CMI - A Simple Yet Powerful Package Manager

by Jason Kaufman

First, Some Guidelines

What is a package manager? Or rather, what should it be? Well, on a very basic level it should be responsible for every file it installs on your system. This responsibility should include the following:

- Every file installed should be logged
- Upon removal, every file logged should be removed

Anything satisfying the above requirements is a package manager, in a sense. However, most so-called package managers fail at accomplishing these two simple tasks. What is even more surprising is the drive to add more esoteric features such as dependency resolution, integrity checking, and built-in updating without making sure that files are installed and removed cleanly. This is where CMI comes in and kicks these sorry excuses for package managers in the teeth.

Why CMI?

<marketing_drivel>
Nine times out of ten, people who develop software aren't the ones who have to package it. For most, as long as it runs on their system or distro no effort (for various reasons) is made to make sure the build, install, and uninstall processes are clean or even functional. Mozilla, for one, does not have a 'make uninstall' routine in any shape or form. Once it's installed, it is near impossible to completely remove.

Some may argue that installing to a path in one's home directory would alleviate that problem, and to some extent, it does. But what if the nice "`--prefix=`" you passed to configure is ignored? Or if the Makefile doesn't care about "`DESTDIR=`"? Who wants to watch out for all this crap every time you build a package? Most certainly, not I. And rather than spend time submitting patches (I know nothing about the inner workings of autotools scripts), I decided to write a sandboxing module that babysits these packages for me.

People who live on the bleeding edge of software releases will love CMI. In the works are "rollback" and "rollfoward" functions for system states. Say you have a stable system and that you might screw it up with an untested release of gtk+2 – all you need to do is tell CMI to take a snapshot of the system and the current state (packages that CMI knows about, and that you've made binary packages of) is saved. From there you can blow up your system (within reason) and rollback to your original state, if needed. Rollforward works in a similar way; in that case you have CMI migrate to another state.
</marketing_drivel>

CMI Overview

CMI is an acronym for **C**onfigure, **M**ake, **I**nstall, and describes the typical function of the package manager. In its current (albeit unstable) state, every effort is made to ensure that every single file, be it text or binary, is accounted for during the build and install stages. That is all I really intend it to do; anything beyond that is a nice-to-have. In turn, CMI is not intended (currently) for use by end users; the focus here is on people who want maximum control and for the package manager to stay out of the way.

What Others Have Done

Before beginning a project or saying, "My app `0\/\/n5 j00`", it is very important to take a critical look at what others have done in the past.

Pros

1. **dpkg**
   - Developers knew enough to abstract apt out of the system
   - Powerful when used in conjunction with apt
2. **Gentoo Portage**
   - Builds from source by default
   - Sandboxing to be implemented soon
3. **pkgsrc**
   - Builds from source by default
   - Simple overall structure, simple tools
   - Supports all or nearly all the arch's that NetBSD runs on
4. **RPM**
   - Supports a wide array of arch's, even those outside of RH's domain

Cons

1. **dpkg**
   - The annoying "debian" subdirectory look familiar to anyone?
   - Packages are non-trivial to create for new software
2. **Gentoo Portage**
   - Enables every freaking thing under the sun by default
   - Enables compiler optimizations in gcc3 that are often more harm than good
3. **pkgsrc**
   - Packages are non-trivial to create for new software
   - Package maintainers may take weeks to get new releases into the tree
4. **RPM**
   - Completely too much overhead and added "features" for a stand-alone PM
   - Source and binaries are not easily extracted from RPMs without external utilities

Project Goals

Here's a start:

1. Provide the basic features of a package manager
2. Give complete control to the user
3. Multiple architecture/OS support
4. Tread lightly on systems (e.g. one directory contains all of CMI)
5. Have a small memory footprint

Development Status

The primary focus of recent development has been to improve the "sandboxing" capabilities of CMI. As a result, it has been completely rewritten three times since June 2002 and is scheduled for another re-write in December 2002. Here's a quick look at what was done in the past, and what's coming up for Version 4:

Version 1 (June 2002)

- Pure Bourne shell code
- Merely scripted builds; no logging of files, etc.
- Became the base of the Bonfirix build system

Version 2 (July 2002)

- Pure Bourne shell code
- Emulated a sandbox by setting the install prefix to a temp directory
- Files installed to temp directory were logged
- Stole .spec concept and command syntax from RPM

Version 3 (October 2002)

- Pure Bourne shell code with external C apps
- Achieved construction of a 500KB sandbox

Version 4 (December 2002, est.)

- Re-implement in C, using standard libs
- Use autotools to ease installation
- Sandboxing for "make install"
- Be able to build and sign binary packages

CMI Internals

A Sample CMI Spec and Defined Fields

```
# begin flac.spec

CMI_NAME  flac
CMI_VERSION 1.0.3
CMI_RELEASE 1
CMI_DATE "$ Date: $"

CMI_CONFIGURE "--prefix=/usr/local"

CMI_DEPS "xmms+"

# end flac.spec
```

And that, my friends, is all that's needed to build flac-1.0.3. You can actually get away with less here; for instance the `CMI_CONFIGURE` field could be axed if CMI's default installation prefix is `/usr/local`, and the `CMI_DATE` is a nice-to-have.

Here's a quick rundown of the defined fields:

`CMI_NAME`: package name
`CMI_VERSION`: package version (major-minor-point)
`CMI_RELEASE`: package revision number
`CMI_DATE`: date string
`CMI_PRE_CONFIGURE`: patch commands, anything that needs to be done first
`CMI_CONFIGURE`: flags to pass to configure
`CMI_MAKE`: either pass flags to make, or substitute with xmkmf & friends
`CMI_INSTALL`: either pass flags to make, or specify your own install routine
`CMI_DEPS`: OK, this requires its own paper

As time progresses (or my memory improves), fields may be added or deleted.

The Sandbox

CMI's sandbox was designed to be a trap of sorts for bad install routines. Based off of uClibc, BusyBox, make, and fileutils, this sandbox can be chroot-ed to. Once inside the sandbox, "make install" can be executed without worry that all the files are contained within.

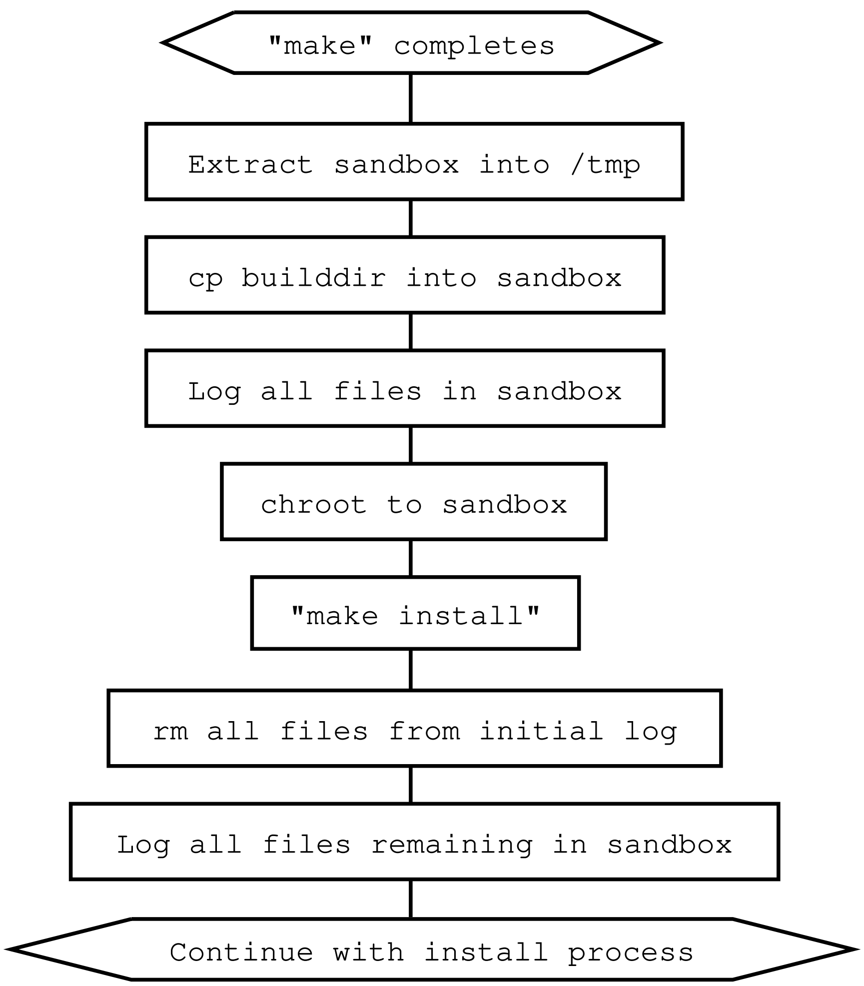

Figure 1: CMI's usage of the sandbox

The sandbox is statically built on the system it runs on, which is why the project will come to rely heavily on the autotools to determine the host architecture and operating system. A few tests have shown the compressed size of the sandbox to be about 200KB, while uncompressed it weighs in at 500KB. Yes, that is even after the binaries have been stripped.

Logging

CMI carries out extensive logging for all package builds. The output of every stage is logged to `$CMI/log/pkg/$CMI_NAME-$CMI_VERSION` in separate files for each stage, usually configure, make, and install.

Directory Structure

A typical CMI install will be self-contained:

```
cmi/log      - log directory, broken into subdirectories according to name
cmi/log/pkg  - package-specific logs
cmi/log/sys  - master logs
cmi/log/snap - snapshots
cmi/sandbox  - where the sandbox is stored/extracted
cmi/build    - build directories
```

Use Cases

Now comes the fun part...what can be done with CMI?. The syntax is currently being rewritten and is not currently defined.

Build

CMI divides all non-query commands into "stages". The build stage is when CMI is supplied with a .spec and told to build the source. The source is then built, everything is logged, and then execution stops. If CMI is told that the source package is actually a binary package, then this stage is ignored. If a package name is supplied, CMI will build a package from the files that exist on the system (outside of CMI's build directories).

Install

CMI executes the sandboxing routine, tar's all the relevant files up, checks the master log for conflicts (overwriting files CMI knows about is a no-no), and then proceeds to install the files into the installation directory. If all goes well, the log collected from the sandbox is added to the master log.

Uninstall

CMI checks the software package's log, and compares the contents to the master log. If there are conflicts, nothing is done by default. This helps CMI to avoid removing files that should not be removed, and reveal possible bugs in CMI.

Query

CMI keeps track of all files installed and which package owns them. Thus queries can be done by package name or file name. Obviously, dependencies are also kept track of.

Snapshot

CMI will copy the contents of the current master log, and a list of all the installed packages into a "snapshot" file. CMI will then verify if a binary package for every installed package exists, and copy all of those and the snapshot file into a timestamp-named directory under `cmi/snapshot`.

Initialize

CMI will make a log of all the files on the system, i.e. ones it is not allowed to overwrite or control.

Manage

CMI will accept a given file or directory and assign it to an existing package. Useful if you make a server config and want CMI to remove it or package it up with the server's binaries later.

Binary Package Structure

Though CMI does not typically require one to build binary packages, the format is given here to show that CMI can handle them. Planned uses of the format include the "rollback" and "rollforward" actions.

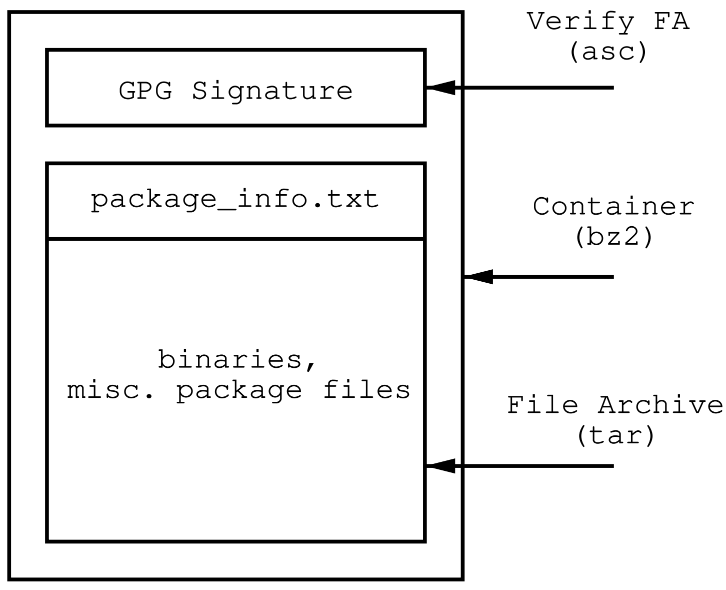

Figure 2: Generic CMI binary format

Notice that the structure is defined by pre-existing and commonplace archiving formats, such as tar and bzip. The ASCII-armored GPG signature is kept outside the internal tar so CMI (or anyone else) can verify the tar. "`package_info.txt`" may contain further integrity information, such as md5sums for each file. It will at least contain version information, the installation prefix, some authoring details, build details, and of course, the OS and architecture it was built for. A sample `package_info.txt` is seen below:

```
# begin package_info.txt

CMI_NAME=flac
CMI_VERSION=1.0.3
CMI_RELEASE=1
CMI_ARCH=ppc
CMI_OS=bfx
CMI_INSTALL_PFX="/usr/local"
CMI_CONFIG_OPTS="--prefix=/usr/local"
CMI_BIN_DEPS="xmms"
CMI_AUTHOR="Bonfire <bonfire@bonfirix.org>"
CMI_DATE="$ Date: $"

CMI_MD5SUMS
4d111ae04d20389f8835709a3b03ae60  bin/flac

# end package_info.txt
```

References

Package Managers

dpkg http://debian.org/doc/maint-guide/

Pacman http://physics.bu.edu/ỹoussef/pacman/

pkgsrc http://netbsd.org/Documentation/software/packages.html

RPM http://rpm.org/

Zoularis http://netbsd.org/zoularis/

Utilities

BusyBox http://busybox.net/

uClibc http://uclibc.org/