amkecpak Documentation

Release 0.0.0

Pierre-Francois Carpentier

2020-07-01

Contents

CHAPTER 1

Requirements

This section details the tools needed by this packaging framework.

Warning: Only the tools necessary for building packages are detailed here. Each individual package requires it's own build dependencies (gcc, cmake...). You must either install those dependencies or build in chroot (make *rpm_chroot* or *deb_chroot*).

1.1 Debian/Ubuntu

To install the Debian requirements:

```
# Debian/Ubuntu (deb)
$ apt-get install make debhelper reprepro cowbuilder wget
```

Debian/Ubuntu also distributes rpm packaging tools. Here are the additionnal packages to install if you want to produce rpms on a Debian based system:

```
# Debian/Ubuntu (rpm)
$ apt-get install createrepo rpm mock expect
```

1.2 RHEL/CentOS/Fedora

To install the RHEL requirements:

```
# CentOS/RHEL (rpm)
$ yum install rpm-sign expect rpm-build createrepo make mock wget

# Fedora (rpm)
$ dnf install rpm-sign expect rpm-build createrepo make mock wget
```

Note: CentOS/RHEL doesn't distribute the .deb tooling. Contrary to Debian/Ubuntu, it's not possible to produce .deb on a CentOS/RHEL.

CHAPTER 2

Cheatsheet

2.1 Scripts

Initialize a package:

```
$ ./common/init_pkg.sh -n <PKG_NAME>
```

2.2 All Makefiles

Display the help:

```
$ make help
```

Listing the the distribution versions available:

```
$ make list_dist
```

Cleaning:

```
# clean everything
$ make clean

# clean everything except upstream downloads
$ make clean KEEP_CACHE=true
```

2.3 Package Makefile

(Re)creating the MANIFEST file:

```
$ make manifest
```

Building .deb:

```
$ make deb
```

Building .rpm:

```
$ make rpm
```

Building .deb (chroot):

```
$ make deb_chroot DIST=<VERSION>
```

Building .rpm (chroot):

```
$ make rpm_chroot DIST=<VERSION>
```

2.4 Global Makefile

Building all .deb:

```
$ make deb
```

Building all .rpm:

```
$ make rpm
```

Building all.deb (chroot):

```
$ make deb_chroot DIST=<VERSION>
```

Building all .rpm (chroot):

```
$ make rpm_chroot DIST=<VERSION>
```

Building the .deb repository:

```
$ make deb_repo DIST<VERSION>
```

Building the .rpm repository:

```
$ make rpm_repo DIST<VERSION>
```

CHAPTER 3

Create a package

Creating a package **foo** involves the following steps:

- Initialize the packaging directories for **foo**.
- Fill **foo/Makefile** (used for metadata and upstream source recovery/preparation).
- Generate the **MANIFEST** file (checksum of upstream source)
- Do distribution specific packaging (dependencies in **debian/control**, **rpm/component.spec**, pre/post install scripts, init scripts, etc)
- Build the package

3.1 Initialize package skeleton

To create a package "**foo**" skeleton, run the follwing command:

```
$ ./common/init_pkg.sh -n foo
```

This script will create the following subtree:

```
tree foo/
foo
├── buildenv -> ../common/buildenv
├── debian
│   ├── changelog
│   ├── compat
│   ├── conffiles
│   ├── control
│   ├── copyright
│   ├── foo.cron.d.ex
│   ├── foo.default.ex
│   ├── init.d.ex
│   ├── postinst.ex
```

(continues on next page)

(continued from previous page)

```
    ├── postrm.ex
    ├── preinst.ex
    ├── prerm.ex
    ├── rules
    └── source
        └── format
├── rpm
    └── component.spec
├── Makefile
└── MANIFEST
```

This tree contains two main directories, two main files, and a symlink:

- **Makefile**: used to download and prepare upstream sources.
- **MANIFEST**: listing of the downloaded files and their hash.
- **debian**: deb packaging stuff.
- **rpm**: rpm packaging stuff (**component.spec** and optionally additional content like **.service** files).
- **buildenv**: symlink to the shared build resources (Makefile.common, and various helper scripts).

Note: Don't rename component.spec, build script for rpm expect this file to exist.

At this point, with default content, "empty" .rpm and .deb packages can be built:

```
$ cd foo/                                       # go in dir
# make help                                     # display Makefile help
$ make deb                                      # build deb
$ make rpm                                      # build rpm
$ dpkg -c out/foo_0.0.1-1\~up+deb00_all.deb # look .deb content
$ rpm -qpl out/foo-0.0.1-1.00.noarch.rpm    # look .rpm content
```

3.2 Package metadata

It's necessary to setup the package metadata (version, description) to their proper values. Package metadata are declared at the top of the package Makefile:

```
# Version
# if possible, keep the upstream version
VERSION=0.0.1

# Revision number
# increment it when fixing packaging for a given release
# reset it to 1 if VERSION is increased
RELEASE=1

# URL of the upstream project
URL=http://example.org/stuff

# short summary of what the package provides
SUMMARY=My package summary
```

(continues on next page)

(continued from previous page)

```
# long version of the summary, (but I could be lazy)
DESCRIPTION=$(SUMMARY)
```

Note: During the package build, these variables are automatically substitute in packaging files. This is done by simple running sed -s 's|@VAR@|$(VAR)|' on these files.

Don't remove the @VAR@ (ex: @SUMMARY@, @URL@, @VERSION@) in the packaging files.

3.3 Download upstream sources

This packaging infrastructure comes with a small tool, **./common/buildenv/wget_sum.sh** to handle downloads.

This tool role is:

- Download upstream sources.
- Check the integrity of the upstream source against the *MANIFEST* file (sha512 sum).
- (Re)Build the *MANIFEST* file if requested.
- Handle a local download cache to avoid downloading sources for each build.

3.3.1 Download tool usage

Inside the Makefile, use it as followed:

```
$(WGS) -u <url> -o $(BUILD_DIR)/<output file>
```

Example:

```
# Name of the package
NAME = libemf2svg

# Version
VERSION = 1.0.1

# URL of the project
URL=https://github.com/kakwa/libemf2svg

# Source recovery url
URL_SRC=$(URL)/archive/$(VERSION).tar.gz

# Including common rules and targets
include buildenv/Makefile.common

$(SOURCE_ARCHIVE): $(SOURCE_DIR) $(CACHE) Makefile MANIFEST
        $(WGS) -u $(URL_SRC) -o $(SOURCE_ARCHIVE)
```

Note: Please note the templatization of the download url "$(URL_SRC)". Specifically the "$(VERSION)" part. This way, when a new upstream version is available, simply updating the "VERSION" variable and updating the manifest is necessary if upstream has not changed drastically.

3.3.2 Build MANIFEST file

To create or update the MANIFEST file, just run the following command:

```
make manifest
```

Note: In case of checksum error, an error like the following one will be displayed:

```
[ERROR] Bad checksum for 'https://github.com/kakwa/mk-sh-skel/archive/1.0.0.tar.gz'
expected:␣
↪2cdeaa0cd4ddf624b5bc7ka5dbdeb4c3dbe77df09eb58bac7621ee7b64868e0d916a1318e4d13e1ee8f50d470d58dd285ec
got:    ␣
↪1cdea044ddf624b5bc7465dbdeb4c3dbe77df09eb58bac7621ee7b64868e0d916a1318e4d13e1ee8f50d470d58dd285ed5
Makefile:38: recipe for target 'builddir/mk-sh-skel_1.0.0.orig.tar.gz' failed
make: *** [builddir/mk-sh-skel_1.0.0.orig.tar.gz] Error 1
```

If it happens, either it's a "legitimate" mismatch (because you have changed the version for example), and you should rebuild the MANIFEST file.

Or it's upstream doing weird things like re-releasing reusing the same version number which is generally bad practice and should be investigated.

3.3.3 Source preparation

The source preparation is made in the **$(SOURCE_ARCHIVE)** target.

The goal of this rule is to create the **tar.gz** archive **$(SOURCE_ARCHIVE)**.

The root directory of the source archive should be **$(NAME)-$(VERSION)**. For example:

```
tar -tvf cache/mk-sh-skel_1.0.0.orig.tar.gz
drwxrwxr-x root/root         0 2015-11-27 00:26 mk-sh-skel-1.0.0/
-rw-rw-r-- root/root      1135 2015-11-27 00:26 mk-sh-skel-1.0.0/LICENSE
-rw-rw-r-- root/root       145 2015-11-27 00:26 mk-sh-skel-1.0.0/Makefile
-rw-rw-r-- root/root       972 2015-11-27 00:26 mk-sh-skel-1.0.0/README.md
-rw-rw-r-- root/root      1037 2015-11-27 00:26 mk-sh-skel-1.0.0/mksh-skel
```

In ideal cases, it's only a matter of downloading the upstream sources as these conventions are quite standards. For example:

```
# Version
VERSION = 1.0.1

# URL of the project
URL=https://github.com/kakwa/mk-sh-skel

# example of source recovery url
URL_SRC=$(URL)/archive/$(VERSION).tar.gz

# Basic source archive recovery,
# this works fine if upstream is clean
$(SOURCE_ARCHIVE): $(SOURCE_DIR) $(CACHE) Makefile MANIFEST
        $(WGS) -u $(URL_SRC) -o $(SOURCE_ARCHIVE)
```

But in some cases, it might be necessary to modify the upstream sources content.

For that two helper variables are provided:

- **$(SOURCE_DIR)**: source directory (with proper naming convention) where to put sources before building the source archive.
- **$(SOURCE_TAR_CMD)**: once **$(SOURCE_DIR)** is filled with content, just call this variable, it will generate the **$(SOURCE_ARCHIVE)** tar.gz and do some cleanup. If present, **$(SOURCE_TAR_CMD)** should be the last step in **$(SOURCE_ARCHIVE)** target.

For example:

```
# Version
VERSION = 1.0.7

# URL of the project
URL=http://repos.entrouvert.org/python-rfc3161.git

# example of source recovery url
URL_SRC=$(URL)/snapshot/python-rfc3161-$(VERSION).tar.gz

# preparation of the sources with removal of upstream, unwanted debian/ packaging
# it does the following:
# * recover upstream archive
# * uncompress it
# * upstream modification (remove the unwanted debian/ dir from upstream source)
# * move remaining stuff to $(SOURCE_DIR)
# * do some cleanup
# * build the archive

$(SOURCE_ARCHIVE): $(SOURCE_DIR) $(CACHE) Makefile MANIFEST
        $(WGS) -u $(URL_SRC) -o $(BUILD_DIR)/python-rfc3161-$(VERSION).tar.gz
        mkdir -p $(BUILD_DIR)/tmp
        tar -vxf $(BUILD_DIR)/$(NAME)-$(VERSION).tar.gz -C $(BUILD_DIR)/tmp
        rm -rf $(BUILD_DIR)/tmp/python-rfc3161-$(VERSION)/debian
        mv $(BUILD_DIR)/tmp/python-rfc3161-$(VERSION)/* $(SOURCE_DIR)
        rm -rf $(BUILD_DIR)/tmp
        rm -f $(BUILD_DIR)/python-rfc3161-$(VERSION).tar.gz
        $(SOURCE_TAR_CMD)
```

CHAPTER 4

Skipping distribution versions

Sometimes, packages cannot be built on certain versions of specific distribution.

This typically happens when the dependencies are too old or not present in older versions.

In such cases, it's possible to skip the build the package for specific distribution versions.

For that, you need to set the **SKIP** variable.

```
# here, we skip build on Debian older than 9, RHEL older than 7, Fedora older than 30␣
↪and Ubuntu older than 18.4
SKIP=<:deb:9 <:el:7 <:fc:30 <:ubu:18.4
```

SKIP contains a space separated list of rules.

each rule have the format **<op>:<dist>:<version>**, with:

- **<op>**: the operation (must be '>', '>=', '<', '<=' or '=')
- **<dist>**: the distribution code name (examples: 'deb', 'el', 'fc')
- **<version>**: the version number to compare with

4.1 Distribution specific packaging

For the most part, just package according to deb/rpm documentation, filling the **rpm/component.spec**, **debian/rules**, **debian/control**, and any other packaging files if necessary.

Note: I would advise you to try to respect the distributions guidelines and standards such as the FHS (https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard).

4.1.1 deb

For Debian packages, just leverage the usual packaging patterns such as the **PKG.init**, **PKG.default**, **PKG.service**, (...) files and use the **override_dh_*** targets in **debian/rules** if necessary. Finally, add your dependencies and architecture(s) in the **debian/control** file.

Note: In many cases, with clean upstreams, there is nearly nothing to do except setting dependencies and architecture, the various dh_helpers will do their magic and build a clean package.

If you are unlucky, uncomment the **export DH_VERBOSE=1** in **debian/rules** and customize the build as necessary using the **override_dh_*** targets.

4.1.2 rpm

For rpm, fill the various sections of the **rpm/component.spec** file such as **BuildRequires:**, **Requires:** or **BuildArch:** parameters and the various sections like **%install**.

If additional files a required for packaging, an init script for example, put these files in the **rpm/** directory.

All additional files in the **rpm/** directory are copied in the rpmbuild **SOURCES** directory. This means that it's possible to treat them as additional source files in **component.spec** with the **Source[0-9]:** directives.

Example for ldapcherry.service systemd service file and it's associated files:

```
# rpm/ directory content
tree rpm/
rpm/
├── component.spec
├── ldapcherry
├── ldapcherry.conf
└── ldapcherry.service
```

```
# component.spec relevant sections
Source: %{pkgname}-%{version}.tar.gz
Source1: ldapcherry
Source2: ldapcherry.conf
Source3: ldapcherry.service

# install section
%install

# install the .service, the sysconfig file and tmpfiles.d (for pid file creation as␣
↪non-root user)
mkdir -p %{buildroot}%{_unitdir}
mkdir -p %{buildroot}/usr/lib/tmpfiles.d/
mkdir -p %{buildroot}/etc/sysconfig/
install -pm644 %{SOURCE1} %{buildroot}/etc/sysconfig/
install -pm644 %{SOURCE2} %{buildroot}/usr/lib/tmpfiles.d/
install -pm644 %{SOURCE3} %{buildroot}%{_unitdir}
```

4.1.3 Version specific packaging files

Depending on the OS version targeted, there might be some differences in packaging. A common difference is the dependency names.

For handling those cases, the present packaging framework provides a simple mechanism.

To override any file **<FILE>** in either the **rpm/** or **debian/** directories for a specific distribution version **<DIST>**, create a file **<FILE>.dist.<DIST>** with the specific content for version **<DIST>**.

For example, with the **debian/control** file and distribution **jessie**:

```
debian/control              # will be used as default
debian/control.dist.jessie # will be used if build is called with DIST=jessie
```

It also permits to handle additional files for specific distribution versions.

CHAPTER 5

Build a package

Here is the general building workflow:

- The steps in orange are common for all packages and must not be modified.
- The steps in green are package specific, it's those steps which must be customized for each package.

5.1 Cleaning cache, out and src-out directories

Example with with package python-asciigraph:

```
# go inside the component directory
$ cd python-asciigraph

# clean everything
$ make clean

# clean, but keep upstream sources to avoid re-downloading them
$ make clean KEEP_CACHE=true
```

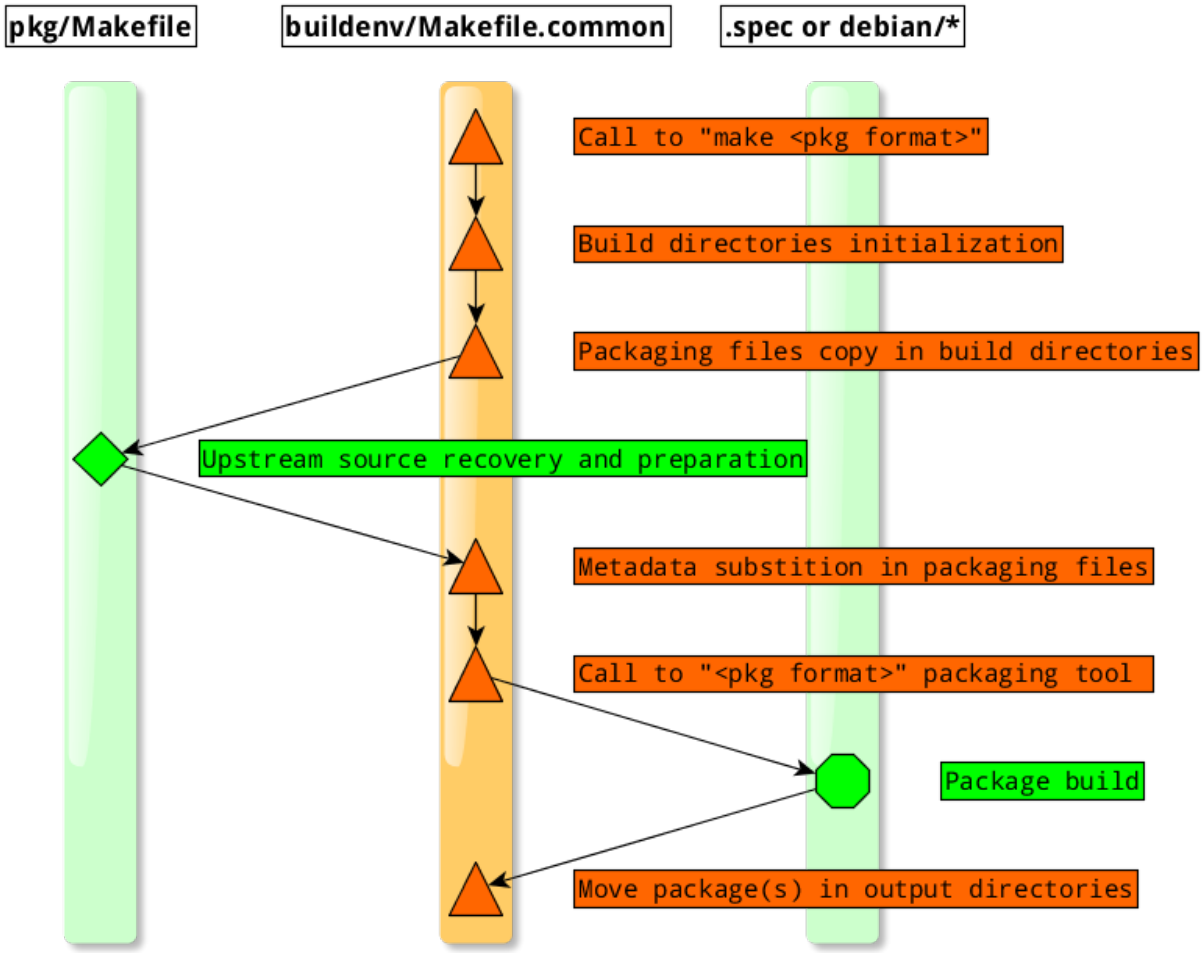
5.2 Build rpm package

Example with package python-asciigraph:

```
# go inside the component directory
$ cd python-asciigraph

# build rpm package
$ make rpm
```

Here are the results:


pkg/Makefile
buildenv/Makefile.common
.spec or debian/*
Call to "make <pkg format>"
Build directories initialization
Packaging files copy in build directories
Upstream source recovery and preparation
Metadata substition in packaging files
Call to "<pkg format>" packaging tool
Package build
Move package(s) in output directories

```
# output packages:
$ ls out/
python-asciigraph-1.1.3-1.kw+unk0.noarch.rpm

# output source package
$ ls src-out/
python-asciigraph-1.1.3-1.kw+unk0.src.rpm
```

5.3 Build rpm inside a clean chroot

```
# go inside the component directory
$ cd python-asciigraph

# Build the rpm in a chroot (using mockchain)
# Replace el7 by the dist version targeted
$ make rpm_chroot DIST=el7
```

5.4 Build deb package

Example with package python-asciigraph:

```
# go inside the component directory
$ cd python-asciigraph

# build deb package
$ make deb
```

Here are the results:

```
# output packages:
$ ls out/
python-asciigraph_1.1.3-1~kw+unk0_all.deb

# output source package
$ ls src-out/
python-asciigraph_1.1.3-1~kw+unk0.debian.tar.xz  python-asciigraph_1.1.3-1~kw+unk0.dsc
python-asciigraph_1.1.3.orig.tar.gz
```

5.5 Build deb package inside a clean chroot

This build system can leverage cowbuilder from Debian to build in a clean chroot.

This is the recommended way to build packages targeted to be used in production.

Building in chroot is heavier but has multiple gains:

- It permits to build in a clean environment every time
- It rapidly exits in error if the build dependencies are not properly declared
- It permits to target different version of Debian (stretch, jessie, wheezy)

- It manages build dependencies, installing them automatically (if properly declared)
- It permits to avoid having to install all build dependencies on your main system

```
# go inside the component directory
$ cd python-asciigraph

# build deb package for dist jessie
$ make deb_chroot DIST=jessie
```

5.6 Chroot building tips

5.6.1 Common tips

Note: Building the chroot can be a long and heavy step but there are several way to accelerate it.

The first is to used a local mirror.

For deb/cowbuilder this can be done using the **DEB_MIRROR** option when calling deb_chroot:

```
$ make deb_chroot DIST=jessie DEB_MIRROR=http://your.local.mirror/debian
```

For rpm/mock, this can be done by changing the appropriate configuration file in **/etc/mock**

```
$ vim /etc/mock/epel-7-x86_64.cfg
```

The second is to use a tmpfs for building, it requires a few GB of RAM however (at least 2GB per distro version targeted, but this may vary depending on the number packages and the size of their dependencies):

For deb/cowbuilder:

```
# as root
$ mount -t tmpfs -o size=16G tmpfs /var/cache/pbuilder/
```

```
# in fstab
tmpfs /var/cache/pbuilder/ tmpfs defaults,size=16G 0 0
```

For rpm/mock:

```
# as root
$ mount -t tmpfs -o size=16G tmpfs /var/lib/mock
```

```
# in fstab
tmpfs /var/lib/mock tmpfs defaults,size=16G 0 0
```

Warning: Some recent distributions may disable the **vsyscall** syscall which is used by older libc (ex: CentOS/RHEL <= 6).

The problem can be diagnosed by running **dmesg** after a failure to create or run anything in the chroot. You would get errors like:

```
[  578.456176] sh[15402]: vsyscall attempted with vsyscall=none ip:ffffffffff600400␣
↪cs:33 sp:7ffd469c5aa8 ax:ffffffffff600400 si:7ffd469c6f23 di:0
[  578.456180] sh[15402]: segfault at ffffffffff600400 ip ffffffffff600400 sp␣
↪00007ffd469c5aa8 error 15
```

In most cases this syscall can be reenabled with **vsyscall=emulate** option in the kernel command line.

5.6.2 Deb/cowbuilder tips

Warning: Building in chroot requires root permission (it's necessary for creating the chroot environment).

If **make deb_chroot** is run as a standard user, **sudo** will be used for cowbuilder calls.

If you want to avoid password promt, the only command that needs to be white listed in sudoers configuration is **cowbuilder**:

```
# replace build-user with the user used to generate the packages
build-user ALL=(ALL) NOPASSWD: /usr/sbin/cowbuilder
```

Warning: If there is an issue or when modifying the chroot (changing the mirror used for example), it may be necessary to removing an existing cowbuilder chroot.

For that, use the **deb_get_chroot_path** target:

```
# show the chroot path:
make deb_get_chroot_path DIST=<code name>

# as root
# remove the chroot
rm -rf `make deb_get_chroot_path DIST=<code name>`
```

Warning: To create the cowbuilder chroot, it's required to have the GPG keys of the targeted DIST.

If you get errors like:

```
I: Checking Release signature
E: Release signed by unknown key (key id EF0F382A1A7B6500)
E: debootstrap failed
```

it means that you don't have the required keys.

The debian-archive-keyring and ubuntu-archive-keyring packages provides these keys. However it might be necessary to use a newer keyring than the one available in your environment, specially if crossing from an old Ubuntu to a new Debian or an old Debian to a new Ubuntu.

For example, with Ubuntu Trusty (14.04), targeting Debian stretch, the following hack is necessary:

```
wget http://cz.archive.ubuntu.com/ubuntu/pool/universe/d/debian-archive-keyring/
↪debian-archive-keyring_2017.5_all.deb \
        && sudo dpkg -i debian-archive-keyring_2017.5_all.deb

ls /etc/apt/trusted.gpg.d/
debian-archive-jessie-automatic.gpg           debian-archive-stretch-security-
↪automatic.gpg
debian-archive-jessie-security-automatic.gpg  debian-archive-stretch-stable.gpg
debian-archive-jessie-stable.gpg              debian-archive-wheezy-automatic.gpg
debian-archive-stretch-automatic.gpg          debian-archive-wheezy-stable.gpg
```

It might also be necessary to pass additionnal parameters to make cowbuilder use this keyring:

```
make deb_chroot DIST=stretch COW_OPTS=--debootstrapopts=--keyring=/etc/apt/trusted.
↪gpg.d/debian-archive-stretch-stable.gpg
```

5.6.3 Rpm/mock tips

Warning: To get the necessary permission to build a package using mock, a **mock** group must be present on the system.

And the user building packages must belong to this group.

```
# Replace USER_ID by the build user
groupadd mock
usermod -a -G mock USER_ID
```

Warning: In Debian/Ubuntu, mockchain may fail because **/usr/bin/createrepo_c** is not available, (Debian bug #875701).

A work around is to install the package **createrepo** and symlink **/usr/bin/createrepo_c** to **/usr/bin/createrepo**.

Also, the dependency **python3-requests** is missing, it's necessary to install this package manually.

```
apt-get install createrepo python3-requests
ln -s /usr/bin/createrepo /usr/bin/createrepo_c
```

CHAPTER 6

Build the complete repositories

6.1 Repository metadata

The repository has a few parameters which must be filled in **common/buildenv/Makefile.config**:

```
# Name of the maintainer
MAINTAINER := Name of the Maintainer

# Email of the maintainer
MAINTAINER_EMAIL := somebody@example.com

# Origin ID (2 or 3 letters ID of origin)
PKG_ORG := or

# Origin of the packages (full name)
PKG_ORIGIN := organization

# The gpg key used to sign the packages
GPG_KEY := GPG_SIGNKEY

# repo component (main/contrib/non-free/universe/etc)
DEB_REPO_COMPONENT := main

# Definition of the debian repository configuration
# "Codename: $(DIST)" and "Components: $(DEB_REPO_COMPONENT)"
# should not be modified.
define DEB_REPO_CONFIG
Origin: $(PKG_ORIGIN)
Label: $(PKG_ORIGIN)
Suite: $(DIST)
Codename: $(DIST)
Version: 3.1
Architectures: $(shell dpkg --print-architecture)
Components: $(DEB_REPO_COMPONENT)
```

(continues on next page)

(continued from previous page)

```
Description: Repository containing misc packages
SignWith: $(GPG_KEY)
endef

export DEB_REPO_CONFIG
```

6.2 Clean before build

Optionally, it's possible to clean everything before building:

```
# use KEEP_CACHE=true if you want to keep cached upstream sources
$ make clean
```

6.3 Create the repositories

Note: use -j <number of jobs> to run multiple packaging jobs in parallele

Note: use **ERROR=skip** to ignore package build failures when calling **make <pkg>_repo** and keep continuing building the repo.

Note: this framework handles build dependencies needing themself to be built.

This is done by a simple retry loop: if packages fails to build, retry with the freeshly generated packages available as installable dependencies.

This should converge to all packages being built. It will stop in error if an iterration doesn't manage to build any new package.

6.3.1 Build .deb repository

To build the .deb repository, run:

```
# Creating the deb repository (using chroots)
# Replace stretch by the distro code name targeted
$ make deb_repo -j 4 DIST=stretch

# Building without chroot, directly on the host
$ make deb_repo -j 4 DIST=stretch NOCHROOT=true

# same ignoring individual package build errors
$ make deb_repo -j 4 DIST=stretch NOCHROOT=true SKIP=true
```

Note: **deb_repo** target supports the same variables as the **deb_chroot** target, like for example **DEB_MIRROR**

Result:

```
out
├── deb.jessie
│   ├── conf
│   │   └── distributions
│   ├── db
│   │   ├── checksums.db
│   │   ├── contents.cache.db
│   │   ├── packages.db
│   │   ├── references.db
│   │   ├── release.caches.db
│   │   └── version
│   ├── dists
│   │   └── jessie
│   │       ├── InRelease
│   │       ├── main
│   │       │   └── binary-amd64
│   │       │       ├── Packages
│   │       │       ├── Packages.gz
│   │       │       └── Release
│   │       ├── Release
│   │       └── Release.gpg
│   ├── pool
│   │   └── main
│   │       ├── c
│   │       │   └── civetweb
│   │       │       ├── civetweb_1.9.1.9999-2~kw+deb8_amd64.deb
│   │       │       ├── libcivetweb_1.9.1.9999-2~kw+deb8_amd64.deb
│   │       │       └── libcivetweb-dev_1.9.1.9999-2~kw+deb8_all.deb
│   │       ├── p
│   │       │   ├── pixiecore
│   │       │   │   └── pixiecore_0.0~2016.02.29-1~kw+deb8_amd64.deb
│   │       │   ├── python-asciigraph
│   │       │   │   └── python-asciigraph_1.1.3-1~kw+deb8_all.deb
│   │       │   ├── python-pygraph-redis
│   │       │   │   └── python-pygraph-redis_0.2.1-1~kw+deb8_all.deb
│   │       │   └── python-rfc3161
│   │       │       └── python-rfc3161_1.0.7-1~kw+deb8_all.deb
│   │       └── u
│   │           └── uts-server
│   │               └── uts-server_0.1.9-1~kw+deb8_amd64.deb
│   └── raw
│       ├── civetweb_1.9.1.9999-2~kw+deb8_amd64.deb
│       ├── Packages
│       ├── pixiecore_0.0~2016.02.29-1~kw+deb8_amd64.deb
│       ├── python-asciigraph_1.1.3-1~kw+deb8_all.deb
│       ├── python-pygraph-redis_0.2.1-1~kw+deb8_all.deb
│       ├── python-rfc3161_1.0.7-1~kw+deb8_all.deb
│       ├── uts-server_0.1.9-1~kw+deb8_amd64.deb
│       └── uts-server-dbgsym_0.1.9-1~kw+deb8_amd64.deb
└── GPG-KEY.pub
```

6.3.2 Build the rpm repository

To build the .rpm repository, run:

```
# Create the rpm repository
# Replace el7 by the distro code name targeted
$ make rpm_repo DIST=el7
```

Warning: mock doesn't support building 2 packages in parallele, don't use -j N with N > 1.

The resulting repositories will look like that:

```
out
├── GPG-KEY.pub
└── rpm.el7
    ├── 7
    │   └── x86_64
    │       ├── civetweb-1.9.1.9999-3.el7.centos.x86_64.rpm
    │       ├── dnscherry-0.1.3-1.el7.centos.noarch.rpm
    │       ├── python-asciigraph-1.1.3-1.el7.centos.noarch.rpm
    │       ├── python-pygraph-redis-0.2.1-1.el7.centos.noarch.rpm
    │       ├── python-rfc3161-1.0.7-1.el7.centos.noarch.rpm
    │       ├── repodata
    │       │   ├── 279156abfa1a5611056b66b7b6481e531977699ee9b5b06462fc58848408cb88-
↪filelists.xml.gz
    │       │   ├── 3221e073b2d2d0a4176d591db070b479975e1341336a96e1c3507366743e4969-
↪other.sqlite.bz2
    │       │   ├── a718d20219a56321fb7c981944d671a6ab79379f064388a5bad4ec9f0d2e6b39-
↪other.xml.gz
    │       │   ├── ab2d5c7943cb6fea596116dc841be8da02f5057903b8e4314de9f302cd20e59f-
↪primary.xml.gz
    │       │   ├── ec4c86e1cf1c6e36c8020b650066db23c112f2357803eb8dfc327aff8197e2c2-
↪filelists.sqlite.bz2
    │       │   ├── fbc3d4f1d6831239ca0a138e24dcbb6ed5b08f5521bae5c5c41d0e46f56e34b2-
↪primary.sqlite.bz2
    │       │   └── repomd.xml
    │       └── uts-server-0.1.9-1.el7.centos.x86_64.rpm
    └── raw
        ├── configs
        │   └── epel-7-x86_64
        │       ├── epel-7-x86_64.cfg
        │       ├── logging.ini
        │       └── site-defaults.cfg
        └── results
            └── epel-7-x86_64
                ├── civetweb-1.9.1.9999-3.kw+el7
                │   ├── build.log
                │   ├── civetweb-1.9.1.9999-3.el7.centos.src.rpm
                │   ├── civetweb-1.9.1.9999-3.el7.centos.x86_64.rpm
                │   ├── libcivetweb-1.9.1.9999-3.el7.centos.x86_64.rpm
                │   ├── libcivetweb-devel-1.9.1.9999-3.el7.centos.x86_64.rpm
                │   ├── root.log
                │   ├── state.log
                │   └── success
                ├── python-asciigraph-1.1.3-1.kw+el7
                │   ├── build.log
                │   ├── python-asciigraph-1.1.3-1.el7.centos.noarch.rpm
                │   ├── python-asciigraph-1.1.3-1.el7.centos.src.rpm
                │   ├── root.log
```

(continues on next page)

(continued from previous page)

```
                    ├── state.log
                    └── success
                ├── python-pygraph-redis-0.2.1-1.kw+el7
                    ├── build.log
                    ├── python-pygraph-redis-0.2.1-1.el7.centos.noarch.rpm
                    ├── python-pygraph-redis-0.2.1-1.el7.centos.src.rpm
                    ├── root.log
                    ├── state.log
                    └── success
                ├── python-rfc3161-1.0.7-1.kw+el7
                    ├── build.log
                    ├── python-rfc3161-1.0.7-1.el7.centos.noarch.rpm
                    ├── python-rfc3161-1.0.7-1.el7.centos.src.rpm
                    ├── root.log
                    ├── state.log
                    └── success
                ├── repodata
                    ├──␣
↪21289fde781204ae80d8b5ccb6409f15298f3d131c5d9b6c83a559023d66117b-primary.sqlite.bz2
                    ├──␣
↪4a058e025303cbaa07d36b869cfa275d3c34eb8d8ce03b973544f449185b6971-primary.xml.gz
                    ├──␣
↪64b282083531afc79e552b977c591e381cded3ace188e35a12f922b6f63e9bd3-filelists.xml.gz
                    ├──␣
↪9efa0da5d3d74511206cfba49adca7179bcd13bf9e48bb39582cce9e4ccdc1a6-filelists.sqlite.
↪bz2
                    ├──␣
↪bb3775aea68d9c6de66a7d466d0af65f0e19a4d0e036cf6df341f2d7f56c16dd-other.xml.gz
                    ├──␣
↪c22678232ad81e43413ccc4e5ac8a3966b7ea70697b499b35dd4c9ad457386fa-other.sqlite.bz2
                    ├── filelists.xml.gz
                    ├── other.xml.gz
                    └── repomd.xml
                └── uts-server-0.1.9-1.kw+el7
                    ├── build.log
                    ├── root.log
                    ├── state.log
                    ├── success
                    ├── uts-server-0.1.9-1.el7.centos.src.rpm
                    └── uts-server-0.1.9-1.el7.centos.x86_64.rpm
```

CHAPTER 7

Misc documentation

7.1 GPG cheat sheet

Package are signed by a gpg key.

Here are some useful commands to manage this key:

Generate the GPG key:

```
$ gpg --gen-key
```

List the keys:

```
$ gpg -K
```

Export the private key (multiple hosts):

```
$ gpg --export-secret-key -a "kakwa" > priv.gpg
```

Import the private key:

```
$ gpg --import priv.gpg
```

import the key in debian:

```
$ cat pub.gpg | apt-key add -
```

CHAPTER 8

Motivation

I found that packaging .deb or .rpm is a little annoying for various reasons:

- Remembering all the options of rpmbuild/dpkg-buildpackage/mock/cowbuilder is a pain.
- The rpm and deb tools don't follow the same patterns, it's hard to switch between one and the other.
- The upstream source recovery is a painful story.
- Easy updates are kind of difficult, specially with several distributions and distribution version to manage.
- Ordering the packages builds (ex: build this lib(s) before the final binary) is painful.
- There is no "end to end" build command (From "simple check out of some packaging file" to "complete repository generated")

To solve these issues, I've created a set of Makefiles with easy to remember targets like **make rpm** or **make deb**. These targets are chained in bigger targets like **make deb_repo** or **make rpm_repo** to build a complete package repository. Using Makefiles also helps for parallel builds, the error handling, and delta (re)builds. I've also created a few scripts to help with things like upstream source recovery, package initialization, or determining OS version.

I've also stolen some patterns I really liked from Gentoo and its .ebuild files:

- **Templatize upstream URLs**: Most upstream normalize how they ship sources. It's generally a tar.gz file with a fixed pattern like **<NAME>-<VERSION>.tar.gz**, it's quite easy to "templatize" upstream download URLs with a **$(VERSION)** variable. Doing so makes updates easier, just change one variable declaration and it's done.
- **MANIFEST files**: Another thing I liked was that they keep a **manifest** file with the **checksum** of upstream sources. It permits to have a safe guard against modification of an existing upstream release, gaining some basic guaranties about build reproductibility and avoiding surprises...
- **Keeping upstream recovery and build distinct**: Also the recovery of the upstream sources in Gentoo .ebuild files is a clearly distinct step from the build and install steps, no call to pip or wget in the middle of a compilation.

However, don't expect this framework to magically package anything for you. Appart from a few metadata substitutions like version, license, packager's email or description, this framework keeps all the knobs of regular .deb or .rpm packaging.

CHAPTER 9

Name

amkecpak is just an anagram of makepack (for Make Package).

CHAPTER 10

Resources

Amkecpak, a makefile based packaging framework.

Doc Documentation on ReadTheDoc

Dev GitHub

10.1 Packaging documentation in a nutshell

```
# Install the packaing tools
$ apt-get install make debhelper reprepro cowbuilder wget
# or
$ yum install make rpm-sign expect rpm-build createrepo mock wget

# Init a package foo
$ ./common/init_pkg.sh -n foo

$ cd foo/

# Implementing the package
$ vim Makefile
$ make manifest
$ vim debian/rules ; vim debian/control
$ vim rpm/component.spec
```

(continues on next page)

(continued from previous page)

```
# Help for the various targets
$ make help

# Building the packages
$ make deb
$ make rpm

# Same in chroots, targeting specific distribution versions
$ make deb_chroot DIST=jessie
$ make rpm_chroot DIST=el7

$ cd ../

# gpg key generation (one time thing)
$ gpg --gen-key

# editing the global configuration
$ vim common/buildenv/Makefile.config

# Building the repositories
# Use ERROR=skip to ignore package build failures and continue building the repo
$ make deb_repo -j 4 DIST=jessie # ERROR=skip
$ make rpm_repo -j 1 DIST=el7    # ERROR=skip
```

If you need more information, read the detailed documentation.