - To work around bugs
  - To include non-Python files and RPM scripts

Putting It All Together

- Everything goes in source control
- Store extracted source code for your packages in src
- Set up a Makefile to create a tarball from src that the .spec file (and rpmbuild) can use
- Store .spec.in and mock.cfg.in files with see-replaceable tokens for arch, version, etc.

Lots of RPMs?

- Collect all your RPMs
- Use createrepo
- Point yum (et al.) at your new repository
- createrepo can create repos for disc media or network accessibility

What I'd Do Differently

- Track Fedora rules more closely
  - Less dissonance with system packages
    - Better fit for initial install environment
    - Ship unconfigured and let a deployment tool do the configuration
- Build a wall between source and packaging

Fin