Package Managers

What are package managers?

An easy way to install programs onto your computer and keep them updated

- `apt-get install cowsay`

Common Package Managers

- apt/apt-get
- pacman
- portage
- yum
- pip
- homebrew

Most common command

- apt install <package>
- pacman -S <package>
- emerge <package>
- yum install <package>
- pip install <package>
- brew install <package>

Searching for packages

- apt-cache search <keyword>
- pacman -Ss <keyword>
- emerge -s <regex>
- yum list <package>
- pip search <package>
- brew search / brew info <package>

Updating

- apt update
- pacman -Syu
- emerge -pvuDN world
- yum update
- Pip install <package> --upgrade
- brew upgrade

Removing packages

- apt remove <package>
- pacman -R <package>
- emerge -cav <package>
- yum remove <package>
- pip uninstall <package>
- brew
    - brew tap beeftornado/rmtree
    - brew rmtree <package>

Theory

Dependencies

Original purpose of package managers was to manage dependencies; software programs that other software programs need to be installed in order to run.

These dependencies would ideally form a tree. Actually, though, cycles and conflicts exist.

- `apt-cache depends python`

Bootstrapping

We address the problem of cycles using bootstrapping.

- Minimal system of a few dozen packages that form cycles with each other
  - Libc, gcc, bash, make, etc.
- Capable of providing the dependencies for any other package
- Important security measure to make the system harder to backdoor

Gentoo is an extreme example.

Alternatives

We address conflicts with alternatives. Sometimes more than one software program implements the same interface.

- Vi and Vim are entirely different programs, written by different authors, but typing vi on Andrew will open Vim, because they both implement the same interface.

How is this controlled? What if you actually want Vi? Alternatives.

- `update-alternatives --all`

Atomicity / Transactionality

Package manager actions (update, install, remove, etc.) will either succeed fully or not change the base system at all

- If you turn off your computer during a Windows update, your system may end up in broken state, so Windows updates are not transactional

Transactionality is uncommon because it is difficult to add to a legacy system like Windows or Debian. Some unusual package managers support it (notably Guix and Nix).

System rollback / versioning

Once we have transactionality, we can think of the state of the system as a series of versioned “snapshots”, with the package manager actions acting as a transition function between snapshots.

- Why not save these snapshots and be able to restore to them?

This is easy when your package manager supports transactionality, so, again, only Guix and Nix do this.

Maintainer-controlled vs Author-controlled packages

Some package managers give software program authors control over packaging their own programs. This is most common with language-specific package managers, like `pip` for Python and `npm` for Node.js.

- When authors control their own packages, they are typically more up-to-date
- When maintainers control the packages, they are typically more stable

Package definitions

How are we actually able to `apt-get install` things? What makes this work?

- Package definitions are scripts / programs expressing how to compile and install other programs
- Every time you run `apt-get install`, you are installing a software program compiled according to a package definition
    - With source-based distributions, you actually run the package definition yourself

Source vs Binary distributions

Source refers to source code, while binary refers to the compiled program that you actually run.

- Source distributions have you actually download everything need to compile the program and compile it yourself.
- Binary distributions just send you everything you need to actually run the compiled program
    - This is frequently a lot less

Source-based distributions have a lot of theoretical benefits, but are highly inconvenient in practice.

Reproducible builds

When we choose to use a binary distribution method, how do we ensure that the binary we actually get is correct? What if someone has backdoored it?

- Reproducible builds means that the package definition is specified well-enough that it will always produce the exact same binary, bit-for-bit

This means that we get the same security as source distributions with the convenience of binary distributions.

Reproducible build are difficult, but increasingly popular. Debian, Red Hat, Arch, Guix, Nix, the BSDs, etc. all at least partially support them.

Language-specific package managers

Many programming languages have their own package managers, that are cross-platform.
For example, `pip` for Python and `npm` for Node.js.

- These work very well in practice
- But sometimes programs depend on programs written in other languages

Package managers vs app stores

The primary difference between package managers and app stores is that app stores don’t manage dependencies.

App stores ignore most of the difficult theoretical ideas we’ve just gone over.

But the end user experience is very similar.

Package managers vs containers

A container is a single binary blob containing all of the dependencies of a certain program.

Containers also ignore most of the theoretical issues with package managers. As a result, they are unbelievably ugly in theory.

However, they are very common because writing container definitions is significantly easier than writing package definitions, and because using containers is really nice in practice.

I recommend using containers at internships and jobs, but do be aware that they have significant problems.