Chapter 5 Package Management Systems - Notes

5.3 Learning Objectives:

- Explain w hy softw are package management systems are necessary.
- Understand the function of both binary and source packages.
- Enumerate the main available package management systems.
- Understand w hy tw o levels of utilities are needed: one that deals w ith just bare packages, and one that deals w ith dependencies among packages.
- Explain how creating your ow n package enhances the control you have over exactly w hat goes in softw are and how it is installed.
- Understand the role of source control systems, and git in particular.

5.4 Software Packaging Concepts

Package management systems supply tools that allow system administrators to automate installing, upgrading, configuring and removing softw are packages in know n, predictable, consistent manner. These systems:

- Gather and compress associated softw are files into single package (archive), w hich may require one or more other packages to be installed first.
- Allow for easy softw are installation or removal
- Can verify file integrity via internal database
- Can authenticate origin of packages
- Facilitate upgrades
- Group packages by logical features
- Manage dependencies betw een packages

Given package may contain executable files, data files, documentation, installation scripts, configuration files. Also included -> **metadata** attributes such as version numbers, checksums, vendor information, dependencies, descriptions, etc.

Upon installation, all information stored locally into internal database, w hich can be queried for version status and update information.

5.5 Why Use Packages?

Softw are package management systems w idely seen as one of biggest advancements Linux brought to enterprise IT environments. By keeping track of files, metadata in automated, predictable, reliable w ay, system administrators can use package management systems to make installation processes scale to thousands of systems w ithout requiring manual w ork on each individual system. Included features:

- Automation: No need for manual installs and upgrades
- Scalability: Install packages on one system, or 10,000 systems
- Repeatability and predictability
- Security and auditing

5.6 Package Types

Several different types of packages:

- **Binary** packages contain files ready for deployment, including executable files and libraries. Architecture-dependent, must be compiled for each type of machine
- **Source** packages used to generate binary packages. Should always be able to rebuild binary package (eg. by using `rpmbuild --rebuild` on **RPM**-based systems) from source package. One source package can be used to multiple architectures
- **Architecture-independent** packages contain files, scripts that run under script interpreters, + documentation, configuration files
- **Meta-packages**: essentially groups of associated packages that collect everything needed to install relatively large subsystem, eg. desktop environement, office suite, etc.

Binary packages -> ones that system administrators have to deal with most of the time.

On 64-bit systems that can run 32-bit programs, may have two binary packages installed for given program, one with **x86-64** or **amd64** in name, and other with **i386** or **i686** in name.

Source packages helpful in keeping track of changes and source code used to come up with binary packages. Usually not installed on system by default, but can always be retrieved from vendor.

5.7 Available Package Management Systems

Two very common package management systems:

1. **RPM** (**Red Hat Package Manager**): system used by all Red Hat-derived distributions (eg. Red Hat Enterprise Linux, CentOS, Scientific Linux, SUSE and its related community openSUSE distribution)
2. **dpkg** (**Debian Package**): system used by all Debian-derived distributions, including Debian, Ubuntu, Linux Mint.

Other package management systems: **portage/emerge** used by Gentoo, **pacman** used by Arch, specialized ones used by Embedded Linux systems and Android.

Another ancient system: supply packages as **tarballs** without any real management or clean removal strategies. Approach still marks Slackware, one of oldest Linux distributions.

Most of time, either **RPM** or **dpkg**, only ones considered in course.
RPM_logo

apt

5.8 Packaging Tool Levels and Varieties

Two levels to packaging systems:

1. **Low Level Utility**: simply installs/removes single package, or list of packages, each one individually/specifically named. Dependencies not fully handled, only warned about:

   - If another package needs to be installed, first installation will fail
   - If package needed by another package, removal will fail

   rpm and **dpkg** utilities play this role for packaging systems that use them.

2. **High Level Utility**: solves dependency problems:

   - If another package/group of packages needs to be installed before software can be installed, such needs will be satisfied
   - If removing package interferes with another installed package, administrator given choice of aborting or removing all affected software

yum, **dnf**, **zypper** and Package Kit utilities take care of dependency resolution for rpm systems, **apt**, **apt-cache** and other utilities take care of it for dpkg systems.

In course, only discuss command line interface to each packaging systems. Graphical frontends used by each Linux distribution *can* be useful, w ould like to be less tied to any one interface, have more flexibility.

5.9 Package Sources

Every distribution -> one or more package **repositories** w here system utilities go to obtain softw are, update w ith new versions. Job of distribution to make sure all packages in repositories play w ell w ith each other.

Alw ays other external repositories w hich can be added to standard distribution-supported list. Sometimes, closely associated w ith distribution, only rarely produce significant problems (eg. **EPEL** (**E**xtra **P**ackages for **E**nterprise **L**inux), set of version-dependent repositories, fit w ell w ith RHEL since source = Fedora + maintainers close to Red Hat)

How ever, some external repositories not very w ell constructed or maintained. Eg. w hen package is updated in main repository, dependent packages may not be updated in external one, can lead to one form of dependency hell.

5.10 Creating Software Packages

Building ow n custom softw are packages -> easy to distribute/install ow n softw are. Almost every version of Linux has some mechanism for doing this.

Building ow n package -> allow s to control exactly w hat goes in the softw are + exactly how it is installed. Can create package so installing it runs scripts that perform all tasks needed to install new softw are and/or remove old softw are:

- Creating needed symbolic links
- Creating directories as needed
- Setting permissions
- Anything that can be scripted

No discussion of mechanisms of how to build **.rpm** or **.deb** packages, as that question mostly for developers, rather than administrators.

5.11 Revision Control System

Softw are projects become more complex to manage as either size of project increases, or number of contributing developers increases.

To organize updates/facilitate cooperation, many different schemes available for source control. Standard features of such programs: ability to keep accurate history (log) of changes, ability to back up to earlier releases, coordinate possibly conflicting updates from more than one developer, etc.

Source Control Systems (or Revision Control Systems, as also commonly called) fill role of coordinating cooperative development.

5.12 Available Source Control Systems

No shortage of available products, both proprietary/open. Brief list of products released under **GPL** license includes:

Product	URL
RCS	https://w w w .gnu.org/softw are/rcs

CVS	https://www.nongnu.org/cvs
Subversion	https://subversion.apache.org
git	https://www.kernel.org/pub/software/scm/git
GNU Arch	https://www.gnu.org/software/gnu_arch
Monotone	https://www.monotone.ca
Mercurial	https://mercurial-scm.org

Will only focus on **git**, widely used product which arose from Linux kernel development community. **git** risen to dominant position in use for open source projects in remarkable short time, often used even in closed source environments.

5.13 The Linux Kernel and the Birth of git

Linux kernel development system has special needs -> widely distributed throughout world, with literally thousands of developers involved. All done very publicly, under GPL license.

For long time, no real source revision control system. Then, major kernel developers went over to use of BitKeeper, commercial project which granted restricted use license for Linux kernel development.

However, in very public dispute over licensing restrictions in spring of 2005, free use of BitKeeper became unavailable for Linux kernel development.

Response: development of git, original author Linus Torvalds. Source code for git here, full documentation here.

5.14 How git Works

Technically, git not source control management system in usual sense. Basic units it works with are not files. Has two important data structures: object database + directory cache.

Object database contains three objects:

- Blobs: chunks of binary data containing file contents
- Trees: sets of blobs including file names/attributes, giving directory structure
- Commits: changesets describing tree snapshots

Directory cache captures state of directory tree.

By liberating controls system from file-by-file-based system, better able to handle changesets involving many files.

git always under rapid development,graphical interfaces also under speedy construction. See kernel.org git repositories webpage. Can easily browse particular changes, source trees.

Sites (eg. Github) now host literally millions of git repositories, both public/private. Host of easy-to-find articles, books, online tutorials etc., on how to to profitably use git.

##

Back to top