~~CONFIDENTIEL DÉFENSE~~ ~~SPECIAL FRANCE~~

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

PREMIER MINISTRE

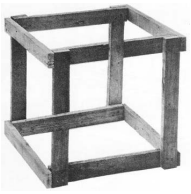
Secrétariat général de la défense et de la sécurité nationale

Agence nationale de la sécurité des systèmes d'information

DÉCLASSIFIÉ
par décision n°15699/ANSSI/SDE/ST/LAM
du 18 juillet 2018

# Documentation CLIP 4001

## Guide de création de paquetage





ANSSI, 51 boulevard de la Tour Maubourg, 75700 Paris 07 SP.

~~CONFIDENTIEL DÉFENSE~~

# HISTORIQUE

| Révision | Date | Auteur | Commentaire |
|---|---|---|---|
| 1.3 | 23/01/2013 | Mickaël Salaün | Précisions sur les conventions typographiques |
| 1.2 | 17/01/2013 | Mickaël Salaün | Multiples ajouts de détails et corrections concernant la création de paquetages |
| 1.1 | 15/01/2013 | Arnauld Michelizza, Mickaël Salaün | Ajout des annexes *Créer des paquetages CLIP pour les nuls* et *Les droits sur CLIP*. |
| 1.0 | 10/06/2011 | Vincent Strubel | Première version complète, à jour pour CLIP version 4.2.3. |

# Table des matières

**Résumé**

Ce document liste les principales spécificités de la génération de paquetages pour CLIP, et identifie les principales difficultés qui peuvent être rencontrées lors de l'écriture de telles paquetages. Il constitue ainsi un document introductif, dont la lecture est nécessaire pour tout développeur CLIP.

# 1 Récapitulatifs sur les *ebuilds*

Les paquetages CLIP se présentent du point de vue des développeurs sous la forme d'*ebuilds gentoo* qui permettent, grâce à un outil *portage* adapté pour CLIP, de générer des paquetages binaires au format *debian,* installables par les postes CLIP. La présente section traite principalement des règles et recommendations spécifiques à CLIP, applicables lors de la création ou de la modification d'*ebuilds*. Le lecteur pourra aussi se référer au document de référence [DEVREL], en particulier les chapitres "Guide pour les *ebuilds*", "Guide pour les *eclass*" et "Erreurs classiques dans les *ebuilds*", pour des informations générales sur l'écriture d'*ebuilds*. Par ailleurs, on pourra consulter le document de référence [CLIP 1101] pour une description plus détaillée du mécanisme de production de paquetages *debian* à partir d'*ebuilds* dans le cadre du développement CLIP.

## 1.1 Arborescences d'*ebuilds*

### 1.1.1 Arborescences *portage*

Comme décrit dans le document de référence [CLIP 3101], les *ebuilds* CLIP sont répartis entre quatre arborescences *portage* (qui constituent autant de sous-répertoires du *repository subversion clip-int*) :

- *portage*, pour les *ebuilds* issus de *gentoo* déployés sans modification sur les postes CLIP et éventuellement sur les postes de développement.
- *portage-overlay*, pour les *ebuilds* issus de *gentoo* déployés avec des modifications spécifiques sur les postes CLIP et éventuellement sur les postes de développement.
- *portage-overlay-clip*, pour les *ebuilds* spécifiques à CLIP (correspondant à des paquetages qui ne sont pas intégrés dans *gentoo*, notamment mais pas exclusivement [1] les paquetages spécifiques à CLIP).
- *portage-overlay-dev*, pour les *ebuilds* issus de *gentoo*, déployés sans modifications sur les postes de développement uniquement.

A ces quatre arborescences sont associés deux répertoires de *distfiles*, eux aussi des sous-répertoires du *repository clip-int*, qui contiennent les archives de sources associées aux *ebuilds* :

- *distfiles* contient les sources associées aux *ebuilds* de *portage*, *portage-overlay* et *portage-overlay-clip* uniquement.
- *distfiles-dev* contient les sources associées aux *ebuilds* de *portage-overlay-dev*, ainsi que des liens symboliques vers toutes les archives présentes dans *distfiles*.

Les différentes arborescences *portage* sont utilisées simultanément pour la compilation de paquetages, grâce au mécanisme d'*overlays* supporté par *portage*. Plus précisément, la compilation de paquetages CLIP fait appel aux arborescences *portage*, *portage-overlay* et *portage-overlay-clip* (donc pas *portage-overlay-dev*), les sources étant récupérées dans *distfiles*. Au contraire, la compilation de paquetages en vue de leur installation locale sur un poste de développement utilise l'ensemble des quatre arborescences, et récupère les sources dans *distfiles-dev* (d'où l'utilité des liens symboliques, qui permettent d'accéder aux sources de *distfiles* depuis ce répertoire). Ainsi, les paquetages spécifiques au poste de développement, qui n'ont pas fait l'objet d'une intégration au sein de CLIP, sont correctement isolés et ne risquent pas d'être par mégarde déployés sur les postes CLIP.

### 1.1.2 Répertoires d'*ebuild*

Chaque arborescence *portage* est subdivisée en catégories, sous la forme de sous-répertoires de l'arborescence, par exemple *sys-apps, sys-libs*, ou encore *app-clip* (spécifique à CLIP). Ces catégories sont à leur tour subdivisées en paquetages, associés chacun à un répertoire. Chaque répertoire de paquetage contient un ou plusieurs *ebuilds*, correspondant à une ou plusieurs versions d'un même paquetage, ainsi qu'un certain nombre de fichiers complémentaires :

1. On trouve également dans *portage-overlay-clip* des *ebuilds* associés à des logiciels publics, mais non intégrés à *gentoo*, comme par exemple *apt* (gestionnaire de paquetages *debian*).

- Un fichier *ChangeLog*, détaillant les modifications successives apportées au paquetage.
- Eventuellement un fichier *ClipChangeLog* détaillant les modifications spécifiques à CLIP apportées au paquetage. L'utilisation de ce fichier est détaillée plus bas.
- Un fichier *Manifest*, contenant les empreintes cryptographiques des différents fichiers associés au paquetage : archives de sources, *ebuilds, ChangeLog*, contenu de *files/*.
- Un répertoire *files/*, qui contient des fichiers complémentaires, complétant ceux des archives de sources. Ce répertoire peut typiquement contenir des patchs à appliquer aux sources avant leur compilation, ou des fichiers de configuration adaptés à recopier dans l'arborescence d'installation temporaire (*src_install()*, cf. 1.2). Par ailleurs, ce répertoire contient systématiquement un fichier *digest-<nom ebuild>* pour chaque *ebuild* présent dans le répertoire parent, contenant les empreintes cryptographiques des archives de sources associées à cet *ebuild*. Ce fichier est redondant avec le *Manifest*, mais nécessaire au fonctionnement de la version de *portage* utilisée dans CLIP.
- Un fichier *metadata.xml* contenant un certain nombre d'informations sur le paquetage. Ces fichiers ne sont en pratique pas utilisés dans le développement CLIP à ce stade, et ne sont pas systématiquement présents.

Les fichiers *Manifest* et *digest-** sont générés automatiquement par la commande *ebuild <nom ebuild>.ebuild digest*, lancée dans le répertoire du paquetage.

L'utilisation de deux fichiers de changements, *ChangeLog* et *ClipChangeLog*, permet de distinguer les évolutions *gentoo* d'un *ebuild* issu de cette distribution d'une part, et les évolutions spécifiques à CLIP de ce même *ebuild* d'autre part. Le fichier *ClipChangeLog* n'est ainsi présent que dans l'arborescence *portage-overlay*. Plus précisément, le schéma utilisé dans les différentes arborescences est le suivant :

- Dans *portage* et *portage-overlay-dev*, seul le fichier *ChangeLog* est présent, et contient les modifications *gentoo* (pas de modifications CLIP). Il est mis à jour en même temps qu'une nouvelle version de l'*ebuild* est importée depuis les miroirs *gentoo*, par copie du *ChangeLog gentoo*.
- Dans *portage-overlay-clip*, seul le fichier *ChangeLog* (et non *ClipChangeLog*) est présent, contenant les modifications spécifiques à CLIP, qui sont par construction les seules.
- Dans *portage-overlay*, les deux fichiers, *ChangeLog* et *ClipChangeLog*, sont présents simultanéments. Le premier est une copie du *ChangeLog gentoo* de l'*ebuild* original, et n'est mis à jour que lors de l'import d'une nouvelle version de l'*ebuild* depuis le miroir *gentoo*, tandis que le second est édité par les développeurs CLIP, pour lister les modifications spécifiques au projet. Il est aussi mis à jour lors de l'import d'une nouvelle version de l'*ebuild* (au minimum pour signaler qu'une nouvelle version a été importée, et reprend les modifications spécifiques à CLIP de la version précédente), ainsi que pour toute autre modification de l'*ebuild*.

Les deux fichiers utilisent la même syntaxe. Les nouvelles versions sont indiquées par des lignes débutant par un '*' et contenant la date d'ajout de la version. Les modifications individuelles, qu'elles donnent lieu ou non à une nouvelle version, sont indiquées avec leur date, le nom du développeur ayant réalisé la modification, et la liste des fichiers modifiés, ajoutés (précédés d'un '+') et supprimés (précédés d'un '-'). Les différents éléments sont écrits en **anglais**. L'exemple ci-dessous illustre la syntaxe attendue :

```
# CLIP ChangeLog for app-clip/clip-example

  26 Jan 2009; Vincent Strubel <clipos@ssi.gouv.fr>
  clip-example-1.0.1.ebuild :
  Fix typo in src\_install().

*clip-example-1.0.1 (24 Jan 2009)

  24 Jan 2009; Vincent Strubel <clipos@ssi.gouv.fr>
  -clip-example-1.0.ebuild, +clip-example-1.0.1.ebuild, files/config-clip :
```

```
        New release that fixes bug #42.
        Updated logging configuration.
```

La coloration syntaxique de Vim (configuration *clip-devstation*) doit être la même que pour un ChangeLog Gentoo, ce qui permet de facilement vérifier la cohérence syntaxique.

La liste suivante résume les opérations à ne pas oublier lors de la mise à jour d'un *ebuild*. On considérera à titre d'exemple la mise à jour d'un paquetage *sys-apps/example* de *portage-overlay*, pour passer de la version *1.0* à la version *1.1* :

- Dans *distfiles*, ajouter l'archive de sources *example-1.1.tbz2*, puis supprimer l'ancienne version *example-1.1.tbz2*.
- Dans *distfiles-dev*, supprimer le lien symbolique *example-1.0.tbz2* et ajouter un lien symbolique *example-1.1.tbz2*. Le script *update.sh* présent dans le répertoire permet d'automatiser ce traitement.
- Dans *portage-overlay/sys-apps/example* :
  - Supprimer *example-1.0.ebuild* et *files/digest-example-1.0*.
  - Ajouter l'*ebuild example-1.1.ebuild*, modifié pour son intégration dans CLIP.
  - Copier le *ChangeLog* et le *Manifest gentoo* à jour dans le répertoire.
  - Mettre à jour le contenu de *files* : supprimer les fichiers utilisés uniquement par *example-1.0.ebuild*, et non plus par *example-1.1.ebuild*, ajouter les fichiers manquants pour *example-1.1.ebuild*.
  - Mettre à jour le fichier *ClipChangeLog*, en y indiquant la nouvelle version, et les évolutions éventuelles des adaptations réalisées pour CLIP.
  - Régénérer le *digest* et le *Manifest* par une commande *ebuild example-1.1.ebuild digest*.
- Réaliser l'ajout/suppression de fichiers et le *commit subversion* de la modification, de préférence en un seul *commit* commun à *distfiles, distfiles-dev* et *portage-overlay*.

## 1.2 Fonctions standards d'un *ebuild*

### 1.2.1 Liste des fonctions

Chaque *ebuild* contient la définition de plusieurs fonctions standards, qui sont appelées dans un ordre prédéfini lors de l'installation de l'*ebuild*. La définition explicite de chacune de ces fonctions n'est pas nécessaire dans chaque *ebuild* : en l'absence de définition spécifique d'une fonction dans un *ebuild, portage* appelle une fonction par défaut, convenant à la plupart des cas simples. Les modifications de *portage* dans CLIP introduisent de nouvelles fonctions spécifiques à la génération de paquetages *debian*. Les paragraphes qui suivent listent ces différentes fonctions, et leur rôle dans la compilation d'un *ebuild*. Sauf mention explicite du contraire, les fonctions ci-dessous sont appelées aussi bien lors de la génération d'un paquetage *debian* que lors de l'installation locale (*gentoo*) sur le poste réalisant la compilation.

On notera par ailleurs que *portage* met en oeuvre un mécanisme de *sandbox*, qui interdit aux fonctions de l'*ebuild* et aux commandes qu'elles invoquent tout accès en écriture au système de fichiers du poste de compilation, en dehors des répertoires temporaires de compilation et d'installation (*${S}* et *${D}*). Sauf mention explicite du contraire, cette *sandbox* est active lors de l'exécution des différentes fonctions ci-dessous.

**pkg_setup()**

La fonction *pkg_setup()* est appelée en dehors de la *sandbox* avant tout autre traitement. La fonction fournie par défaut par l'environnement ne réalise aucun traitement. Elle peut être définie par un *ebuild* afin principalement de tester la configuration du poste sur lequel la compilation est réalisée, par exemple :

- Tester les drapeaux *USE* avec lesquels les dépendances de compilation ont été installées sur le poste, typiquement à l'aide de la fonction *built_with_use()* de l'*eclass eutils* (cf. 3.3), et éventuellement sortir en erreur si ces drapeaux ne permettent pas la compilation.

- Tester les versions installées des dépendances de compilation, par exemple avec la fonction *has_version()* fournie par l'environnement *ebuild*, et adapter le comportement de l'*ebuild* (paramètres du *./configure*, etc.) en fonction.
- Tester la présence de fichiers (en-têtes C, fichiers de configuration) pouvant modifier le comportement de la compilation (cas de figure assez rare).
- Avertir l'utilisateur de particuliarités de l'*ebuild*, à l'aide des fonctions *ewarn / ebeep* de *eutils.eclass*.

**fetch()**

Cette fonction réalise le téléchargement des archives de sources (définies dans la variable *${SRC_URI}* en tête de l'*ebuild*), pour en placer des copies locales dans *${DISTDIR}*. Si les archives de sources sont déjà présentes dans ce dernier répertoire, leur empreintes cryptographiques sont comparées à celles définies dans le *Manifest* qui accompagne l'*ebuild*, et les fichiers ne sont à nouveau téléchargés qu'en cas d'incohérence entre ces empreintes.

La fonction *fetch()* par défaut n'est pas surchargeable par l'*ebuild*. On notera que cette fonction ne réalise aucun traitement lorsqu'un *ebuild* n'a pas de sources (*${SRC_URI}* laissé vide). Par ailleurs, les fonctions de téléchargement ne sont a priori pas utilisées dans le cadre du développement CLIP, car les archives de sources nécessaires sont normalement présentes dans le *distfiles* ou *distfiles-dev* de la copie locale *subversion*.

Cette fonction est exécutée en dehors de la *sandbox*.

**src_unpack()**

Cette fonction réalise la décompression des archives de sources (définies par la variable *${SRC_URI}*, et précédemment placées dans *${DISTDIR}* par *fetch()*) dans le répertoire de travail *${WORKDIR}*. Ce dernier correspond par défaut au répertoire */var/tmp/portage/<catégorie ebuild>/<nom et version ebuild>/work*. La fonction *src_unpack()* est appelée automatiquement avec le répertoire *${WORKDIR}* comme répertoire courant. Il est permis de changer ce répertoire durant l'exécution de la fonction (par exemple par un '*cd ${S}*'.

La fonction *src_unpack()* par défaut réalise simplement la décompression de toutes les archives sources, par une commande :

```
        [[ -n \$\{A\} ]] \&\& unpack \$\{A\}
```

Dans cette commande, la variable *${A}*, automatiquement définie, représente la liste[2] des archives de sources correspondant à *${SRC_URI}*, dans laquelle les *URI* sont remplacées par des chemins dans *${DISTDIR}*. La première de ces archives (s'il y en a plusieurs) détermine le nom donné automatiquement à la variable *${S}*, répertoire de travail des fonctions *src_compile()* et *src_install()*. La première source de *${SRC_URI}* doit donc correspondre au *${S}* souhaité. On notera que *unpack* choisit automatiquement une méthode de décompression fonction de l'extension de chaque nom d'archive : une archive dont le nom est terminé par *.tar.gz* ou *.tgz* est décompressée avec *tar zxf*, un *.tar.bz2* ou *.tbz2* avec *tar jxf*, etc.

**src_prepare()**

La fonction *src_prepare()* est celle normalement utilisée pour appliquer des patchs aux sources à compiler. Elle n'est cependant disponible qu'à partir de la version *EAPI="2"*. Elle est appelée automatiquement avec comme répertoire courant *${S}*, répertoire racine de l'arborescence de sources principale créée par *src_unpack()*. *${S}* est automatiquement positionné au chemin absolu de l'arborescence principale de compilation, correspondant au sous-répertoire de *${WORKDIR}* créé par décompression de la première archive de sources. L'application des patchs est normalement réalisée à l'aide de la commande *epatch*, qui nécessite d'hériter l'*eclass eutils*. Au besoin, les fichiers *configure* et *Makefile.in* doivent être regénérés à l'aide de commandes *eautoreconf()* ou équivalentes (cf. 3.5). Ainsi, une fonction *prepare()* typique pourra prendre la forme suivante :

```
src_prepare() {
        # Patch applique depuis WORKDIR  :fournit par un distfile
        epatch "${WORKDIR}"/patch-all.patch || die "failed to patch"
        if use clip ; then
                epatch "${FILESDIR}"/patch-clip.patch || die "failed to patch"
        fi
```

2. Afin de maintenir l'interprétation comme une liste par *unpack*, la variable doit être passée sans guillemets : *${A}* et non *"${A}"*.

```
        eautoreconf # si necessaire
}
```

Dans des *ebuilds* antérieurs à *EAPI="2"*, l'application des éventuels patchs est réalisée par la fonction *src_unpack()*, après un changement de répertoire courant pour se placer dans *${S}*. Cependant, il est préférable, en cas de modification d'un tel *ebuild* pour ajouter un patch, d'en profiter pour le mettre à jour en *EAPI="2"*.

**src_configure()**

Cette fonction, qui n'est également disponible qu'à partir de la version *EAPI="2"*, réalise la configuration des sources, en étant lancée dans le répertoire *${S}*. Une fonction *src_configure()* consiste typiquement à appeler *econf* pour lancer le *./configure* des sources, si un tel fichier existe, avec des paramètres adaptés. La fonction par défaut, lorsqu'elle n'est pas surchargée par un *ebuild*, lance simplement *econf* sans argument supplémentaire.

La commande *econf*, fournie par l'environnement *ebuild*, passe par défaut le préfixe de compilation */usr* (sauf utilisation de *CPREFIX*, cf. 2.1), ainsi que les préfixes complémentaires associés (*--sysconfdir=/etc --localstatedir=/var*, etc.). S'y ajoutent les arguments éventuellement définis dans *${EXTRA_ECONF}* (variable interne à *portage*, typiquement modifiée par une *eclass*.

Deux raisons principalement peuvent motiver la surcharge locale de la fonction *src_configure()* par défaut :

- Soit le paquetage ne respecte pas les conventions GNU, et doit être compilé par une séquence spécifique [3].
- Soit des arguments supplémentaires doivent être passés au script de configuration standard.

Le premier cas de figure est spécifique au paquetage concerné et ne sera pas évoqué ici. Le second correspond souvent au paramétrage complémentaire de *configure* en fonction des drapeaux *USE*. A cette fin, il est utile de noter que l'environnement *ebuild* fournit deux commandes aidant à la gestion de ces options :

- *use_with <use> [<option>]* rend *"--with-<option>"* (ou par défaut *--with-<use>* lorsque *<option>* n'est pas passé) lorsque le drapeau *<use>* est actif, et *"--without-<option>"* dans le cas contraire.
- *use_enable <use> [<option>]* rend de manière similaire *"--enable-<option>"* (ou *"--enable-<use>"*) ou *"--disable-<option>"* selon le positionnement du drapeau *<use>*.

Ainsi, une fonction *src_configure()* locale pourra prendre la forme suivante :

```
src_configure() {
        econf \
                $(use_enable debug) \
                $(use_enable debug debug-extra) \
                $(use_with clip clip-support) \
                --with-config-path=${CPREFIX}/etc/toto \
                || die "econf failed"
}
```

On notera que les variables *USE* testées dans cette fonction, ou en tout autre point de l'*ebuild*, doivent être déclarées en tête de celui-ci, dans la variable *${IUSE}*.

Dans les *ebuilds* antérieurs à la version *EAPI="2"*, la fonction *src_configure()* n'existe pas et la configuration des sources est réalisée directement par le *src_compile()*, avant l'appel à *emake* (cf. paragraphe suivant). De même que pour ce qui concerne la fonction *src_prepare()* décrite plus haut, il est préférable, en cas de modification d'un tel *ebuild*, d'en profiter pour le mettre à jour et ajouter une fonction *src_configure()*.

**src_compile()**

La fonction *src_compile()* réalise la compilation des sources décompressées par *src_unpack()* et configurées par *src_configure()*. Elle est lancée avec *${S}* comme répertoire courant d'exécution. La fonction *src_compile()* par

3. On notera que le cas d'un paquetage ne contenant aucun code source compilable (paquetage de scripts, ou disponible uniquement sous forme binaire, ne rentre pas dans cette catégorie. En effet, le *src_configure()* par défaut ne réalise aucune opération dans ce cas, ce qui constitue le traitement adapté.

défaut appelle simplement *emake*, afin de lancer *make* avec quelques options adaptées. Cette fonction n'est appelée que si un fichier *Makefile* ou *GNUMakefile* est présent à la racine des sources. *emake* passe automatiquement les options *make* définies dans *${MAKEOPTS}* (variable modifiable par l'utilisateur de *portage*) et *${EXTRA_EMAKE}* (variable interne à *portage*, typiquement modifiée par une *eclass*).

Tout comme pour *src_configure()*, il peut être nécessaire de modifier le *src_compile()* d'un *ebuild* si jamais le paquetage concerné doit être compilé par une autre procédure qu'un simple *make* à la racine, ou si des options supplémentaires doivent être passées à *emake*. Dans ce dernier cas, la fonction redéfinie pourra prendre la forme suivante :

```
src_compile() {
        local mkopts="WITH_SSL=1"
        use clip && mkopts="${mkopts} WITH_CLIP=1"
        emake ${mkopts} || die "emake failed"
}
```

**src_install()**

La fonction *src_install()* est appellée après *src_compile()* afin d'installer le paquetage précédemment compilé dans une arborescence temporaire, dénomée *${IMAGE}* ou *${D}*. Ces deux variables sont positionnées automatiquement à *${WORKDIR}/image*. La fonction *src_install()* est automatiquement lancée avec un répertoire courant d'exécution correspondant à *${S}*, comme *src_compile()*.

La fonction *src_install()* par défaut ne réalise aucun traitement. Il est ainsi nécessaire en général de définir une fonction spécifique pour chaque *ebuild*, soit dans l'*ebuild* concerné lui-même, soit dans une des *eclass* dont il hérite. Une telle fonction réalisera en général un *make install*, sous la forme suivante :

```
        emake DESTDIR=${D} install
```

L'utilisation de *DESTDIR* est obligatoire dans ce cas, afin de réaliser l'installation dans le répertoire temporaire *${D}* plutôt que dans le système de fichiers principal du poste de compilation.

Outre ce *make install*, la fonction *src_install()* peut réaliser d'autres opérations sur l'arborescence temporaire *${D}*, typiquement pour installer des fichiers supplémentaires (issus de l'arborescence de sources *${S}*, d'une autre archive de sources, ou de *${FILESDIR}*, ou modifier certains fichiers de cette arborescence. De telles opérations sont de préférence réalisées à l'aide des différents *wrappers* fournis par l'environnement *ebuild* pour la plupart des commandes de base, par exemple *dodir, doins, doexe*, etc. Ces *wrappers*, dont une liste complète est donnée dans le document de référence , travaillent implicitement sur *${D}*. Ainsi, la commande *dodir /usr/lib/toto* créera automatiquement le répertoire *${D}/usr/lib/toto* (ou *${D}/${CPREFIX}/lib/toto* si *CPREFIX* est définie, cf. 2.1). Dans les cas où aucun *wrapper* n'est fourni pour réaliser une commande donnée, il est important de passer explicitement *${D}* dans le chemin des fichiers manipulés, afin d'éviter une violation de *sandbox*.

**deb()**

Cette fonction réalise la création d'un paquetage *debian*, par le traitement décrit dans le document de référence [CLIP 1101]. Cette fonction n'est pas directement surchargeable, mais appelle *pkg_predeb()* si elle est définie dans l'*ebuild*.

**pkg_predeb()**

Cette fonction est appelée par *deb()*, et permet de réaliser des traitements spécifiques avant la génération de paquetage *debian*. Elle n'est pas définie par défaut, et peut librement être définie dans l'*ebuild*. Elle est exécutée automatiquement avec *${IMAGE}* comme répertoire de travail. On notera qu'il s'agit bien de *${IMAGE}* et non de *${D}*, ces deux chemins n'étant pas équivalents en cas d'utilisation de *CLIP_VROOTS* (cf. 2.1.3

Un intérêt majeur de cette fonction est de n'être invoquée que lors de la création de paquetage *debian*, et non lors de l'installation sur le poste de compilation. Elle est ainsi particulièrement adaptée à la création de *maintainer scripts debian* (cf. 2.3). On notera de plus qu'elle est appellée après la suppression par *strip* des symboles non nécessaires au sein des exécutables installés par *src_install()*, et permet donc de prendre des empreintes cryptographiques fiables de ces exécutables (par exemple pour la gestion de *veriexec*, cf. 2.4), qui

ne sont pas modifiés après cela jusqu'à leur inclusion dans le paquetage binaire. Au contraire, des empreintes calculées lors du traitement de *src_install()* ne correspondraient pas aux exécutables installés par le paquetage *debian,* du fait de leur modification par *strip*.

**pkg_preinst()**

Cette fonction est appellée avant *qmerge()*. Elle permet d'agir sur *${D}* et sur le système de fichiers du poste de compilation avant la création du paquetage *debian* ou l'installation sur le poste. La différence majeure de cette fonction par rapport à *src_install()* tient au fait qu'elle n'est pas exécutée dans la *sandbox*. Par ailleurs, la fonction est, tout comme *pkg_predeb()*, appelée après le *strip* des exécutables installés par *src_install()*.

On notera bien que, contrairement à ce que pourrait laisser supposer son nom, la fonction n'est pas invoquée avant *src_install()*, mais bien après celle-ci et avant *qmerge()*. En particulier, cette fonction n'est jamais appelée dans le cadre de la création d'un paquetage *debian*.

La fonction par défaut ne réalise aucune opération.

**qmerge()**

Cette fonction réalise le transfert de l'arborescence d'installation temporaire *${IMAGE}* vers l'arborescence racine du poste de compilation. Plus précisément, les fichiers de l'arborescence *${IMAGE}* sont déplacés dans l'arborescence dont la racine correspond à la variable *${ROOT}* si elle est définie, ou à défaut à la racine du poste de compilation. Après cette installation, les éventuelles versions précédentes du même *ebuild* sont désinstallés du système (seuls les fichiers de l'ancienne version qui n'ont pas été écrasés par la nouvelle sont supprimés).

Cette fonction n'est en aucun cas surchargeable par un *ebuild*. Elle est naturellement exécutée en dehors de la *sandbox*.

**postinst()**

Cette fonction est exécutée en dehors de la *sandbox,* après le traitement de *qmerge()* (donc en particulier pas lors de la génération d'un paquetage *debian*). Elle permet de modifier le système hôte (ou le système situé au sein de l'arborescence *${ROOT}*, lorsque celle-ci est définie) après installation du nouveau paquetage, par exemple pour regénérer un cache [4] ou modifier un fichier de configuration).

La fonction par défaut ne réalise aucun traitement.

### 1.2.2 Séquence d'appels

Comme le mettent en évidence les paragraphes précédents, la séquence d'appels de fonctions *ebuild* n'est pas strictement équivalente lors de la génération d'un paquetage *debian* et lors de l'installation sur le poste hôte de la compilation. La présente section décrit précisément ces deux séquences, afin d'en éclaircir les différences.

Ainsi, lors d'une installation sur le poste hôte, la séquence suivante est utilisée :

- *pkg_setup()*
- *fetch()*
- *src_unpack()*
- *src_prepare()*
- *src_configure()*
- *src_compile()*
- *src_install()*
- *pkg_preinst()*
- *qmerge()*
- *pkg_postinst()*

A contrario, lors de la génération d'un paquetage *debian* (*emerge --builddebonly* / *-M*), la séquence utilisée est la suivante :

- *pkg_setup()*
- *fetch()*

4. Le cache de recherche de bibliothèques est automatiquement régénéré après *qmerge*, donc il n'est pas nécessaire d'appeler *ldconfig* directement à ce stade.

- *src_unpack()*
- *src_prepare()*
- *src_configure()*
- *src_compile()*
- *src_install()*
- *pkg_predeb()*
- *deb()*

Il en résulte que *pkg_preinst()* et *pkg_postinst()* peuvent être utilisée pour des traitements spécifiques à l'installation locale, jamais réalisés pour la production de paquetages CLIP, tandis que *pkg_predeb()* permet au contraire des traitements réservés aux paquetages CLIP, sans incidence sur le poste hôte.

**Remarque 1 : mode combiné génération de paquetage debian / installation locale**

*L'outil* portage *modifié pour CLIP supporte à ce stade un mode combiné d'installation locale et de génération de paquetage* debian*, invoqué par la commande* emerge --builddeb*. Cette option n'est présente que pour des raisons historiques, et ne doit pas être employée, car elle conduit souvent à l'installation locale de paquetages peu adaptés à un poste de développement (du fait de l'application de* pkg_predeb()*).*

## 1.3 Fonctions partagées : *eclass*

### 1.3.1 Principe des *eclass*

Les *eclass* constituent des portions de code partagées entre plusieurs *ebuilds*. Un *ebuild* peut hériter d'une ou plusieurs *eclass* par une déclaration *inherit <eclass1> [<eclass2>...]* en début de fichier. L'héritage d'une *eclass* est pour l'essentiel équivalent à une directive *source <fichier>* en *bash*, et importe dans l'*ebuild* appelant les variables et fonctions définies par l'*eclass*.

Ainsi, les *eclass* peuvent être utilisées pour mettre en commun des sources (définition commune de *SRC_URI*), des dépendances (définition ou modification commune de *RDEPEND, DEPEND,* etc.), des options de compilation (*CFLAGS, EXTRA_EMAKE*, etc.), ou fournir des fonctions utilitaires à un ensemble d'*ebuilds*. Elles peuvent également fournir des versions spécialisées des fonctions principales d'*ebuild* décrites en 1.2. A cette fin, elles utilisent préférentiellement un mécanisme de définition reposant sur la macro *EXPORT_FUNCTIONS*, qui permet une double définition d'une fonction *ebuild* donnée, par exemple *src_install()*, à la fois sous le nom *src_install()* lui-même et sous le nom *<nom eclass>_src_install()*, où *<nom eclass>* désigne le nom du fichier *eclass,* hors extension *.eclass*. De ce fait, un *ebuild* héritant d'une telle *eclass* pourra utiliser automatiquement la fonction *src_install()* de l'*eclass*, en ne définissant pas lui-même cette fonction, ou alternativement redéfinir *src_install()* en appelant dans celle-ci *<nom eclass>_src_install()*, afin d'appeler le traitement commun et de le compléter au besoin d'opérations spécifiques. Il est à noter que les différentes *eclass* surchargent leurs fonctions et variables dans l'ordre de leur inclusion sur la ligne *inherit*. Ainsi, si *eclass1.eclass* et *eclass2.eclass* définissent toutes deux une fonction *src_install()*, un *ebuild* incluant une directive *inherit eclass1 eclass2* utilisera par défaut le *src_install()* de *eclass2*, à moins de définir un *src_install()* appelant explicitement *eclass1_src_install()* (et au besoin *eclass2_src_install()*).

Les *eclass* sont stockées dans le répertoire *eclass/* d'une arborescence *portage*. Le mécanisme d'*overlay portage* ne s'applique que partiellement pour les *eclass*. En effet, si une *eclass* peut bien être présente dans le répertoire *eclass/* d'un *overlay*, elle ne peut en revanche pas masquer une *eclass* du répertoire *portage* principale. De ce fait, la répartition suivante est utilisée au sein des arborescences CLIP :

- Les *eclass* issues de *gentoo* et non modifiées sont placées dans *portage/eclass*.
- Les *eclass* issues de *gentoo* et modifiées sont placées dans *portage-overlay/eclass*, mais des liens symboliques pointant vers ces *eclass* sont conservés dans *portage/eclass*.
- Les *eclass* spécifiques à CLIP sont placées dans *portage-overlay-clip/eclass*.

On notera bien que les *eclass* d'un *overlay* peuvent être librement incluses dans les *ebuilds* d'un autre *overlay* : un *ebuild* de *portage* ou *portage-overlay-dev* utilisera automatiquement la version adaptée d'une *eclass*

présente dans *portage-overlay*, et un *ebuild* de *portage-overlay* pourra hériter d'une *eclass* spécifique de *portage-overlay-clip*.

### 1.3.2 *Eclass* utiles

Les exemples ci-dessous constituent des *eclass* très fréquemment utilisées dans les *ebuilds*. Seules les fonctions les plus courantes de ces *eclass* sont mentionnées. Sauf mention explicite du contraire, ces *eclass* sont issues de *gentoo* et non spécifiques au projet CLIP, bien qu'elles puissent faire l'objet de certaines adaptations pour celui-ci.

**eutils**

Contient un ensemble de fonctions très couramment utilisées, en particulier :

- *built_with_use <pkg> <use>* : permet de tester si le paquetage installé *<pkg>* (nom du paquetage) a été compilé avec le drapeau *USE <use>*.
- *epatch <patch>* : permet d'appliquer le patch *<patch>* aux sources d'un paquetage. La fonction doit être appelée dans le répertoire *${S}*. *<patch>* peut être le chemin vers un patch unique (éventuellement compressé, avec une extension -- *.bz2,.zip,*etc. -- permettant de déterminer l'algorithme de compression), ou un répertoire contenant un ensemble de patchs, qui sont dans ce dernier cas appliqués successivement dans l'ordre alphabétique.
- *ewarn / einfo <msg>* : permettent d'afficher un message *<msg>* sur la console de compilation, avec un code de couleur approprié.
- *make_desktop_entry()* : permet de créer un fichier *.desktop* pour ajouter une application aux menus *KDE, gnome,* etc.
- *enewuser() / enewgroup()* : permettent de créer respectivement un nouvel utilisateur et un nouveau groupe sur le système.

On notera que le dernier couple de fonctions ne fonctionne correctement que dans le cas de l'installation directe sur le poste de compilation, et non dans le cas d'un paquetage *debian* pour CLIP. Dans ce dernier cas, la manipulation des utilisateurs doit être faite par des moyens ad-hoc : par modification du paquetage *sys-apps/baselayout-clip*, qui installe les fichiers */etc/passwd* et */etc/group* initiaux, et/ou par un script *postinst* spécifique (uniquement possible dans le cas d'un paquetage du socle CLIP).

**toolchain-funcs**

Permet (entre autres) de tester le compilateur *gcc* par défaut du système sur lequel un *ebuild* est compilé, et ses fonctionalités.

- *gcc-specs-ssp()* : teste si le compilateur utilise le profil *hardened/ssp* (mise en oeuvre de *stack-smashing-protection*, cf. [CLIP 1101], par défaut).
- *gcc-specs-pie()* : teste si le compilateur utilise le profil *hardened/pie*(génération de *Position Independent Executables*, cf. [CLIP 1101], par défaut).
- *gcc-specs-relro(), gcc-specs-now()* : testent si l'éditeur de lien par défaut utilise les options respectives *relro* (sections marquées en lecture seule après relocalisation) et *now* (édition de liens dynamique au lancement du programme, plutôt que "paresseuse", cf. [CLIP 1101].
- *gcc-version(), gcc-major-version(), gcc-minor-version(), gcc-micro-version()* : retournent le numéro de version complet ou partiel du compilateur par défaut.

**flag-o-matic**

Fournit des fonctions de manipulations des options passées au compilateur (*CFLAGS, CXXFLAGS*) ou à l'éditeur de lien (*LDFLAGS*), notamment :

- *append-flags <flags>* : permet d'ajouter le ou les options *<flags>* aux drapeau de compilation *CFLAGS* et *CXXFLAGS*.
- *filter-flags <flags>* : permet de supprimer les options *<flags>* des *CFLAGS* et *CXXFLAGS* passés par l'utilisateur.

- *append-ldflags() / filter-ldflags()* : permettent les mêmes traitement sur les *LDFLAGS* passés par l'utilisateur.
- *replace-flags <flags1> <flags2>* : permet de remplacer les options *<flags1>* par les options *<flags2>* dans les *CFLAGS* et *CXXFLAGS* passés par l'utilisateur.

Pour être efficaces, ces fonctions doivent être appelées en général dans *src_compile()*, avant l'appel à *econf* ou équivalent.

Un cas d'usage typique est la désactivation au besoin de *SSP* dans les paquetages qui ne le supportent pas, sous la forme suivante :

```
# suppression des drapeaux passes par l'utilisateur
filter-flags -fstack-protector -fstack-proctor-all

# annulation des drapeaux par defaut, au besoin
gcc-specs-ssp && append-flags -fno-stack-protector -fno-stack-protector-all
```

Un autre cas d'emploi est la désactivation de drapeaux d'optimisation causant des erreurs dans le programme concerné, par exemple pour un paquetage à ne compiler qu'avec *-O2* (à l'exclusion de *-Os, -O3, -O1,...*) :

```
replace-flags -O? -O2
```

**autotools**

Cette *eclass* permet l'appel des outils *autotools* pour générer ou régénérer les fichiers *configure* et *Makefile.in* d'un paquetage source construit à l'aide de ces outils. La principale fonction à retenir est *eautoreconf()*, qui appelle *autoreconf* avec des arguments adaptés, et permet de régénérer l'ensemble des fichiers *autotools* d'un paquetage. La fonction doit être appelée avec *${S}* comme répertoire courant. Son usage est recommandé dans le *src_unpack()* d'un *ebuild*, après l'application d'un patch modifiant le *configure.in* ou un *Makefile.am* des sources, afin de régénérer les fichiers *autotools*. Alternativement, *autotools.eclass* fournit également des fonctions plus élémentaires, *eautoconf(), eautomake(), eaclocal()*, etc., à n'utiliser qu'en connaissance de cause.

**verictl2**

Cette *eclass* est spécifique à CLIP, et permet la gestion d'entrées *veriexec*. Elle est détaillée en 2.4.

**deb**

Cette *eclass* est spécifique à CLIP, et permet la gestion de *maintainer-scripts* dans les paquetages *debian*. Elle est détaillée en 2.3.

**pax-utils**

Cette *eclass* fournit des fonctions utiles pour positionner des drapeaux déclarant des "exceptions" *PaX* sur des exécutables ELF, afin typiquement de désactiver certains mécanismes *PaX* sur certains exécutables spécifiques (cf. [CLIP 1203]). Les fonctions utiles fournies par l'*eclass* sont principalement :

- *pax-mark <flags> <exec1> ... <execN>* : ajoute les drapeaux *PaX <flags>* sur les exécutables *<exec1> ... <execN>*, en utilisant l'utilitaire approprié (*chpax*, *paxctl* ou *scanelf*, selon le type de drapeaux supportés et les utilitaires disponibles [5], et en créant au besoin les segments ELF nécessaires.
- *list-paxables <path>* : liste les fichiers susceptibles de recevoir des drapeaux *PaX* (c'est-à-dire les binaires ELF) dans le chemin *<path>* et ses sous-répertoires.

La fonction *pax-mark* peut être utilisé dans un *ebuild* pour désactiver certaines protections mémoires incompatibles du fonctionnement d'un exécutable, typiquement *PAX_MPROTECT*, qui est incompatible avec le fonctionnement d'un interpréteur réalisant de la compilation de code *Just In Time*. Ainsi, on pourra placer un appel de la forme suivante dans la fonction *src_install* (cf. 1.2) d'un *ebuild* installant une machine virtuelle *java* :

```
pax-mark 'm' "${D}${CPREFIX :-/usr}/lib/java/jre/bin/java"
```

On notera que le chemin complet doit être passé à *pax-mark*, qui ne gère pas automatiquement le préfixe *$D*, ni *$CPREFIX*.

5. L'utilisation de *paxctl* est à privilégier sous CLIP, qui utilise les drapeaux *PT_PAX_FLAGS* et ajoute certains drapeaux *PaX* spécifiques (*PAX_ELFRELOCS*) supportés uniquement par un utilitaire *paxctl* modifié.

De même, on pourra utiliser des drapeaux *PaX*, cette fois spécifiques à CLIP, pour autoriser le chargement d'une bibliothèque contenant des relocalisations dans le texte (*TEXTREL* [6]). Il est nécessaire pour cela d'ajouter le drapeau *PAX_ELFRELOCS* non seulement sur la bibliothèque contenant les relocalisations dans le texte, mais aussi sur chaque exécutable utilisant cette bibliothèque, par des commandes :

```
pax-mark 'L' "${D}/chemin/vers/fichier"
```

**Familles de composants**

Un certain nombre de familles de composants logiciels liés entre eux ou très similaires utilisent des *ebuilds* extrèmement simples, l'essentiel du code étant mis en commun dans une ou plusieurs *eclass* par famille. Il s'agit notamment des composants du serveur *X.org* (*x-modular.eclass*), des paquetages *KDE* (*kde.eclass, kde-meta.eclass*) ou encore des modules *java* (*java.eclass, java-*.eclass*). Il n'est pas dans le propos du présent guide de détailler ces différentes classes, mais leur existence est à connaitre pour tout développement touchant aux composants concernés.

### 1.3.3 Ajout ou modification d'une *eclass*

L'ajout ou la modification d'*eclass* dans les arborescences *portage* CLIP est possible, sous réserve de respecter quelques règles simples. Tout d'abord, comme pour l'ajout d'*eclass* dans *gentoo*, l'attention des développeurs est attirée sur la nécessité de permettre un héritage multiple de plusieurs *eclass*, sans conflit. En particulier :

- Les définitions de variables doivent prendre la forme d'ajouts plutôt que de redéfinition : on utilisera *DEPENDS="${DEPENDS} toto"* plutôt que *DEPENDS="toto"*.
- Toute utilisation de drapeau *USE* dans l'*eclass* doit entrainer l'ajout des drapeaux concernés à *IUSE* dans la même *eclass*. On se souviendra que la redondance des drapeaux dans *IUSE* n'est pas gênante, alors que l'utilisation d'un drapeau *USE* non déclaré dans *IUSE* est potentiellement problématique.
- Les fonctions standards d'*ebuild* doivent toujours être définies à l'aide de la macro *EXPORT_FUNCTIONS*, plutôt que directement.

Pour ce dernier point, on utilisera typiquement une construction de la forme (en supposant l'*eclass* nommée *toto.eclass*) :

```
toto_src_install() {
        # code...
}
EXPORT_FUNCTIONS src_install
```

Une telle directive crée automatiquement un alias de *src_install()* appelant *toto_src_install()*.

Par ailleurs, l'environnement de développement CLIP impose deux contraintes supplémentaires :

- Chacun des trois répertoires *eclass* des arborescences CLIP contient un fichier supplémentaire, *eclasses.version*. Ce fichier doit contenir la date de la dernière modification d'une *eclass* dans le répertoire concerné, sous la forme YYYYMMJJ. Il doit être mis à jour et "commité" dans *subversion* pour toute modification d'*eclass*.
- Le répertoire *portage-overlay/eclass* contient un fichier *ClipChangeLog* spécifique, détaillant les modifications spécifiques à CLIP apportées aux *eclass gentoo* (même si celles-ci ne sont pas elles-mêmes accompagnées d'un *ChangeLog*). Le format de ce fichier est le même que dans un répertoire d'*ebuild* (cf. 1.1.2), à ceci près qu'il contient autant d'entrées que d'*eclass* dans le répertoire. Il doit également être mis à jour pour toute modification.

## 1.4 Ajout de patchs dans un *ebuild*

Il peut être nécessaire dans certains cas d'ajouter pour les besoins de l'intégration dans CLIP un patch aux sources compilées par un *ebuild*. Dans ce cas, les règles suivantes doivent être respectées :

6. La présence de tels *TEXTREL* est signalée par *portage* par des avertissements après l'installation d'un paquetage. La mise en oeuvre de relocalisations dans le texte est, en l'absence de traitement spécifique, incompatible avec les protections *Pax* (*PAX_MPROTECT* en particulier).

- Le patch doit être créé comme un *diff* unifié (commande *diff -u*), applicable par *patch -p0* ou *patch -p1* dans l'arborescence *${S}*.
- Un patch unique est normalement stocké dans le *${FILESDIR}* de l'*ebuild*, sauf s'il est de taille très importante (supérieure à la dizaine de kilo-octets), auquel cas il est préférable de le compresser avec *bzip2*, et de l'ajouter aux *distfiles* de l'*ebuild*, dans *${SRC_URI}*. Il est recommandé de nommer un patch unique suivant la forme *${PN}-<version>-<fonction>.patch*, avec :
  - *${PN}* le nom hors version de l'*ebuild* (variable automatiquement définie),
  - *<version>* le numéro de la première version des sources auxquelles le patch s'applique, éventuellement complété d'un *-r#* si plusieurs révision du même patch ont été réalisées,
  - *<fonction>* une indication de ce qui est corrigé ou ajouté par le patch, par exemple *CVE-20YY-XXXX* dans le cas d'une correction de vulnérabilité ou *clip-mayexec* pour une restriction spécifique au système CLIP.
- Lorsqu'un grand nombre de patchs doivent être appliqués par un même *ebuild*, il est utile de les rassembler dans un *distfile* unique, ajouté à *${SRC_URI}*. Dans ce cas, le nom des patchs individuels doit être préfixé d'un numéro *XXXX* correspondant à l'ordre dans lequel ces patchs sont appliqués, par exemple *0001-foobar-1.0-do-something.patch*, *0002-foobar-1.0-do-something-else.patch*. Le *distfile clip-patches* appliqué par l'*ebuild sys-kernel/clip-kernel* fournit une illustration de ce principe.
- Les différents patchs doivent être appliqués aux sources dans le traitement de *src_unpack()*, en utilisant la fonction *epatch* de *eutils.eclass* (cf. *src_unpack* en section 1.2).
- Lorsqu'il est nécessaire de modifier les fichiers liés à la configuration ou à la compilation et produits par *autotools* (*configure, Makefile.in*), le patch doit porter sur les fichiers d'entrées *autotools* correspondants (*configure.in, Makefile.am*), et non sur les fichiers produits. Ceci permet de limiter la taille du patch, et d'améliorer sa portabilité. Après application d'un tel patch, il est nécessaire de régénérer les fichiers produits par *autotools* à l'aide de l'une des fonctions de *autotools.eclass* (*eautoreconf()* ou autre, cf. 1.3.2).

# 2 Contraintes spécifiques aux *ebuilds* CLIP

## 2.1 Spécificités des chemins d'installation dans CLIP

CLIP utilise une arborescence de fichiers assez complexe dans ces différentes cages et vues, qui impose un certain nombre de contraintes supplémentaires sur les chemins de configuration et d'installation, qui doivent être prises en compte lors de l'écriture de nouveaux *ebuilds*, et qui nécessitent parfois des modifications des *ebuilds* existants.

### 2.1.1 Répertoires partagés et non partagés

Un certain nombre de chemins d'installation sont à éviter sous CLIP dans bon nombre de paquetages, dans la mesure où ils ne sont pas partagés entre cages. Ainsi, un paquetage destiné à être utilisé dans la vue USER d'une cage RM ne doit pas installer des fichiers dans */opt*, car celui-ci n'est pas partagé entre la vue UPDATE et la vue USER : les fichiers seraient créés dans le */opt* (sous réserve de droit d'accès en écriture) de la vue UPDATE, mais ne seraient jamais accessibles sous */opt* dans la vue USER. Une liste exhaustive des chemins utilisables pour les différents usages nécessite de consulter la composition exacte des arborescences des cages considérées, dans les documents de référence [CLIP 1304] et [CLIP 1401]. Il est néanmoins possible de décrire ici un certain nombre de principes génériques :

- Le */opt* n'est pas partagé entre cages et vues. Les paquetages installant des fichiers dans */opt* doivent être modifiés dans CLIP pour les installer dans un autre répertoire, typiquement un sous-répertoire de */usr/lib* ou de */usr/local/lib*.
- Les répertoires */bin*, */sbin* et */etc* sont généralement propres à chaque compartiment. Les fichiers normalement installés dans ces répertoires et qui ont vocation à être partagés entre compartiment doivent être déplacés dans des sous-répertoires équivalents de */usr* ou */usr/local*, ou dans */etc/shared*.
- Le répertoire */var* et ses sous-répertoires sont généralement (sauf montage spécifique, par exemple */var/shared* dans les vues USER et UPDATE des cages RM) propres à chaque compartiment. Il peut donc être nécessaire d'initialiser une arborescence de répertoires nécessaires à un paquetage donné dans un autre paquetage, correspondant à l'arborescence de base du compartiment concerné, ou de déplacer cette arborescence dans un répertoire partagé (*/var/shared* en particulier).
- De même, le répertoire */tmp* est généralement spécifique à chaque compartiment, et souvent non rémanent (c'est-à-dire créé par le montage d'un système de fichiers *tmpfs*, dont le contenu sera perdu au prochain démarrage). Il n'est donc pas souhaitable de chercher à installer des fichiers ou répertoires dans */tmp* depuis un paquetage. En cas de besoin incontournable, la création des répertoires ou fichiers devra être réalisée par un script ad-hoc invoqué à chaque démarrage ou ouverture de session, plutôt que par le paquetage lui-même ou ses *maintainer scripts*.
- Seul */usr/lib* est partagé entre les cages ou vues UPDATE et ADMIN ou AUDIT. Les fichiers qui ont vocation à être partagés entres ces compartiments (normalement uniquement des bibliothèques ou fichiers de configuration) doivent être placés dans ce répertoire, ou dans */etc/shared*, plutôt que par exemple dans */usr/libexec* ou dans un sous-répertoire spécifique de */usr* (*/usr/i686-pc-linux-gnu* ou équivalent).

Dans des *ebuilds* existants, ces contraintes peuvent nécessiter des modifications au niveau des chemins passés à *configure* ou des commandes lancées dans *src_install*(). Ces modifications sont typiquement conditionnées par le drapeau *USE clip*, qui identifie la compilation à destination de postes CLIP. On pourra avoir par exemple :

```
src_install() {
        <...>
        local path="/opt/toto/plugins"
        use clip && path="/usr/lib/toto/plugins" # Ajoute pour CLIP
        dodir "${path}"
        insinto "${path}"
        doins "${S}"/toto_plugin.so
```

```
}
```

On notera bien que l'absence de partage de certaines arborescences entre compartiments logiques peut a contrario être une propriété désirable pour certains paquetages. Ainsi, bon nombre de fichiers de configuration et d'exécutables du socle CLIP sont installés à dessein dans */etc* et */sbin* plutôt que dans */etc/shared* et */usr/sbin* de manière à justement éviter leur exposition dans les cages CLIP.

Par ailleurs, certains répertoires jouent un rôle spécifique dans le partage entre cages et vues de fichiers de configuration, et peuvent être utilisés à cette fin dans les *ebuilds*. Il s'agit plus particulièrement des répertoires suivants :

- */etc/core* : permet le partage de fichiers de configuration globaux entre tous les compartiments du système. Un fichier ne peut être installé dans ce répertoire que par un paquetage primaire CLIP. Les fichiers *passwd* et *group*, listant les comptes et groupes utilisateurs définis au sein du système sont des exemples de fichiers ainsi partagés.
- */etc/shared* : permet le partage de fichiers entre toutes les cages CLIP d'une part, et entre toutes les vues de chaque cage RM d'autre part. Un fichier ne peut être placé dans ce répertoire que par un paquetage primaire CLIP ou RM, respectivement. Les fichiers de configuration associés à la *glibc* (*/etc/shared/nsswitch.conf*, etc.) ou *ncurses* (*/etc/shared/terminfo/**) sont typiques de fichiers partagés de cette manière.
- */etc/admin* : permet le partage de fichiers modifiables par l'administrateur local entre toutes les cages CLIP d'une part, et entre toutes les vues de chaque cage RM d'autre part. Contrairement à */etc/shared*, des fichiers peuvent être placés dans ce répertoire par n'importe quel paquetage (secondaire aussi bien que primaire) de la distribution concernée (CLIP ou RM, respectivement). En revanche, ces fichiers doivent en général être accessibles de l'administrateur local, et donc appartenir à l'utilisateur d'*uid 4000*, et au groupe de *gid 4000*. De plus, de tels fichiers modifiables doivent généralement être définis spécifiquement comme des fichiers de configuration (à l'aide de la variable *CLIP_CONF_FILES*, cf. 2.5) dans l'*ebuild*, afin de ne pas écraser les modifications locales des fichiers lors de la mise à jour du paquetage qui les a installés sur les postes clients.

### 2.1.2 Utilisation de */usr/local*

Outres les répertoires problèmatiques évoqués en 3.4.1, CLIP présente la particularité de nécessiter des paquetages secondaires installés dans une arborescence entièrement située au dessus de */usr/local*, plutôt que dans */etc* et */usr*, qui sont les préfixes de configuration et d'installation par défaut de *gentoo*. Ce chemin spécifique est pris en compte à travers une variable d'environnement spécifique à CLIP, *CPREFIX*. La signification exacte de cette variable, lorsqu'elle est définie, est la suivante :

- Les exécutables, bibliothèques et données des *ebuilds* doivent être installés dans des sous-répertoires (*bin/, sbin/, lib/, share/*) de *${CPREFIX}* plutôt que dans ceux de */usr*, par exemple *${CPREFIX}/lib* au lieu de */usr/lib*. Ces chemins correspondent typiquement à ceux déterminés par les options *--prefix, --exec-prefix, --datadir*, etc. des scripts *configure* de paquetages sources.
- Les fichiers de configuration (correspondant à l'option *--sysconfdir* des scripts *configure*) sont déplacés de */etc* à *${CPREFIX}/etc*.
- Le répertoire */var* (typiquement *--localstatedir*) reste inchangé, de même que */tmp*.
- Le répertoire */opt* devient *${CPREFIX}/opt*, bien que ce répertoire soit à éviter sous CLIP (cf. 2.1.1).

Lors de la compilation des paquetages secondaires CLIP, *CPREFIX* est typiquement définie à */usr/local* par le fichier *spec* configurant la génération de paquetages (cf. 3.1). Cette variable est normalement laissée indéfinie (vide) pour la compilation des paquetages primaires. Les outils *portage* sont modifiés dans CLIP de manière à prendre en compte cette variable *CPREFIX* automatiquement dans la plupart des cas. Cette prise en compte est réalisée à deux niveaux :

- Les arguments par défaut passés à *./configure* par la fonction *econf* sont ajustés aux nouveaux préfixes.
- Les fonctions *"wrappers" portage* travaillant de manière transparente sur le répertoire d'installation temporaire *${D}* et typiquement invoquées dans *src_install()* (cf. 1.2), sont modifiées de manière à prendre en

compte la variable *${CPREFIX}*, lorsqu'elle est définie, dans les chemins de fichiers manipulés, sans modification des arguments. Cette modification s'applique à toutes les fonctions **into()* (*insinto(), exeinto()*, etc.), *do*()* (*dobin(), doman(), dosed()*, etc.) et *new*()* (*newexe(), newins()*, etc.), ainsi que *keepdir()*.

Il est important de bien noter le fait que cette deuxième modification s'applique à arguments constants. Ainsi, une commande *dodir "/usr/lib/toto"* créera automatiquement un répertoire *${CPREFIX}/lib/toto* si *CPREFIX* est définie, et */usr/lib/toto* dans le cas contraire. En revanche, mentionner explicitement *CPREFIX* dans les arguments d'un tel *wrapper* est généralement une erreur. Par exemple, une commande *dodir "${CPREFIX :-/usr}/lib/toto"* créera un répertoire */usr/local/local/toto* (double *"local"*) dans le cas *CPREFIX="/usr/local"*.

Il peut être nécessaire de modifier spécifiquement certains *ebuilds* pour compléter cette gestion automatique de *CPREFIX*, principalement dans les cas suivants :

- Arguments spécifiques passés à *econf* et contenant des chemins (par exemple l'argument *--sysconfdir=/etc/toto* pour surcharger le */etc* utilisé par défaut), ou appel direct de *./configure* ou équivalent sans passer par *econf*.
- Préfixes passés directement en argument de *make*, plutôt qu'à travers le *./configure*, par exemple *make PREFIX=/usr* dans *src_compile()*.
- Commandes invoquées dans *src_install()* et manipulant des fichiers dans *${D}* sans passer par les *wrappers portage* (par exemple commandes *rm*, pour lesquelles il n'existe pas de *wrapper*).

Dans ces différents cas, les chemins manipulés devront être modifiés directement dans l'*ebuild*, en remplaçant notamment les occurences de */usr* par *${CPREFIX :-/usr}* (qui est correct que *CPREFIX* soit définie ou non), et celles de */etc* par *${CPREFIX}/etc*. On prendra particulièrement garde aux chemins utilisés dans *src_install()* et passés aussi bien à des *wrappers portage* que par des commandes directes. De plus, il est important de prendre en compte dans une telle analyse les traitements qui pourraient être réalisés au sein d'*eclass* héritées par l'*ebuild*, plutôt que dans l'*ebuild* lui-même.

En résumé, si l'on considère les portions de code suivantes :

```
src_compile() {
        econf \
                --with-toto-dir="/usr/lib/toto"
        emake TOTO_CONFPATH="/etc/toto"
}

src_install() {
        emake DESTDIR=${D} install
        local totopath="/usr/lib/toto"
        exeinto "${totopath}"
        doexe "${FILESDIR}/toto-wrapper.sh"
        rm -fr "${D}${totopath}/plugins"
        [...]
}
```

Celles-ci devront être modifiées pour s'intégrer dans CLIP de la manière suivante :

```
src_compile() {
        econf \        # Parametres par defaut ajustes automatiquement
           --with-toto-dir="${CPREFIX :-/usr}/lib/toto" # ajustement specifique
        emake TOTO_CONFPATH="${CPREFIX}/etc/toto" # idem
}

src_install() {
        emake DESTDIR=${D} install
        local totopath="${CPREFIX :-/usr}/lib/toto" # chemin reel
        local vtotopath="/usr/lib/toto" # chemin virtuel pour les wrappers
        exeinto "${vtotopath}" # ajustement
        doexe "${FILESDIR}/toto-wrapper.sh"
        rm -fr "${D}${totopath}/plugins" # ajuste via ${totopath}
```

```
        [...]
}
```

Par ailleurs, un traitement spécifique à *CPREFIX* peut être nécessaire dans les fichiers de configuration de paquetages qui spécifient un chemin, voire éventuellement dans les codes sources eux-mêmes, si ceux-ci contiennent des chemins "en dur". Dans ce cas, on privilégiera autant que possible des ajustements dynamiques, par invocation de *sed* dans le *src_unpack()* ou *src_install()* pour prendre en compte tous les cas possibles (*CPREFIX* non définie, ou définie à une valeur arbitraire). Il est cependant acceptable de recourir à des fichiers de configuration et patchs statiques, pour des *ebuilds* systématiquement compilés comme paquetages secondaires, si une telle simplification facilite la lecture et la maintenance des codes.

Un dernier point à noter quant à la gestion de *CPREFIX* concerne la définition des chemins de recherche de bibliothèques. CLIP ne met pas en oeuvre de cache *ldconfig* qui permettrait la résolution automatique de bibliothèques installées dans */usr/local/lib*. Il est donc nécessaire, de compiler tout exécutable dépendant de telles bibliothèques de telle sorte qu'il inclue un champ *ELF DT_RUNPATH* permettant la recherche de bibliothèques dans des chemins supplémentaires. Ce positionnement du *RUNPATH* n'est pas réalisé automatiquement par *portage*, mais doit être fait explicitement en définissant la variable *LDFLAGS* à *"-Wl,-rpath,/usr/local/lib"* (et autres chemins complémentaires éventuels) dans le fichier *spec* de génération (cf. 3.1).

### 2.1.3 Installation de copies multiples

Il peut être nécessaire d'installer plusieurs copies des fichiers installés par un paquetage dans des répertoires différents, par exemple */vservers/rm_h* et */vservers/rm_b*. Les outils *portage* sont modifiés dans CLIP de manière à gérer de manière transparente ce cas de figure à travers une variable *CLIP_VROOTS*. Lorsque cette variable est définie comme une liste de un ou plusieurs chemins séparés par des espaces, le *src_install()* du paquetage est appelé plusieurs fois, sur des répertoires d'installation *${D}* correspondant chacun à la concaténation du *${D}* original (c'est-à-dire *${IMAGE}*, qui reste inchangé) et d'un des chemins de *CLIP_VROOTS*. Ainsi, avec par exemple une définition de *CLIP_VROOTS* à *"/vservers/rm_h /vservers/rm_b"*, *src_install()* sera appelée deux fois, avec respectivement *${D}* valant *"${IMAGE}/vservers/rm_h"* et *"${IMAGE}/vservers/rm_b"*, résultant dans l'installation des copies souhaitées des différents fichiers du paquetage.

La gestion de *CLIP_VROOTS* est intégrée dans les autres traitement réalisés par défaut par *portage* pour *CLIP*, en particulier la gestion des fichiers de configuration (cf. 2.5), et la création d'empreintes *veriexec* de bibliothèques (cf. 2.4). Pour cette dernière, des empreintes distinctes sont créées pour chacunes des copies de bibliothèques (par exemple deux pour */vservers/rm_h/lib/libtoto.so* et */vservers/rm_b/lib/libtoto.so*), et stockées dans un seul fichier d'empreinte, qui ne tient pas compte de *CLIP_VROOTS* (*/etc/verictl.d/<nom paquetage>* en l'occurence, ce qui est correct puisqu'un tel paquetage serait forcément un paquetage primaire de CLIP.

En revanche, un traitement spécifique de *CLIP_VROOTS* est nécessaire dans les autres fonctions *ebuild*, en particulier *pkg_predeb()*, qui ne sont quant à elles pas appelées automatiquement sur des *${D}* "virtuels". Ainsi, un paquetage créant des entrées *veriexec* d'exécutables, et ayant vocation à être utilisé avec *CLIP_VROOTS*, devra définir le *pkg_predeb()* suivant :

```
pkg_predeb() {
        for vroot in ${CLIP_VROOTS} ; do
                doverictld ${vroot}/sbin/toto <arguments...>
        done
}
```

## 2.2 Gestion des dépendances

La gestion des dépendances entre paquetages CLIP impose des contraintes spécifiques sur les *ebuilds*, aussi bien du fait de la transformation des dépendances *gentoo* en dépendances *Debian*, qui ne respectent pas nécessairement les mêmes conventions, qu'à cause de limitations spécifiques des outils de développement et de gestion des mises à jour CLIP.

### 2.2.1 Dépendances masquées dans CLIP : *clip-deps*, etc.

Un premier point concerne les dépendances entre paquetages secondaires et paquetages primaires. Les paquetages d'une distribution CLIP (*clip* ou *rm*) sont normalement répartis entre deux configurations : une configuration primaire, paquetage virtuel qui rassemble tous les paquetages primaires dans ses dépendances, et une configuration secondaire, qui rassemble de manière similaire tous les paquetages secondaires. Ces configurations doivent normalement être "complètes" au titre des dépendances. En d'autres termes, l'ensemble transitif des dépendances des paquetages dont dépend une configuration doit entièrement être compris dans les dépendances de la configuration elle-même. Cette contrainte résulte principalement d'une limitation des outils de développement de configuration d'une part (*clip-create-config, clip-config-template-editor*), et des outils de téléchargement de mises à jour d'autre part.

Il peut donc être nécessaire de supprimer toutes les dépendances portant sur des paquetages primaires, dans les *ebuilds* associés aux paquetages secondaires CLIP. Dans la mesure où certains paquetages qui sont primaires dans la distribution *clip* sont également secondaires dans la distribution *rm*, cette suppression de dépendance doit parfois être adaptée à la distribution pour laquelle chaque *ebuild* est compilé à un instant donné. A cette fin, plusieurs drapeaux *USE* sont réservés à la suppression de dépendances :

- *clip-deps* est défini pour toute compilation de paquetage CLIP, et permet de supprimer des dépendances envers des paquetages qui sont systématiquement primaires dans toute distribution CLIP, par une condition *!clip-deps ?*.
- *core-deps* est défini uniquement lors de la compilation de paquetages de la distribution *clip*, et permet de supprimer des dépendances envers des paquetages qui ne sont primaires que dans cette distribution.
- *rm-deps* est défini uniquement lors de la compilation de paquetages de la distribution *rm*, et permet de supprimer des dépendances envers des paquetages qui ne sont primaires que dans cette distribution.

Ainsi, si l'on considère un *ebuild* associé à un paquetage secondaire, déclarant dans ses *RDEPENDS zlib* (primaire aussi bien dans *clip* que dans *rm*), *openssl* (primaire uniquement dans *clip*), et *libX11* (secondaire dans les deux distributions), cet *ebuild* devra être modifié comme suit :

```
RDEPENDS="x11-libs/libX11                    (non modifie)
          !clip-deps? ( sys-libs/zlib)       (modifie)
          !core-deps? (dev-libs/openssl)"    (modifie)
```

On notera que d'autres drapeaux *USE* partagent ces conditions de définition, et que par exemple *clip, clip-core* et *clip-rm* pourraient être utilisés en lieu et place de *clip-deps, core-deps* et *rm-deps* respectivement. Cependant, il est important de bien utiliser ces derniers drapeaux pour la suppression de dépendances, et à cette fin uniquement, de manière à faciliter le rétablissement futur des dépendances concernées, lorsque les limitations des outils de développement et de téléchargement auront été levées.

Cette limitation est dans la pratique assez lourde à gérer dans les *ebuilds*, et est la seule raison de la présence de nombreux *ebuilds* dans *portage-overlay* plutôt que *portage*. Elle n'introduit cependant pas en général de réels problèmes dans la gestion de dépendances, dans la mesure où les dépendances ainsi exprimées sont systématiquement satisfaites dans CLIP. Une attention particulière à ces dépendances est néanmoins recommandée lors de l'élaboration des configurations.

**Remarque 2 : Conséquences de dépendances inter-configurations**
La publications d'une mise à jour CLIP contenant des dépendances inter-configurations (un paquetage secondaire dépendant d'une configuration ou d'un paquetage primaires) n'entraîne pas à proprement parler une erreur, mais est sous-optimale. La sous-optimalité est liée au fait qu'en cas de présence dans le miroir local d'un poste CLIP d'une configuration secondaire dépendant de paquetages primaires présents dans le miroir mais pas encore installés, l'installation de la configuration secondaire n'échouera qu'après vérification des signatures de l'ensemble de ses paquetages, opération coûteuse en temps de calcul, et que cet échec sera répété à chaque invocation périodique du client d'installation, jusqu'à mise à jour de la configuration primaire.

Par ailleurs, l'existence d'une dépendance d'un paquetage secondaire CLIP envers un paquetage primaire CLIP peut avoir également comme conséquence la non-détection du téléchargement d'une nouvelle configuration primaire CLIP par l'indicateur de redémarrage. L'utilisateur ne seront donc pas prévenu dans ce cas qu'il doit redémarrer son poste pour appliquer les mises à jour.

Les outils de mise à jour CLIP supportent en revanche un mécanisme simple permettant d'exprimmer des dépendances entre configurations, en limitant cette expression aux paquetages configuration eux-mêmes, cf. 3.4.3.

### 2.2.2 Contraintes sur les versions

Les *ebuilds gentoo* peuvent utiliser des versions comportant, en plus de la version numérique (*1.0.1, 20071227*, etc.), un suffixe de type *"_alpha", "_beta", "_pre"* (pré-versions) ou *"_pX"* (post-versions). Les suffixes de pré-versions sont problématiques après conversion en paquetage *debian*. En effet, selon les conventions de versions *debian*, *1.0.1beta* ou *1.0.1-beta* sont des versions postérieures à *1.0.1*. Ainsi, si une telle pré-version était installée sur un système CLIP, la version stable *1.0.1* ne serait jamais considérée comme une mise à jour de la version *1.0.1beta*, mais plutôt comme un retour en arrière [7]. Plus grave encore, la version *1.0.1* ne satisfera pas, pour les utilitaires *debian*, une dépendance du type *>= 1.0.1beta*, ce qui introduira tôt ou tard des blocages dans la gestion de mises à jour. Il est donc nécessaire d'éviter toute pré-version dans les paquetages *debian* pour CLIP. On notera au passage que les post-versions ne posent en revanche aucun problème : *1.0.1p1* est postérieur à *1.0.1* dans les conventions *debian*, tout comme *1.0.1_p1* est postérieur à *1.0.1* dans les conventions *gentoo*.

L'élimination des pré-versions doit être réalisées à deux niveaux : au niveau des *ebuilds* dont la version est susceptible d'être une préversion, et dans les dépendances des autres *ebuilds*. Idéalement, il est préférable de ne pas déployer du tout de pré-versions dans CLIP, et de préferer uniquement des versions finales, pour des raisons évidentes de stabilité. Dans ce approche, aucun *ebuild* dont la version est une pré-version n'est introduit dans les arborescences *portage*. Il reste néanmoins nécessaire de modifier les *ebuilds* qui mentionnent des pré-versions dans leurs dépendances, en remplaçant systématiquement les pré-versions par les versions stables correspondantes. Par exemple :

```
RDEPENDS=">=dev-libs/libtruc-1.0.1_beta2
          >=sys-apps/machin-2.0_pre3-r2"
```

devra idéalement être remplacé dans les *ebuilds* CLIP par :

```
RDEPENDS=">=dev-libs/libtruc-1.0.1
          >=sys-apps/machin-2.0"
```

Cependant, il peut arriver qu'il soit incontournable d'intégrer dans CLIP des paquetages disponibles uniquement en versions *beta, pre* voire *alpha*. Un cas typique (tiré de faits réels) pourrait par exemple prendre la forme suivante :

- Une mise à jour de *mozilla-firefox* de la version *2.0.0.** vers la version *3.0.5* est nécessaire pour des raisons de sécurité.
- La version *3.0.5* dépend de *dev-libs/nss* en version supérieure ou égale à *3.12.2_beta*. Toute version précédente cause un *bug* dans *firefox*.
- Or aucune version stable *3.12.2* n'a été publiée par les auteurs de *mozilla*. La *3.12.2_beta* est donc la seule convenable.

Dans ce cas, il est nécessaire pour CLIP de trouver une autre représentation de la notion de pré-version, qui ne fasse pas apparaître de mots clés *"_beta"* ou autres. Plus spécifiquement, la solution recommandée consiste à affecter un autre numéro de version à la pré-version, qui soit "final" (sans suffixe de pré-version), tout en maintenant l'ordre des différentes versions. Ainsi, en reprenant l'exemple *firefox* décrit ci-dessus, il est nécessaire d'introduire pour CLIP une version *3.12.1.90* de *nss*, correspondant à la version publique *3.12.2_beta*. Un tel numéro de version est adapté, dans la mesure où il est postérieur à toute version *3.12.1* publiée, antérieur à toute version stable *3.12.2* ou ultérieure, et laisse la possibilité d'introduire des versions correspondant à d'autres pré-versions ultérieures (*3.12.1.91, 3.12.1.92*, etc.). Ainsi, l'*ebuild nss-3.12.2_beta.ebuild* est importé dans CLIP sous le nom *nss-3.11.1.90.ebuild* (ce qui nécessite une modification du *SRC_URI* de l'*ebuild*, pour continuer à utiliser le *distfile 3.12.2_beta*), et les dépendances dans *mozilla-firefox-3.0.5.ebuild* sont transformées de :

```
>=dev-libs/nss-3.12.2_beta
```

7. Cette première difficulté ne se pose réellement que pour les paquetages optionnels supportés par les systèmes de gestion de mises à jour CLIP récent, dans la mesure où la mise à jour impérative des seules configurations ne laisse pas l'installeur juger de la pertinence d'une mise à jour.

en

```
>=dev-libs/nss-3.12.1.90
```

### 2.2.3 Gestion des *SLOTS*

La gestion des *SLOTS* constitue l'une des principales fonctionnalités de *portage* qui ne peuvent pas être reprises directement dans les paquetages binaires CLIP. La notion de *SLOT* permet dans *portage* de distinguer deux versions d'un même paquetage comme occupant des emplacements distincts, et pouvant par conséquent être installés simultanément au sein d'un système, sans conflit, ni remplacement d'une version par l'autre. Ainsi, les *ebuilds* pour les paquetages *x11-libs/qt-2* et *x11-libs/qt-3* occupent deux *slots* différents, respectivement 2 et 3, dans la mesure où ils sont installés dans deux chemins entièrement distincts (*/usr/qt/2* et */usr/qt/3*), et peuvent donc être installés côte à côte.

La gestion de paquetages *DEBIAN* ne permet pas de tel mécanisme : une seule version d'un paquetage donné peut être installée à un moment donné, l'installation d'une version différente remplaçant systématiquement l'ancienne version. La seule manière de gérer l'installation simultanée de deux versions d'un même paquetage consiste en fait à conférer artificiellement des noms de paquetage distincts aux deux versions. On pourra par exemple, dans le cas de *x11-libs/qt* présenté plus haut, nommer les paquetages *qt2* (version *2.2.1* par exemple) et *qt3* (version *3.3.6* par exemple), ou encore *qt2* et *qt*. Cette approche doit aussi être mise en oeuvre dans CLIP lorsque l'installation simultanées de deux versions d'un même paquetage est nécessaire [8] A cette fin, *portage* reconnaît, lorsqu'il est patché pour CLIP, une variable d'environnement *DEB_NAME_SUFFIX*, dont la valeur, lorsqu'elle est définie, est ajoutée à la fin du nom de paquetage *debian*, lorsqu'un tel paquetage est généré. Cette variable est mise en oeuvre par exemple pour le paquetage *app-crypt/gnupg*, qui doit être installé simultanément en version *1.4.** (*SLOT=1* dans *gentoo*) et *1.9.** (*SLOT=2* dans *gentoo*). La version *1.4.** est compilée pour CLIP avec *DEB_NAME_SUFFIX=1*, ce qui crée un paquetage *debian gnupg1*, tandis que la version *1.9.** est compilée sans cette définition, pour obtenir un paquetage *gnupg*. On obtient ainsi deux paquetages *debian*, *gnupg1* et *gnupg*, qui peuvent être simultanément installés, sous réserve que leurs fichiers respectifs ne causent pas de conflit (ce qui est normalement garanti lorsque les *ebuilds* utilisent des *SLOTS* différents). La variable *DEB_NAME_SUFFIX* peut être définie aussi bien dans l'*ebuild* lui-même que dans le *spec xml* utilisé pour le compiler (cf. 3.1).

Un autre aspect problématique de la gestion des *SLOTS* concernent leur apparition éventuelle dans les dépendances d'un paquetage. Certains *ebuilds* récents issus de *gentoo* spécifient ainsi les *SLOTS* requis de certaines de leurs dépendances. Par exemple, un paquetage *kdelibs-3.** pourra avoir dans ses dépendances l'élément *x11-libs/qt :3* pour préciser que cette dépendance n'est satisfaite que par un paquetage *qt* occupant le *SLOT 3*. Lors de la génération de paquetage *debian*, le script *gencontrol.pl* utilisé dans CLIP ignore systématiquement cette spécification de *SLOT*, remplaçant silencieusement *x11-libs/qt :3* par exemple par *x11-libs/qt*. On perd ainsi une partie de l'information d'origine. Cette information peut être rajoutée au besoin par une dépendance spécifique à CLIP, par exemple *clip ? (x11-libs/qt3)*, si l'on suppose les paquetages *qt* en version *3* générés pour CLIP avec *DEB_NAME_SUFFIX=3*.

### 2.2.4 Autres contraintes sur les dépendances

De manière comparable à la gestion des *SLOTS*, *gentoo* permet également de définir des versions "*wildcards*" dans les dépendances d'un paquetage. Par exemple, une dépendance *=x11-libs/gtk+-2** sera satisfaite par toute version de *gtk+* commençant par le chiffre 2, mais pas par *gtk+-1.2* ou un hypothétique *gtk+-3.0*. De telles dépendances ne peuvent en revanche pas être définies dans un paquetage *debian*. Ainsi, les scripts utilisés dans CLIP pour la génération de paquetages *debian* simplifient automatiquement une telle dépendance en

8. Ce qui est en particulier le cas lorsqu'une mise à jour majeur d'une bibliothèque utilisée par de nombreux paquetages s'accompagne d'un changement d'interface binaire (ABI). Dans ce cas, l'ancienne version (par exemple *libpng12.so*) doit être conservée aux côtés de la nouvelle (*libpng14.so*) le temps que tous les paquetages utilisant l'ancienne version aient été recompilés et mis à jour.

*>=x11-libs/gtk+2*, condition qui est encore satisfaite par toute version de *gtk+* commençant par le chiffre 2, et pas par *gtk+-1.2*, mais qui, à la différence de la dépendance d'origine, est également satisfaite par *gtk+-3.0*. Ce relachement des contraintes ne pose en général pas de problème dans la pratique. Au besoin, il peut être palié de la même manière que celui lié au support des *SLOTS*, par des dépendances spécifiques à CLIP, et l'utilisation de *DEB_NAME_SUFFIX*.

Enfin, on notera aussi que *portage* supporte un type de dépendance spécifique *PDEPEND*, correspondant aux dépendances nécessaires au fonctionnement du paquetage considéré, mais qui doivent préférentiellement être installées après ce dernier. De telles dépendances ne peuvent là encore pas être définies dans un paquetage *debian,* et *gencontrol.pl* est ainsi contraint à une autre simplification, consistant à traiter les *PDEPEND* comme des *DEPEND* classiques, c'est-à-dire des dépendances à installer avant le paquetage. Cette simplification n'est en général pas problématique dans la mesure où le *PDEPEND gentoo* traduit généralement un ordre imposé par la compilation [9], problème qui ne se pose pas lors de la distribution de paquetages binaires.

### 2.2.5 Utilisation de paquetages virtuels

Ce dernier point ne constitue pas une contrainte, mais plutôt une recommendation. L'emploi de paquetages virtuel peut largement simplifier certaines problématiques de gestion de dépendances. Un paquetage virtuel est un paquetage qui n'installe aucun fichier, mais qui permet de satisfaire des dépendances. Un tel paquetage peut typiquement être utilisé pour satisfaire virtuellement une dépendance satisfaite indifféremment par différent paquetages non virtuels. Par exemple, si une dépendance peut être satisfaite indifféremment par les paquetages *foo, bar* et *baz*, il peut être intéressant de définir un paquetage virtuel *quux*, qui est automatiquement fourni par l'installation de n'importe lequel des trois paquetages, et qui peut être utilisé pour définir la dépendance correspondante dans d'autres paquetages, de manière plus simple qu'une disjonction *foo || bar || baz* (et sans avoir besoin de modifier tous les paquetages qui définissent une telle dépendance si un quatrième paquetage satisfaisant la même dépendance est ajouté à la distribution).

La notion de paquetage virtuel est présente aussi bien dans *debian* que dans *gentoo*, mais avec une différence conceptuelle :

- Un paquetage virtuel *debian* n'est jamais installé, il n'a pas de version et ne correspond pas à un paquetage *.deb*. Un tel paquetage n'est manipulé qu'à travers les champs *Provides :* de paquetages non virtuels installés sur le système.
- La notion de paquetage virtuel *gentoo* recouvre deux concepts. Un premier type de paquetage virtuel est comparable aux paquetages virtuels *debian*. Ces paquetages sont manipulés uniquement à travers les champs *PROVIDE* des *ebuilds* de paquetages non virtuels, et ne se voient pas associer d'*ebuild* spécifique. Ils ne sont donc jamais installés directement sur le système. Par ailleurs, un deuxième type de paquetage virtuel se voit quant à lui associer un *ebuild* spécifique, dans la catégorie *virtual* (par exemple, *virtual/x11*). Les *ebuilds* de la catégorie *virtual* n'installent aucun fichier, mais se voient associer une version et des dépendances comme tout autre *ebuild*.

Le premier type de paquetage virtuel *gentoo* est converti automatiquement en paquetage virtuel *debian*, par la transformation des *PROVIDE* d'*ebuilds* en *Provides : debian*. Le second type est converti comme un *ebuild* classique, avec pour résultat un paquetage *debian* installable, mais qui n'installe aucun fichier dans le système. Les deux types de paquetages virtuels sont utilisables indifféremment dans CLIP, en tenant compte des différences suivantes :

- Un paquetage virtuel sans *ebuild* est plus simple à créer, et ne nécessite pas le téléchargement et la vérification de signature d'un paquetage vide par chaque poste client.
- Seul un paquetage virtuel associé à un *ebuild* de la catégorie *virtual/* peut se voir affecter une version. Ainsi, cette deuxième approche est la seule utilisable lorsqu'il est nécessaire de spécifier une dépendance précise envers une version spécifique de paquetage virtuel. On notera cependant que pour les cas

9. Par exemple, *x11-base/xorg-server* dépend en *PDEPEND* de *x11-term/xterm*, car il a besoin de ce dernier pour fonctionner nominalement, mais ne peut pas le déclarer en *DEPEND* car *xterm* ne compilera pas si *xorg-server* n'est pas installé.

simples, une notion de version peut aussi être intégrée au nom d'un paquetage virtuel (par exemple, *x11v7* pour exprimmer la dépendance envers un serveur X11 supportant la version 7 du protocole).

Un cas d'emploi typique de paquetages virtuels au sein du système CLIP concerne la gestion des différences entre configuration du système CLIP, correspondant à des paquetages différents. Par exemple, certains scripts de configuration réseau sont intégrés à des paquetages spécifiques chacun à une configuration de système CLIP : *app-clip/clip-net* pour les clients CLIP-RM, *app-clip/clip-gtw-net* pour les passerelles CLIP, etc. Afin de simplifier la maintenance, les *ebuilds* qui dépendent d'une version particulière des scripts de configuration réseau ne dépendent pas directement de ces paquetages, mais plutôt d'un paquetage virtuel *virtual/clip-net-virtual*, associé à une version précise. L'*ebuild* correspondant dans *virtual/clip-net-virtual* est à son tour le seul à dépendre directement des différents *clip-*net* spécifiques, et serait le seul à modifier en cas d'ajout d'une nouvelle configuration et du paquetage réseau associé.

### 2.3 Scripts d'installation : *maintainer-scripts*

Les paquetages *debian* sont susceptibles d'inclure des scripts appelés *maintainer scripts*, appelés automatiquement à différents stades (fonction de leur nom : *postinst, prerm*, etc.) de l'installation ou de la désinstallation du paquetage concerné. L'ajout de tels scripts à un paquetage est simple : il suffit de les copier dans le répertoire *DEBIAN/* de l'arborescence de fichiers du paquetage, avant de créer ce dernier par une commande *dpkg -b* par exemple. Les *ebuilds gentoo* permettent des traitements similaires, sur la base non pas de scripts, mais de fonctions *pkg_preinst()* et *pkg_postinst()* éventuellement définies par chaque *ebuild* (cf. 1.2). Cependant ces traitements ne sont pas automatiquement convertis en *maintainer scripts* par *portage* CLIP lors de la création de paquetage *debian*. Une telle conversion n'est d'ailleurs pas souhaitable en général, car les traitements effectués en *preinst()* et *postint()* peuvent être très spécifiques à *gentoo* et non applicables à CLIP (cas par exemple d'appels à *eselect* pour ajuster la configuration locale, alors qu'*eselect* n'est pas employé sous CLIP). Les différents traitement réalisés en *pkg_preinst()* et *pkg_postinst()* n'ont en réalité aucune incidence sur la création de paquetage *debian*.

Il est de ce fait nécessaire de créer spécifiquement les *maintainer scripts* des paquetages CLIP, lorsque le besoin se présente. A cette fin, deux approches sont possibles :

- **Une approche statique** : des scripts peuvent être copiés directement dans le répertoire *${FILESDIR}/_debian* de l'*ebuild*. Le contenu de ce répertoire est intégralement copié dans le répertoire *${D}/DEBIAN* avant la génération d'un paquetage *debian*. Ainsi, un fichier *postinst* placé dans ce répertoire se retrouverait automatiquement intégré comme script *postinst* du paquetage *debian*.
- **Une approche dynamique** : des scripts peuvent être créés et dynamiquement écrits dans *${D}/DEBIAN* lors du traitement de la fonction *pkg_predeb()* de l'*ebuild*. Cette écriture de scripts n'est par construction réalisée que lors de la création d'un paquetage *debian*, et non lors de l'installation locale sur un poste de développement.

La deuxième approche, qui peut sembler plus complexe (notamment du fait de la nécessité d'échapper certains caractères, comme décrit plus bas), mais est la seule qui permette de gérer certaines problématiques liées au comportement paramétrable des *ebuilds*. Ainsi, si le traitement réalisé en *postinst* du paquetage *debian* dépend de la définition ou non d'un drapeau *USE* auquel est sensible l'*ebuild*, il sera nécessaire d'écrire ce script *postinst* (ou du moins cette portion du script) dynamiquement, en fonction du *USE* effectif lors de la génération du paquetage. De même, la prise en compte d'un chemin d'installation variable (par exemple du fait des différentes valeurs possibles de *CPREFIX*) nécessite la génération dynamique des *maintainer-scripts*.

On notera que l'utilisation simultanées des deux approches est possible, dans la mesure où le contenu de *${FILESDIR}/_debian/* est copié dans *${D}/DEBIAN* avant l'appel à *pkg_predeb()*. Il est donc possible, dans cette dernière fonction, de compléter dynamiquement un *maintainer-script* statiquement défini. Dans un tel cas de figure (et plus généralement, pour toute génération dynamique de *maintainer-script*, afin de faciliter la maintenance), l'utilisation de la fonction *init_maintainer* de l'*eclass deb.eclass* est impérative. Cette fonction est appelée sous la forme suivante, dans *pkg_predeb()* :

```
init_maintainer <nom script>
```

avec *<nom script>* égal à *"postinst"* par exemple. Cette fonction crée un fichier sous le chemin *${D}/DEBIAN/<nom script>*, avec les droits adaptés, si ce fichier n'est pas présent. Dans ce cas, elle ajoute également les lignes suivantes en tête du fichier :

```
# !/bin/sh
set -e
```

Lorsque le fichier considéré existe déjà, la fonction ne réalise aucun traitement. Elle se prête donc aussi bien au cas où des scripts statiques sont utilisés qu'au cas purement dynamique. Après un tel appel, le script pourra être complété, typiquement par des commandes *cat >> ${D}/DEBIAN/<script> << ENDSCRIPT*, suivies d'un *here-document bash* terminé par *ENDSCRIPT*. On prendra garde dans ce cas à la nécessité d'échapper certains caractères par un '\', notamment $ (pour la résolution de variable dans le script, et non dans l'*ebuild* lui-même), ou \, comme dans l'exemple suivant :

```
cat >> "${D}"/DEBIAN/postinst << ENDSCRIPT
        var="\$(cat /etc/toto.conf)"
        sed -i -e "s/VAR/\${var}/g' "${CPREFIX}/etc/toto.local.conf"
ENDSCRIPT
```

(on notera l'échappement de $ devant *$(cat* et *${var}*, mais pas devant *${CPREFIX}*, qui doit être résolu dans l'*ebuild* et non dans le script qu'il génère.

Par ailleurs, il est important de prendre en compte les contraintes et recommendations suivantes concernant l'écriture de *maintainer scripts* dans CLIP :

- Sauf pour ce qui concerne les paquetages primaires du socle CLIP, les *maintainer scripts* CLIP sont appelés avec un */bin/sh* correspondant au *shell ash busybox* (cf. [BUSYBOX]), plutôt qu'à *bash*. De ce fait, toute construction syntaxique avancée propre à *bash* est à proscrire dans de tels scripts.
- La lecture de paramètres modifiables par l'administrateur local est soumise aux mêmes contraintes d'import sécurisé que dans les scripts de démarrage du système (cf. [CLIP 1301]), sous peine de créer des possibilités d'escalade de privilèges violant le modèle de sécurité CLIP. En particulier, de telles lecture devront être réalisée à l'aide des fonctions *import_conf()* et autres de */lib/clip/import.sub*, ou être validées de manière équivalente par un traitement spécifique.
- Les *maintainer scripts* doivent normalement commencer par un *set -e*, afin de garantir une sortie en erreur immédiate pour tout cas d'erreur non pris en compte. La fonction *init_maintainer* ajoute automatiquement cette commande en début de script. On prendra néanmoins garde aux contraintes de codage qu'impose ce mode : toute commande qui peut retourner un code d'erreur non nul doit faire l'objet d'un traitement spécifique (soit sous la forme *<cmd> || <traitement>*, soit sous la forme *set +e ; <cmd> ; if [[ $ ? -ne 0 ]] ; then <traitement> fi ; set -e*).
- Contrairement à l'installation d'un paquetage *debian* classique, il n'y a généralement personne pour lire la sortie standard des *maintainer scripts* lors de leur exécution sous CLIP. Ainsi, tout message qui doit être signalé aux utilisateurs doit être transmis au système d'audit par une commande *logger*, plutôt que seulement écrit sur la sortie standard par un *echo*.

## 2.4 Gestion de *veriexec*

La mise en oeuvre du sous-système *veriexec* de CLIP LSM ([CLIP 1201]) nécessite la définition, sur les postes CLIP, d'entrées *veriexec*, associant un exécutable ou une bibliothèque à son empreinte cryptographique et à un certains nombre de propriétés (privilèges, options). Ces entrées *veriexec* sont intégrées aux paquetages binaires CLIP, sous la forme de fichier de configuration qui peuvent ensuite être chargés dans la base d'entrées *veriexec* du noyau, à l'aide d'appels à l'utilitaire *verictl*. Les différentes entrées associées à un paquetage sont réunies dans un unique fichier de définition d'entrées, intégré au paquetage et installé par ce dernier sous le chemin */etc/verictl.d/<pkg>* pour un paquetage primaire, ou */usr/local/etc/verictl.d/<pkg>* pour un paquetage

secondaire, où *<pkg>* représente le nom du paquetage. Les différents fichiers de définition d'empreintes ainsi créés sont automatiquement chargés par *verictl* lors du démarrage du système (cf. [CLIP 1301]). En revanche, la mise à jour de la base d'entrées *veriexec* lors d'une mise à jour en ligne de paquetage secondaire est de la responsabilité du paquetage concerné, qui doit réaliser cette opération dans ses *maintainer scripts*.

Ainsi, l'intégration d'une empreinte *veriexec* dans un paquetage nécessite deux actions spécifiques :

- Le calcul d'une empreinte cryptographique pour l'exécutable concerné, après la génération de ce dernier lors de la compilation du paquetage, et la création de la définition d'entrée *veriexec* dans *${CPREFIX}/etc/verictl.d/<pkg>* dans l'arborescence de fichiers du paquetage. On notera que le calcul de l'empreinte cryptographique doit être réalisé à un stade de la procédure de création de paquetage tel que l'exécutable ne soit plus modifié avant son inclusion dans le paquetage. En particulier, ce calcul pourra être réalisé dans le traitement de *pkg_predeb()* (cf. 1.2), mais pas par exemple dans celui de *src_install()*, dans la mesure où *strip* est appelé sur les exécutables après *src_install()* et modifie les empreintes d'exécutables.
- La création de *maintainer scripts* permettant de supprimer les entrées associées au paquetage lors de sa désinstallation, et de charger ses empreintes lors de son installation. Ces opérations sont typiquement réalisées dans les scripts *prerm* et *postinst* du paquetage, respectivement. On notera que ce traitement n'est nécessaire que lors de la mise à jour ou de l'installation en ligne d'un paquetage secondaire, mais pas lors de l'installation initiale (pendant laquelle le noyau utilisé est celui du support d'installation, qui n'intègre généralement pas de sous-système *veriexec*), ni lors de la mise à jour de paquetages primaires (pour lesquels le système ou la cage concernée sont redémarrés avant l'utilisation des exécutables mis à jour, ce qui garantit un rechargement automatique des entrées *veriexec*). Afin d'éviter des opérations *veriexec* inutiles, voire non supportées, les *maintainer scripts* de recharchement d'entrées *veriexec* ne doivent réaliser leur traitement que lorsque la variable d'environnement *BOOTSTRAP_NOVERIEXEC* n'est pas définie. Cette variable est définie automatiquement lors de l'installation initiale et de la mise à jour de paquetages primaires du système.

Les outils *portage* modifiés pour CLIP répondent à cette nécessité de créer des empreintes *veriexec* de deux manières complémentaires. D'une part, des entrées *veriexec* sont créées automatiquement pour toutes les bibliothèques dynamiques partagées de chaque paquetage (fichiers **.so* dans l'arborescence du paquetage). Cette création est rendue automatique dans la mesure où les bibliothèques ne se voient pas attribuer de privilèges spécifiques par leurs entrées *veriexec*, contrairement aux exécutables. D'autre part, une *eclass* spécifique, *verictl.eclass*, permet aux développeurs de définir au cas par cas des entrées *veriexec* pour les exécutables privilégiés qui nécessitent une telle empreinte. Dans les deux cas, les deux aspects de la création d'entrée sont traités simultanéments : une empreinte est calculée pour chaque fichier concerné et ajoutée aux fichier de définition d'entrées du paquetage d'une part, et des commandes sont ajoutés d'autre part aux scripts *prerm* et *postinst* du paquetage (qui sont créés au besoin), pour respectivement décharger ces entrées et les recharger lors de la désinstallation et de l'installation du paquetage, lorsque *BOOTSTRAP_NOVERIEXEC* n'est pas définie.

La génération d'entrées pour les bibliothèques est réalisée automatiquement lors de la création de paquetage *debian*, avant l'appel au *hook pkg_predeb()* de l'*ebuild* concerné, dès lors que la variable d'environnement *VERICTL_DOSHLIBS* est définie à une valeur non nulle (ce qui est généralement assuré par le fichier *spec* utilisé pour la génération, cf. 3.1) Cette génération utilise par défaut la fonction de hachage *ccsd*, qui nécessite l'installation sur le poste de l'utilitaire *ccsd-hash* (*clip-dev/ccsd-utils*). Le contexte *veriexec* (correspondant au contexte *vserver* d'utilisation de ces entrées, cf. [CLIP 1201]) pour lequel ces entrées sont créées est par défaut *-1*, ce qui signifie que le contexte sera défini par celui de l'appel *verictl* réalisant le chargement de l'entrée. Ce comportement est adapté dans la plupart des cas. Cependant, il est possible de spécifier explicitiment un contexte *veriexec* pour ces créations automatiques d'entrées, en définissant la variable d'environnement *VERIEXEC_LIB_CTX* au numéro de ce contexte (définition qui peut être réalisée dans l'*ebuild* ou dans le fichier *spec xml*).

La génération d'entrées spécifiques pour les exécutables peut être réalisée explicitiment dans le *pkg_predeb()* des *ebuilds*, en faisant appel à la fonction *doverictld* de *verictl.eclass*. Le "prototype" de cette fonction est le suivant :

```
doverictld <fichier> <options> <cap eff> <cap perm> <cap inh> <algo> <privs>
```

avec :

- *<fichier>* le chemin "absolu" du fichier pour lequel l'empreinte doit être créée, dans l'arborescence temporaire d'installation (par exemple */usr/local/foo* pour un fichier temporairement installé dans *${D}/usr/local/foo*).
- *<options>* les options *veriexec* de l'entrée, sous la forme d'une concaténation de lettres clés (cf. [CLIP 1201]), par exemple *er* pour une entrée applicable à un exécutable, uniquement lors d'une exécution par *root*.
- *<cap eff>*, *<cap perm>* et *<cap inh>* les masques de capacités effectif, permis et héritable à attribuer à l'exécutable, sous forme numérique, par exemple *0x40000* pour *CAP_SYS_CHROOT*. Il est rappelé que ces masques sont automatiquement limités par les masques maximaux autorisés du contexte *veriexec* cible lors du chargement de l'entrée.
- *<algo>* l'algorithme à utiliser pour créer l'empreinte cryptographique, par exemple *ccsd* ou *sha256*. La fonction de hachage *ccsd* est à privilégier.
- *<privs>* les privilèges CLSM à attribuer à l'exécutable, sous la forme d'une concaténation de lettres clés, par exemple *cN* pour *CLSM_PRIV_NETCLIENT* et *CLSM_PRIV_NETLINK*. Ce paramètre peut être laissé vide, ou défini comme *'-'*, si aucun privilège ne doit être attribué.

Tout comme pour la création automatiques des entrées de bibliothèques, ces entrées sont crées par défaut avec le contexte -1, et un contexte spécifique peut être défini par l'intermédiaire de la variable d'environnement *VERIEXEC_CTX*. Par exemple, pour attribuer le privilège d'accès réseau *CLSM_PRIV_NETCLIENT* à */usr/bin/wget* dans le contexte $\text{UPDATE}_{\text{clip}}$ (501), on pourra utiliser les commandes suivantes dans l'*ebuild wget* :

```
pkg_predeb() {
        VERIEXEC_CTX=501 doverictld /usr/bin/wget e 0 0 0 ccsd c
        [... autres traitements ...]
}
```

Les appels à *doverictld* n'ont pas d'effet sur la création automatique d'entrées *veriexec* pour les bibliothèques, qui est réalisée avant *pkg_predeb()*. Ils sont aussi compatibles avec la création de *maintainer-scripts* pour d'autres traitements, aussi bien de manière statique que dynamique (cf. 2.3).

## 2.5 Gestion des fichiers modifiables localement

Au sein d'un système CLIP, un certain nombre de fichiers de configuration, initialement installés par des paquetages, sont éditables par l'administrateur local. Les paquetages associés contiennent une version par défaut de ces fichiers, généralement modifiée ensuite de manière spécifique à chaque poste. Il est nécessaire d'adopter un traitement spécifique de ces fichiers, afin de répondre à deux objectifs :

- L'installation initiale des paquetages concernés doit créer les fichiers de configuration dans leur version par défaut, mais les installations ultérieures (dans le cadre de mises à jour) ne doivent pas écraser les fichiers existants, susceptibles de contenir des modifications locales.
- Les fichiers de configuration modifiés localement doivent être recopiés dans le nouveau jeu de partitions lors d'une mise à jour du coeur du système CLIP, qui s'accompagne du basculement sur un jeu de partitions système, afin d'éviter que la mise à jour n'entraîne la perte des dernières modifications de ces fichiers.

Un tel traitement adapté est possible dans un *ebuild* CLIP grâce à une variable spécifique à CLIP, *CLIP_CONF_FILES*. La liste des fichiers installés par le paquetage et susceptibles d'être ensuite modifiés localement doit être affectée à cette variable (fichiers décrits par leurs chemins complet, par rapport à la racine de l'arborescence temporaire d'installation *${D}*, et séparés par des espaces ou retour à la ligne). Pour chacun de ces fichiers, les opé-

rations suivantes sont réalisées lors de la génération du paquetage debian (juste avant l'appel à *pkg_predeb()*, cf. 1.2.1) :

- Renommage des fichiers de configuration, de manière à ne pas écraser les fichiers existant sur le système cible. La convention de nommage est la suivante :

```
/chemin/vers/fichier => /chemin/vers/.fichier.confnew
```

- Génération (ou ajout à des scripts préexistants, le cas échéant) de scripts *Debian postinst* et *prerm*, assurant les traitements décrits décrits ci-dessous.

Le script *postinst* généré à cette occasion réalisera lors de l'installation les opérations suivantes :

- Si aucune version locale du fichier n'existe, le fichier fournit par le paquetage est renomé pour reprendre son nom d'origine, et ainsi créer la version par défaut du fichier de configuration. Au contraire, si une version locale du fichier existe, le fichier fournit par le paquetage est comparé à celle-ci, puis supprimé s'il lui est identique et conservé sinon (de manière à permettre l'analyse éventuelle des différences par un administrateur local).
- Le nom original du fichier (sans modification) est ajouté (s'il n'y est pas déjà présent) à la liste */etc/admin/clip_install/conffiles.list*, qui définit les fichiers qui seront automatiquement copiés dans le nouveau jeu de partition lors d'une mise à jour du coeur du système.

Le script *prerm* assure quant à lui uniquement la suppression du nom du fichier de la liste */etc/admin/clip_install/conffiles.list*. On notera que lors de la suppression, les fichiers de configuration effectivement installés par le paquetage (c'est-à-dire ceux dont le nom contient *.confnew*) sont supprimés automatiquement s'ils sont encore présents. En revanche, les versions locales de ces fichiers (sans *.confnew*) ne sont jamais supprimées, même lorsqu'elles ont été créées par le paquetage et n'ont fait l'objet d'aucune modification locale.

On notera que la mise en oeuvre de *CLIP_VROOTS* (cf. 2.1.3) est transparente du point de vue de la gestion des fichiers de configuration : la variable *CLIP_CONF_FILES* n'a pas besoin d'être modifiée pour prendre en compte les différentes racines d'installation.

Une seconde variable peut par ailleurs être utilisée, d'une manière similaire à celle associée à *CLIP_CONF_FILES*, pour gérer les fichiers de configuration modifiables localement mais qui ne sont pas installés par un paquetage quelconque (leur version initiale étant typiquement créée par l'installeur CLIP lors de l'installation du poste). Ces fichiers peuvent être déclarés dans la variable *CLIP_CONF_FILES_VIRTUAL* d'un *ebuild* quelconque (typiquement celui qui utilise ces fichiers de configuration), de manière à ce que l'installation du paquetage ainsi généré ajoute les chemins des fichiers à la liste */etc/admin/clip_install/conffiles.list*, et que la désinstallation du paquetage les en supprime. Les autres traitements associés à *CLIP_CONF_FILES* (renommage de fichiers, comparaison à l'installation) ne sont en revanche évidemment pas réalisés dans ce cas.

# 3 Génération de paquetages et de configurations

La génération de paquetages binaires CLIP (au format *debian*), est réalisée à l'aide de *portage*. Cependant, la génération par un appel direct à la commande *emerge* est peu souhaitable car source d'erreurs, étant donné la multiplicité des paramètres (drapeaux *USE* et *FEATURES*, variables d'environnement, etc.) à ajuster. Il est nettement préférable d'utiliser à cette fin les outils de génération spécifiques à CLIP : *clip-build*, un outil permettant de lancer des commandes *emerge* dans un environnement maîtrisé, et *clip-compile*, un *wrapper* simple construit autour de *clip-build*. Par ailleurs, une fois des paquetages binaires CLIP correctement générés, leur déploiement nécessite un certain nombre d'opérations complémentaires, notamment leur référencement dans un paquetage spécifique dit **configuration** et la création d'un miroir au format adapté.

## 3.1 Fichiers *spec*

### 3.1.1 Principe des fichiers *spec*

La génération de paquetages binaires à l'aide de *clip-build* ou *clip-compile* est pilotée par un fichier de configuration en format XML, dit fichier *spec*. Chaque distribution CLIP (*clip* et *rm* pour CLIP-RM, *clip* pour CLIP-GTW) est normalement associée à un fichier *spec* dédié, géré en configuration dans le répertoire *specs* de la branche considérée du *repository clip-int*.

Le format du fichier *spec* est décrit en détail dans [CLIP 1101]. Ce format permet une définition d'options en cascade : options globales de l'ensemble du fichier, options spécifiques à un groupe de paquetages <*conf*> ou à un sous-groupe <*pkg*>. Chaque bloc <*pkg*> peut référencer un ou plusieurs paquetages dans un élément <*pkgnames*> - la compilation de tous les paquetages de ce bloc sera alors réalisée avec la combinaison des options propres au bloc, de celles définies par le bloc <*conf*> courant, et des options globales du fichier. Le regroupement des paquetages au sein de blocs <*pkg*>, et le regroupement de ces blocs au sein de blocs <*conf*>, ne sont pas soumis à des règles strictes, mais guidés en pratique par un principe de factorisation des options.

Les options les plus souvent ajustées sont celles définies dans les éléments suivants

- <*use*>...</*use*> : définition de la variable *USE* passée à *portage*.
- <*ldflags*>...</*ldflags* : définition des *LDFLAGS* passés à l'éditeur de liens statique, notamment utile pour préciser les chemins *RPATH* associés à l'utilisation de *CPREFIX* (cf. 2.1.2) ou à un chemin d'installation de bibliothèques particulier (cf. remarque ci-dessous).
- <*env*>...</*env*> : définition de variables d'environnement spécifiques, sous la forme d'une liste de couples *VARIABLE=valeur* séparés par des virgules, ce qui permet de définir les différentes variables d'environnement spécifiques à la génération de paquetages CLIP (*CPREFIX*, *DEB_NAME_SUFFIX*, etc. - cf. liste exhaustive en 3.2.3).
- <*cflags*>...</*cflags*> : définition des *CFLAGS* applicables à la compilation - cette variable est généralement positionnée globalement pour l'ensemble du fichier *spec*, et il est rare qu'il soit opportun de la modifier spécifiquement pour un paquetage [10].
- <*features*>...</*features*> : définition de la variable *FEATURES* passée à *portage*, pour par exemple demander la suppression automatique des fichier d'en-tête (*noinclude*), des bibliothèques statiques (*nostatlib*) ou de la documentation et des pages *man* et *info* (*noman noinfo nodoc*). Ces *FEATURES* sont généralement ajustées avec une granularité correspondant au bloc <*conf*> plutôt qu'au bloc <*pkg*>.

**Remarque 3 : Gestion des chemins d'installation des bibliothèques via les *LDFLAGS***

Le système CLIP ne met pas en oeuvre de cache de résolution de bibliothèques dynamiques (*ld.so.cache*). La résolution des chemins de bibliothèques se fait donc de manière purement statique. Chaque bibliothèque est automatiquement recherchée dans les chemins par défaut */lib* et */usr/lib*. La recherche dans un autre chemin, par exemple */usr/local/lib* ou un sous-répertoire de ce dernier, nécessite en revanche une adaptation spécifique de chaque binaire (exécutable ou bibliothèque dynamique) utilisant la bibliothèque recherchée, afin d'inclure ce ou ces chemins dans la directive *RPATH* du segment dynamique des fichiers ELF correspondant. Le positionnement

10. Si des ajustements de *CFLAGS* spécifiques sont nécessaires à la compilation d'un *ebuild* particulier, ces derniers devront plutôt être réalisés directement dans l'*ebuild*, à l'aide de l'*eclass flag-o-matic*, cf. 1.3.2.

de cette directive est réalisé par des options *-Wl,-rpath,<chemin>* ajoutées aux *LDFLAGS* de compilation des paquetages utilisant la bibliothèque concernée, via les options *<ldflags>* du fichier *<spec>* - cf. l'exemple donné ci-dessous.

Une telle adaptation des *LDFLAGS* est en général réalisée pour l'ensemble des paquetages secondaires, compilés avec *CPREFIX=/usr/local*, afin d'inclure */usr/local/lib* dans leur chemin de recherche de bibliothèques dynamiques (cf. 2.1.2). Des adaptations spécifiques sont par ailleurs nécessaires pour certains paquetages, qui utilisent des bibliothèques stockées dans d'autres chemins alternatifs (par exemple */usr/local/lib/qt4* ou */usr/local/kde/3.5/lib*).

### 3.1.2 Exemple

A titre d'exemple, la définition d'un bloc *<pkg>* permettant la compilation de paquetages *app-misc/foo* et *dev-libs/bar* avec des options de debug (activées par le drapeau *USE debug*), en supposant qu'il s'agit de paquetages secondaires utilisant les bibliothèques *qt4* (installées dans */usr/local/lib/qt4* sous CLIP) et que le fait qu'il s'agisse de paquetages de debug est marqué par un suffixe *-debug* ajouté aux noms des paquetages, pourrait prendre la forme suivante :

```
<pkg>
    <pkgkey>foo-debug</pkgkey>
    <pkgnames>
        dev-libs/bar
        app-misc/foo
    </pkgnames>
    <use>debug</use>
    <ldflags>-Wl,-rpath,/usr/local/lib -Wl,-rpath,/usr/local/lib/qt4</ldflags>
    <env>CPREFIX=/usr/local, DEB_NAME_SUFFIX=-debug</env>
</pkg>
```

La compilation de ce bloc (par exemple par une commande *clip-compile <chemin spec> -pkgkey foo-debug*, cf. 3.3) entraînera la création de deux paquetages binaires, *bar-debug_<version>_i386.deb* et *foo-debug_<version'>_i386.deb*. Dans la pratique, la variable d'environnement *CPREFIX* et la directive *LDFLAGS -Wl,-rpath,/usr/local/lib* correspondante seraient probablement héritées du bloc *<conf>*, et n'auraient pas besoin d'être reprécisées dans le bloc *<pkg>* (ce qui laisserait dans ce cas une simple directive *<ldflags>-Wl,-rpath,/usr/local/lib/qt4</ldflags>* à spécifier dans ce bloc).

### 3.1.3 Prétraitement de fichiers *spec*

Les fichiers *spec* sont automatiquement préprocessés par l'utilitaire *clip-specpp* avant d'être lus par *clip-build*. Ce prétraitement comprend notamment un appel au préprocesseur C, *cpp*, ce qui permet d'inclure des macros et notamment des directives *#ifdef… #endif* dans les fichiers *spec*, et de différencier ceux-ci en fonction de variables de configuration. Les définitions passées au préprocesseur C sont celles listées dans la variable *CLIP_SPEC_DEFINES* de */etc/clip-build.conf* (cf. 3.3).

Ainsi, l'exemple suivant :

```
<pkg>
    <pkgkey>foo</pkgkey>
    <pkgnames>
        app-misc/foo
    </pkgnames>
    #ifdef CLIP_ANSSI
    <use>clip-anssi qt4</use>
    #else
    <use>qt4</use>
    #endif
</pkg>
```

entraînera la compilation de *app-misc/foo* avec ou sans le drapeau *USE clip-anssi*, selon que la variable *CLIP_ANSSI* apparaîtra ou non dans *CLIP_SPEC_DEFINES*. Ce mécanisme est particulièrement utile pour personnaliser la génération de paquetages à un déploiement particulier, sans créer un fichier *spec* distinct.

## 3.2 Principales variables affectant la génération de paquetages

Outre les variables d'environnement et drapeaux *USE* et *FEATURES* propre à l'outil *portage*, qui sont décrits dans les pages de manuel associées à ce dernier, la génération de paquetages CLIP fait intervenir un certains nombre de variables spécifiques, qui sont listées ci-dessous.

### 3.2.1 Drapeaux *USE* spécifiques

Les drapeaux *USE* spécifiques à CLIP sont les suivants :

- **clip** : drapeau générique pour l'activation de fonctionnalités spécifiques à CLIP. Ce drapeau doit être activé lors de la génération de tout paquetage binaire CLIP, mais est automatiquement masqué par *clip-compile* lors de la compilation des dépendances de compilation (cf. 3.3).
- **clip-devstation** : activation de fonctionnalités spécifiques à l'environnement de compilation CLIP, activé automatiquement par *clip-compile* pour l'installation des dépendances de compilation. Ce drapeau ne doit en revanche jamais être positionné pour la génération de paquetages binaires CLIP.
- **clip-rm** : activation de fonctionnalités spécifiques à l'environnement RM, activé par le fichier *spec* correspondant à la distribution *rm*.
- **clip-core** : activation de fonctionnalités spécifiques à l'environnement CLIP (socle, par opposition à RM), activé par le fichier *spec* correspondant aux distributions *clip*.
- **rm-core** : activation de fonctionnalités spécifiques aux paquetages essentiels RM.
- **core-rm** : activation de fonctionnalités spécifiques à l'environnement CLIP, lorsque celui-ci doit supporter des cages RM (activé pour la distribution *clip* d'une configuration CLIP-RM, mais pas pour celle d'une configuration CLIP-GTW).
- **clip-gtw** : activation de fonctionnalités spécifiques à des configurations CLIP-GTW.
- **clip-ccsd** : utilisation des primitives cryptographiques CCSD.
- **clip-tcb** : utilisation du mécanisme TCB pour le stockage des empreintes de mots de passe (cf. [CLIP 1302]). En pratique, cette option est aujourd'hui indispensable au fonctionnement correct du système CLIP.
- **clip-x11** : installation de l'affichage X11 (options spécifiques au système CLIP). En pratique, cette option est aujourd'hui indispensable au fonctionnement correct du système CLIP.
- **clip-kde3** : installation de composants KDE3 avec des options nécessaires à leur cohabitation avec l'environnement principal KDE4.
- **clip-devel** : activation de fonctionnalités spécifiques à une configuration de test pour le développement CLIP (correspondant essentiellement à la désactivation de certains mécanismes de sécurité).
- **clip-strongswan** : utilisation de *strongswan* comme démon IKEv2.
- **clip-racoon2** : utilisation de *racoon2* comme démon IKEv2, alternativement à *strongswan* (cette option est désormais obsolète).
- **clip-deps**, **core-deps** et **rm-deps** : drapeaux permettant la suppression de dépendances de paquetages secondaires vis-à-vis de paquetages essentiels, cf. 2.2.1.
- **clip-hermes** : activation de fonctionnalités spécifiques au déploiement HESTIA.
- **clip-anssi** : activation de fonctionnalités spécifiques au déploiement ANSSI (réseau de test CLIP).

### 3.2.2 *FEATURES* spécifiques

Les options *portage* (drapeaux positionnables dans la variable d'environnement *FEATURES*) spécifiques à CLIP sont les suivantes :

- **noinclude** : suppression automatique des répertoires */usr/include* (*/usr/local/include* pour un paquetage secondaire) avant l'installation ou la création du paquetage binaire.
- **nostatlib** : suppression automatiques des bibliothèques statiques (fichiers *.a*), ainsi que des fichiers *libtool .la* sauf si la variable d'environnement *NOSTATLIB_KEEPLA* est définie, avant l'installation ou la création du paquetage binaire.

### 3.2.3 Variables d'environnement spécifiques

Outre les mécanismes propres à *portage* que sont les variables *USE* et *FEATURES*, la génération de paquetages CLIP peut également être paramétrée à l'aide d'un certain nombre de variables d'environnement. Celles-ci se répartissent en plusieurs catégories.

**Options de configuration des chemins d'installation**

- **CPREFIX** : définition d'un préfixe de configuration autre que */usr* (cf. 2.1.2).
- **CLIP_VROOTS** : définition de préfixes d'installation multiples (cf. 2.1.3).
- **NOSTATLIB_KEEPLA** : lorsque cette variable est définie à une valeur non vide, la *FEATURE nostatlib* n'entraîne pas la suppression des fichiers *libtool .la*, dont la présence peut être nécessaire à certaines applications.

**Options de génération de paquetages *debian***

Les variables suivantes permettent la définition de certains champs de contrôle des paquetages *debian* générés.

- **DEB_DISTRIBUTION** : définit le champ *Distribution* des paquetages (valeur obligatoire : *clip* ou *rm*).
- **DEB_PRIORITY** : définit le champ *Priority* des paquetages (valeur obligatoire : *Required* ou *Important*).
- **DEB_ESSENTIAL** : si non vide, le champ *Essential* des paquetages est défini à *yes*. La définition de ce champ est à réserver à certains des paquetages essentiels uniquement : elle dénote des paquetages pour lesquels une désinstallation, même temporaire afin de réduire un conflit, n'est en aucun cas autorisée. Elle est typiquement associée au paquetage *dpkg* et à ses dépendances.
- **DEB_URGENCY** : définit le champ *Urgency* (valeur numérique ou *NA*) des paquetages (champ spécifique à CLIP).
- **DEB_IMPACT** : définit le champ *Impact* (valeur numérique) des paquetages (champ spécifique à CLIP).
- **DEB_NAME_SUFFIX** : permet de définir un suffixe à concaténer aux noms des paquetages (cf. 2.2.3).
- **DEB_JAILS** : définit le champ (spécifique à CLIP) *CLIP-Jails* des paquetages optionnels, permettant de limiter les cages RM dans lesquelles le paquetage est proposé à l'installation (cf. 3.4.2).

**Options de paramétrage des entrées *veriexec***

Les options suivantes affectent la génération des entrées *veriexec* associées aux paquetages, et sont décrites plus en détail en 2.4 :

- **VERIEXEC_DO_SHLIBS** : création automatique d'empreintes pour les bibliothèques dynamiques si définie à une valeur non nulle.
- **VERIEXEC_CTX** : contexte associé aux entrées créées explicitement par *doverictl* - on notera que certains *ebuild* redéfinissent eux-mêmes cette variable, et ignorent par conséquent celle éventuellement passée par l'environnement (par défaut : contexte -1, c'est-à-dire utilisation du contexte du processus qui charge les entrées).
- **VERIEXEC_LIB_CTX** : contexte associé aux entrées créées automatiquement pour les bibliothèques, du fait du positionnement de *VERIEXEC_DO_SHLIBS* (par défaut : contexte -1).

## 3.3 Utilisation de *clip-compile*

### 3.3.1 Environnement de développement

La génération de paquetages CLIP est réalisée dans un environnement *chroot*, au sein duquel les paquetages sont installés directement par des commandes *portage*, et non en passant par des paquetages binaires *debian* comme c'est le cas sur les postes CLIP eux-mêmes. L'environnement de développement est ainsi très proche d'un environnement *Gentoo* classique, la seule différence étant liée à certaines options des *ebuilds* contrôlées par le drapeau *USE clip-devstation*.

L'environnement de développement se présente sous la forme d'une arborescence autosuffisante, qui peut être installée dans un chemin quelconque sur un poste Linux, sans contrainte particulière sur la distribution utilisée par le poste hôte [11]. L'utilisation de cet environnement de développement se fait par une simple commande *chroot*, utilisée pour lancer un *shell root* dans l'arborescence de développement, après avoir réalisé un montage *bind* de certains répertoires du système hôte sur les répertoires correspondants de l'arborescence de développement : */proc*, */dev* et */dev/pts*, ainsi éventuellement qu'un système de fichiers *tmpfs* sur */var/tmp/portage*.

Afin de supporter la génération de paquetages CLIP, l'environnement de développement doit intégrer un certain nombre de composants logiciels :

- les outils de compilation classiques : *gcc*, *binutils*, etc., mais aussi *javac*, dans les versions adaptées et avec les éventuelles adaptations spécifiques à CLIP ;
- les outils de création de paquetages : *portage*, *dpkg* et *clip-build* principalement ;
- un certains nombre d'utilitaires facilitant la manipulation des paquetages binaires CLIP, essentiellement fournis par le paquetage *clip-devutils* ;
- l'ensemble des dépendances de compilation des différents paquetages CLIP à générer : paquetages fournissant en particulier les fichiers d'en-tête, bibliothèques statiques et dynamiques, et éventuels exécutables spécifiques nécessaires à la compilation des paquetages CLIP.

L'ensemble de ces composants logiciels peut être installé et mis à jour à l'aide de commandes *emerge* classiques, à partir des différents répertoires *portage* du *repository clip-int* (y compris le répertoire *portage-overlay-dev*, spécifique à cet environnement de développement). Il est rappelé (cf. 1.1.1) que les sources de paquetages (*distfiles*) utilisées pour l'installation au sein de l'environnement de développement sont cherchées dans le répertoire *distfiles-dev/*, et non *distfiles/*, de *clip-int*.

Cependant, de préférence à l'utilisation directe de commandes *emerge*, il est recommmandé d'installer des composants logiciels au sein de l'environnement de développement à l'aide de commandes *clip-compile --depends*, comme décrites dans la section suivante, afin d'assurer une bonne prise en compte des dépendances de compilation et options spécifiques adaptées aux paquetages CLIP. Dans la pratique, l'environnement de développement est initialement fourni sous une forme minimaliste, n'intégrant pas l'essentiel des dépendances de compilation de paquetages CLIP, ce qui rend nécessaire l'utilisation de commandes *clip-compile --depends* avant toute génération de paquetages binaires.

### 3.3.2 Commandes de génération de paquetage

La commande *clip-build* supporte un nombre important d'options, qui peuvent être listées par *clip-build -h*. Ces options permettent une configuration très fine des opérations réalisées par l'outil, mais rendent son utilisation relativement lourde (lignes de commande complexes). La commande *clip-compile* permet de réaliser de manière plus simple les opérations les plus classiques de génération de paquetages CLIP, en appelant elle-même la commande *clip-build* avec des options prédéfinies. Cette commande *clip-compile* ne supporte quant à elle qu'un nombre réduit d'options. Elle s'invoque sous les formes suivantes :

11. Il est en revanche évidemment nécessaire que le poste hôte soit compatible avec l'environnement de développement pour ce qui est de l'architecture CPU - qui doit donc être une architecture compatible *Intel 32bits* (y compris processeurs *Intel* ou *AMD 64 bits*, dès lors que le noyau du système hôte supporte le mode compatibilité 32 bits)

```
clip-compile <chemin spec> [<options>]
clip-compile <chemin spec> [<options>] -conf {config}
clip-compile <chemin spec> [<options>] -pkgkey {key}
clip-compile <chemin spec> [<options>] -pkgname {name}
```

Dans les exemples de commandes présentés ci-dessus, le paramètre *<chemin spec>* représente le chemin du fichier *<spec>* à utiliser (cf. 3.1), relativement au répertoire *specs/* de la branche courante du *repository clip-int*, et sans l'extension *.spec.xml*. Le chemin de cette branche est lui-même issu du fichier de configuration *clip-build.conf*, comme détaillé ci-dessous. Les valeurs typiquement données à ce paramètre *<chemin spec>* sont ainsi *clip-rm/clip* (CLIP-RM, distribution *clip*), *clip-rm/rm* (CLIP-RM, distribution *rm*) ou *clip-gtw/clip* (CLIP-GTW, distribution *clip*).

Les quatres lignes de commande données en exemple ci-dessus diffèrent par leur portée. La première réalise la compilation de l'ensemble des paquetages référencés dans le fichier *spec*, ou de leur dépendances de compilation si l'option *--depends* a été passée à la commande. Les deux suivantes réalisent une compilation limitée, respectivement, aux paquetages du bloc *<conf>...</conf>* identifié par le nom *<confname>{config}</confname>* pour la première, et à ceux du block *<pkg>...</pkg>* identifié par la clé *<pkgkey>{key}</pkgkey>* pour la seconde. Enfin, la dernière ligne permet de limiter les opérations à un unique paquetage référencé par le fichier *spec*, en identifiant ce paquetage par le nom sous lequel il est référencé (typiquement de la forme *<categorie>/<nom>*, par exemple *app-misc/foo*).

Les options *<options>* supportées par la commande sont les suivantes :

**--pretend** : affiche les opérations qui seraient effectuées (paquetages générés pour CLIP ou installés dans l'environnement de développement, avec les options associées), mais n'effectue pas ces opérations.

**--depends** : réalise la compilation et l'installation locale au sein de l'environnement de développement des dépendances de compilation (*DEPEND* des *ebuilds*) des paquetages spécifiés, plutôt que la génération des paquetages binaires correspondants.

**--buildpkg** : cette option, qui n'a de sens que lorsqu'elle est combinée avec *--depends*, active la création de paquetages binaires *Gentoo*[12] pour toutes les dépendances de compilation installées au sein de l'environnement - ce qui permet ensuite leur réutilisation par d'autres postes de développement sans recompilation.

Le lecteur pourra aussi se référer à la page de manuel *man clip-build(7)*.

### 3.3.3 Fichier de configuration

La commande *clip-compile*, ainsi qu'un certain nombre d'utilitaires d'aide au développement fournis par le paquetage *clip-dev/clip-devutils*, font appel à des paramètres de configuration stockés dans le fichier */etc/clip-build.conf* au sein de l'environnement de développement. Ces paramètres permettent en particulier de personnaliser les chemins utilisés au sein de l'environnement de développement, et d'adapter ce dernier à un développeur particulier. Ce fichier prend la forme classique d'un fichier de configuration destiné à être chargé par des scripts *shell* via la commande *source*. Il contient ainsi un ensemble de définitions de variables sous la forme *VARIABLE="valeur"*. Les variables susceptibles d'être définies dans ce fichier sont énumérées ci-dessous.

**Chemins**

Les variables suivantes permettent de préciser certains chemins au sein de l'environnement de développement :

- **CLIP_BASE** : chemin de la copie locale de la branche courante du *repository clip-int*. Par exemple, si *CLIP_BASE* est défini à */toto*, les répertoires */toto/portage-overlay-clip* et */toto/distfiles* doivent corres-

12. Paquetages au format *.tbz2*, utilisables par *portage*, à ne pas confondre avec les paquetages au format *.deb* déployés sur les postes CLIP.

pondre respectivement à l'*overlay* CLIP et aux répertoire de *distfiles* utilisés pour la génération des paquetages.

- **DEBS_BASE** : répertoire de base pour le stockage des paquetages binaires CLIP générés. Les paquetages seront en pratique stockés dans des sous-répertoires de ce chemin.
- **CLIP_SPEC_MAP** : correspondance entre le nom de chaque fichier *spec* (défini par le champ <*specname*> du fichier *spec*) et sous-répertoire de *$DEBS_BASE* où seront stockés les paquetages générés pour ce fichier, exprimées par des lignes <*nom spec*> => <*nom sous-répertoire*>, par exemple :

```
CLIP_SPEC_MAP="
        CLIP    =>      clip
        RM      =>      rm
"
```

qui stockera les paquetages issus des fichiers *spec* nommés *CLIP* et *RM* dans les répertoires *${DEBS_BASE}/clip* et *${DEBS_BASE}/rm*, respectivement.

- **PKG_DIR** : chemin de base de stockage des paquetages binaires *Gentoo* produits par *clip-compile --depends --buildpkg*.

**Options de génération**

Les variables suivantes permettent d'ajuster certains paramètres de génération des paquetages :

- **CLIP_SPEC_DEFINES** : définition de variables à passer au prétraitement des fichiers *spec* (cf. 3.1.3), afin de personnaliser ces derniers pour un déploiement particulier. Par exemple, la définition *CLIP_SPEC_DEFINES="VAR1* entraînera le passage des options *-DVAR1 -DVAR2=val -DVAR3* au préprocesseur C lors du prétraitement des fichiers *spec*.
- **CLIP_MAKEOPTS** : options à passer aux commandes *make* lancées par *clip-compile* via *emerge* - permettant typiquement d'ajuster le nombre de fils d'exécution réalisés en parallèle. Exemple : *CLIP_MAKEOPTS="-j8"*.
- **CLIP_CHOST** : définition de la variable *CHOST* pour choisir le compilateur à utiliser pour la génération des paquetages, par exemple *CLIP_CHOST="i686-pc-linux-gnu"*. Cette variable ne doit en général pas être modifiée.

**Paramètres de signature**

Les outils de signature (cf. 3.5.2) des paquetages utilisent implicitement les variables suivantes :

- **DEV_SIGN_KEY** : chemin du trousseau de clés privées ACID à utiliser pour la création des signatures développeur.
- **DEV_SIGN_PWD** : chemin du fichier contenant le mot de passe de la clé privée ACID à utiliser pour la création des signatures développeur. Si ce chemin n'est pas défini, la saisie du mot de passe sera demandée pour chaque opération de signature.
- **DEV_SIGN_CERT** : chemin du trousseau de clés publiques ACID à utiliser pour la création des signatures développeur.
- **CTRL_SIGN_KEY** : chemin du trousseau de clés privées ACID à utiliser pour la création des signatures validateur.
- **CTRL_SIGN_PWD** : chemin du fichier contenant le mot de passe de la clé privée ACID à utiliser pour la création des signatures validateur. Si ce chemin n'est pas défini, la saisie du mot de passe sera demandée pour chaque opération de signature.
- **CTRL_SIGN_CERT** : chemin du trousseau de clés publiques ACID à utiliser pour la création des signatures validateur.

**Autres options**

- **CLIP_BUILDER** : texte en forme libre (typiquement, nom et adresse e-mail de l'utilisateur de l'environnement de développement) à inclure dans le champ *Built-By* de chaque paquetage binaire CLIP, et

à positionner dans les squelettes d'entrées dans les *ChangeLog* lors de l'incrémentation du numéro de version d'un *ebuild* par la commande *clip-bump* (cf. *man clip-devutils*).
- **USE_SVN** : lorsque cette variable est positionnée à *yes*, certains utilitaires du paquetage *clip-devutils* (par exemple *clip-bump*) répercutent automatiquement leurs opérations dans la gestion de version *subversion*.

## 3.4 Création d'une configuration

### 3.4.1 Principe des configurations

Le déploiement de paquetages binaires sur des postes CLIP nécessite leur référencement par un paquetage spécifique, dit *configuration*. Une configuration est un paquetage *debian* qui n'installe aucun fichier (hors journaux de modifications éventuels), mais qui référence l'ensemble des autres paquetages à installer dans ses dépendances, avec des contraintes *exactes* sur les numéros de version de ces paquetages (c'est-à-dire des contraintes =, et non >= par exemple). Les paquetages de configuration doivent porter un nom de la forme **-conf*, par exemple *clip-apps-conf*, et aucun paquetage autre qu'une configuration ne doit porter un nom de cette forme. A l'instar des autres paquetages CLIP, les configurations sont générées à partir d'*ebuilds*, dans ce cas rattachés à la catégorie *clip-conf* (dans *portage-overlay-clip*).

Les configurations suivent la répartition par distribution et priorité (essentiel ou secondaire) des paquetages : il y a une et une seule configuration par distribution et niveau de priorité : *clip-core-conf* et *clip-apps-conf* pour la distribution CLIP (paquetages essentiels et secondaires, respectivement), *rm-core-conf* et *rm-apps-conf* pour la distribution RM. Chaque configuration référence dans ses dépendances (champ *RDEPEND* de l'*ebuild*, champ *Depends* du paquetage *debian*) l'ensemble des paquetages à installer pour la distribution et le niveau de priorité considérés, à l'exception de la configuration elle-même et des éventuels paquetages optionnels (qui font l'objet de la sous-section suivante). Ainsi, la mise à jour de paquetages d'une distribution CLIP nécessite systématiquement la mise à jour correspondante des versions de ces paquetages dans la configuration qui les référence, et la génération du paquetage binaire associé à cette configuration.

Les paquetages de configuration n'installent en général pas de fichiers particuliers, à l'exception d'un fichier **-release* contenant le numéro de version de la configuration, et d'un fichier **-release.html* contenant un journal des modifications de la configuration, au format HTML. Les chemins de fichiers associés aux différentes configurations sont les suivants :
- *clip-core-conf* : */etc/shared/clip-release{,.html}*
- *clip-apps-conf* : */usr/local/etc/clip-apps-release{,.html}*
- *rm-core-conf* : */etc/shared/rm-release{,.html}*
- *rm-apps-conf* : */usr/local/etc/rm-apps-release{,.html}*

Par ailleurs, une configuration peut naturellement incorporer des *maintainer scripts*, selon les mécanismes décrits en 2.3.

On notera également que les suffixes ajoutés aux noms de certains paquetages binaires via la variable d'environnement *DEB_NAME_SUFFIX* (cf. 2.2.3) doivent être explicitement rajoutés aux noms correspondants dans les dépendances de la configuration associée.

### 3.4.2 Gestion des paquetages optionnels

Les paquetages optionnels ne sont pas référencés dans le *RDEPEND* de la configuration correspondante (qui définit les paquetages systématiquement installés). En revanche, la liste des paquetages optionnels autorisés par une configuration doit apparaître dans le champ *Suggests* du paquetage *debian* associé à cette configuration. Afin de permettre la définition de ce champ, les modifications de *portage* spécifiques à CLIP ajoutent le support d'un champ **DEB_SUGGESTS** dans les *ebuilds*, exprimé sous la même forme que les champs standards *DEPEND* et *RDEPEND*. Le contenu de ce champ est transformé en champ *Suggests* lors de la création de

paquetages *debian*, tout comme le contenu de *RDEPEND* est transformé en *Depends*. Ainsi, l'ensemble des paquetages optionnels autorisés par une configuration doit être listé dans le champ *DEB_SUGGESTS* de l'*ebuild* associé à celle-ci.

Par ailleurs, pour être affiché dans le menu de sélection de paquetages optionnels au sein du système CLIP, un paquetage optionnel doit incorporer un champ *debian Description-fr*, contenant la description en français du paquetage. Ce champ *debian* est automatiquement produit à partir de la valeur de la variable *DESCRIPTION_FR*, lorsqu'une telle variable est définie dans l'*ebuild* correspondant. On notera que tous les paquetages optionnels ne comportent pas nécessairement une telle description : bon nombre de ces paquetages ne présentent un intérêt que comme dépendance nécessaire à un autre paquetage optionnel, et n'ont pas de raison d'être sélectionnés en tant que tels par l'administrateur d'un système CLIP. Seuls les paquetages amenés à pouvoir être explicitement sélectionnés doivent inclure un champ *DESCRIPTION_FR*. Ainsi, le paquetage *media-gfx/gimp* incorporera une telle description, mais pas le paquetage *media-libs/babl*, qui ne présente un intérêt que comme dépendance de *gimp*.

Enfin, il peut être souhaitable, pour certains paquetages optionnels RM, de limiter les cages RM au sein desquelles ils pourront être installées. Ce serait par exemple le cas d'un paquetage optionnel donnant accès à certaines fonctionnalités propres au niveau RM_H et sans équivalent dans RM_B, au sein d'un déploiement CLIP-RM biniveau. La limitation de la possibilité de sélectionner un paquetage à certaines cages RM uniquement est réalisée à l'aide d'un champ spécifique, *CLIP-Jails*, potentiellement présent dans les paquetages *debian*. Lorsqu'il est présent dans un paquetage et contient une liste de cages RM (noms en minuscules, séparés au besoin par des virgules), le paquetage ne pourra être sélectionné que dans les cages listées. En l'absence de ce champ, le paquetage peut être sélectionné pour n'importe quelle cage RM. La définition de ce champ est possible via la variable *DEB_JAILS*, qui doit être passée par l'environnement via le fichier *spec* (cf. 3.2.3), plutôt que définie dans l'*ebuild* comme c'est le cas pour *DESCRIPTION_FR*[13].

### 3.4.3 Dépendances entre configurations

Comme évoqué en section 2.2.1, il n'est pas possible pour un paquetage secondaire CLIP de dépendre directement d'un paquetage essentiel. En revanche, les outils de mise à jour supportent un type de dépendance particulier, permettant d'exprimmer une dépendance entre une configuration secondaire et une version particulière de la configuration essentielle associée à la même distribution. Ce mécanisme repose sur l'utilisation d'un champ *debian* spécifique, *ConfDepends*, généré à partir d'un champ *CONF_DEPENDS* inclus dans l'*ebuild* associé à la configuration secondaire. Ce champ peut être défini selon les mêmes règles syntaxique que le champ standard *RDEPEND*, et fait l'objet d'une transformation de format similaire à ce dernier lors de la production d'un paquetage *debian*.

Le champ *ConfDepends*, contrairement au champ *Depends*, n'est pris en compte que par la procédure d'installation des mises à jour CLIP, et pas par la procédure de téléchargement de ces mises à jour. Ainsi, une configuration secondaire pourra sans problème être téléchargée, ainsi que tous les paquetages qu'elle référence, même si la configuration essentielle dont elle dépend via *ConfDepends* n'est pas installée, ni même présente dans le miroir local. En revanche, une fois la configuration secondaire intégrée dans le miroir local, l'installation de cette configuration ne sera pas tentée tant que la configuration essentielle n'aura pas été installée dans la bonne version. Cette distinction évite de bloquer inutilement le téléchargement d'une configuration, mais permet d'éviter toute incohérence entre les configurations installées.

On notera bien que l'expression de dépendances entre configurations via *ConfDepends* n'est possible qu'au sein de la même distribution - une configuration ne peut en aucun cas dépendre d'une configuration relevant d'une autre distribution, par exemple *rm* vis-à-vis de *clip*. De plus, même s'il est théoriquement possible de faire dépendre une configuration essentielle d'une version particulière d'une configuration secondaire, une telle dépendance aurait de fortes chances de créer une situation d'autoblocage du fait de dépendances croisées.

13. La différence d'approche s'explique par le fait que l'attribution d'un paquetage à une ou des cages particulières peut être spécifique à un déploiement, contrairement à la description de ce paquetage qui reste générique.

Il est donc très fortement préférable d'adapter les configurations essentielles afin qu'elles supportent n'importe quelle version de configuration secondaire, ou à défaut de telle sorte qu'une incohérence de version n'entraîne le dysfonctionnement que des fonctionnalités secondaires, et dans tous les cas pas de celles de mise à jour.

## 3.5 Création d'un miroir

Une fois qu'un ensemble cohérent de paquetages binaires, et la configuration associée, ont été générés, leur déploiement sur des postes CLIP (par mise à jour ou installation) nécessite la création d'un miroir de paquetages signés, selon les étapes listées ci-dessous.

### 3.5.1 Tests de cohérence

Avant de procéder au déploiement d'une configuration, une première étape importante consiste à vérifier sa cohérence. A cette fin, le paquetage *clip-devutils*, installé au sein de l'environnement de développement, fournit un utilitaire *clip-checkconfig* permettant d'effectuer automatiquement un ensemble de tests de validation. Cet utilitaire doit être lancé dans le répertoire contenant l'ensemble des paquetages binaires de la configuration *<config>.deb* à tester, et peut être invoqué selon deux modes :

- *clip-checkconfig <config>.deb*, d'exécution rapide, permet de simplement vérifier que l'ensemble des paquetages référencés par la configuration (via ses champs *Depends* et *Suggests*) est bien présent dans le répertoire courant.
- *clip-checkconfig -f <config>.deb*, plus lent, effectue la vérification précédente, mais aussi une vérification récursive des dépendances des paquetages référencés par cette configuration, afin de s'assurer de la cohérence d'ensemble, c'est-à-dire de l'absence de toute dépendance qui ne serait pas satisfaite par cet ensemble de paquetages, et de tout conflit entre paquetages, et de vérifier la cohérence globale des champs distribution et priorité. Une vérification des chemins des fichiers installés par les paquetages, afin de s'assurer qu'ils sont acceptables pour le niveau de priorité (essentiel ou secondaire) considéré, est également réalisé par l'utilitaire.

Il est fortement conseillé de systématiquement effectuer une vérification complète (*clip-checkconfig -f*) avant le déploiement d'une configuration.

### 3.5.2 Signature des paquetages

Afin de pouvoir être acceptés par le système de mise à jour CLIP, les paquetages binaires doivent être doublement signés [14]. La répartition organisationnelle des prérogatives de signature n'est pas du ressort du présent document, qui se limitera donc à une présentation des utilitaires associés à la gestion des signatures. Le paquetage *clip-devutils* fournit à ce titre deux utilitaires :

- **clip-checksign** : permet la vérification de présence (et non de validité) de signatures dans les paquetages.
- **clip-sign** : permet l'ajout de signatures dans les paquetages.

Ces deux utilitaires attendent de la même manière une option obligatoire afin de préciser le type de signature à prendre en compte :

- *-d* : signature développeur uniquement.
- *-c* : signature validateur uniquement.
- *-a* : signatures développeur et validateur.

Par ailleurs, *clip-checksign* n'accepte pas d'autre argument : il vérifie simplement l'ensemble des fichiers *.deb* présents dans le répertoire courant, et liste sur sa sortie standard ceux qui n'intègrent pas la ou les signatures spécifiées. En revanche, *clip-sign* attend également sur sa ligne de commande la liste des paquetages

14. Par ailleurs, même si l'installation d'un poste CLIP est possible avec des paquetages non signés, un poste ainsi installé ne disposera initialement pas des fonctionnalités de mise à jour complètes, en particulier pas de la possibilité de revenir à une configuration cohérente en cas d'échec de mise à jour.

auxquels ajouter la ou les signatures spécifiées. Cet outil produit la ou les signatures à l'aide des clés définies par le fichier de configuration */etc/clip-build.conf*, cf. 3.3.3. Ainsi, en supposant l'ensemble des clés nécessaires disponibles et correctement configurées, la double signature de l'ensemble des paquetages non signés peut-être réalisée par la commande :

```
clip-checksign -a | xargs clip-sign -a
```

### 3.5.3 Structure d'un miroir, création et mise à jour

Un miroir de paquetages CLIP est en pratique constitué de plusieurs miroirs de format *debian*, à raison d'un miroir par configuration, répartis selon la hiérarchie suivante :

- *clip/clip-core-conf* : configuration essentielle CLIP
- *clip/clip-apps-conf* : configuration secondaire CLIP
- *rm/rm-core-conf* : configuration essentielle RM (le cas échéant)
- *rm/rm-apps-conf* : configuration secondaire RM (le cas échéant)

Chacun de ces miroirs est à son tour organisé selon un schéma classique de miroir *debian*, avec un sous-répertoire *pool* contenant l'ensemble des paquetages, et un fichier *dists/<dist>/main/binary-i386/Packages.gz*, avec *<dist>* la distribution *clip* ou *rm* concernée, listant les métadonnées du miroir. Ce dernier doit être produit par *apt-ftparchive* [15] et compressé par *gzip*, par exemple en lançant la commande suivante à la racine du miroir :

```
apt-ftparchive packages pool | gzip - > dists/*/main/binary-i386/Packages.gz
```

La création initiale d'un miroir peut être réalisée avec la commande */opt/clip-livecd/get-mirrors.sh* au sein de l'environnement de développement (commande fournie par le paquetage *clip-livecd*, les arguments à passer sont explicités lorsque la commande est lancée sans argument).

La mise à jour d'un miroir existant à partir de nouveaux paquetages (et de la configuration qui les référence) peut être réalisée en copiant les nouveaux paquetages dans le sous-répertoire *pool/* du miroir, puis en invoquant, depuis ce même répertoire, les commandes suivantes, issues du paquetage *clip-devutils* :

- *clip-prunepkgs* : permet de ne conserver que la dernière version de chaque paquetage, en supprimant les doublons anciens.
- *clip-cleanconfigs -a <configuration>.deb* : permet de supprimer tous les paquetages non référencés par la configuration *<configuration>.deb* (en demandant confirmation).
- *clip-checkconfig <configuration>.deb* : permet de vérifier la présence de tous les paquetages référencés par la configuration *<configuration>.deb*.

A l'issue de ces manipulations, l'index du miroir peut être mis à jour par la commande *apt-ftparchive* évoquée plus haut, puis le miroir peut être déployé sur les serveurs de mise à jour et supports d'installation.

15. La création de l'index ne doit pas être réalisée avec *dpkg-scanpackages*, qui ne conserve pas les métadonnées nécessaires aux mises à jour CLIP.

# A Créer des paquetages CLIP pour les nuls

## A.1 Pré-requis

Il est nécessaire d'avoir un environnement de développement installé (cf. [CLIP 1103]) et configuré (cf. 3.3.3).

## A.2 Créer un ticket dans Bugzilla

### A.2.1 Créer le ticket

Créer un ticket dans le bugzilla de **développement**, sur *clip-dev* : `https://clip.ssi.gouv.fr/`.

Un ticket permet de suivre les changements liés à une ou plusieurs modifications de CLIP. Il peut être judicieux de créer un ticket ayant uniquement pour but de contenir la description initiale macroscopique et chapeautant (champs *Depends* et *Blocks*) différents autres tickets qui décrivent en détail les changements à apporter et les commits Subversion liés.

Le titre et la description initiale doivent présenter clairement le but du ticket et éventuellement les différentes étapes prévues pour résoudre le problème ou ajouter la fonctionalité. Tous commentaires pouvant être utile à la compréhension des modifications (tests, discussions, choix) doivent être précisées. Il peut également être utile de joindre certains fichiers tel que des logs ou éventuellement des patchs de test.

Il faut ensuite renseigner tous les champs disponibles (Composant, Gravité, Priorité...).

### A.2.2 Modifier son statut

Lorqu'on est prêt à prendre le ticket, se l'assigner et changer son statut en *ASSIGNED*.

Mémoriser le numéro de ticket pour pouvoir l'utiliser dans les messages de commits.

## A.3 Récupérer le paquetage Gentoo et ses dépendances

### A.3.1 Récupérer l'*ebuild*

Il faut d'abord récupérer les fichiers qui indiquent comment les sources doivent être compilées :

1. Choisir un miroir FTP Gentoo (e.g. `ftp://ftp.free.fr/mirrors/ftp.gentoo.org`).
2. Télécharger la clé publique PGP de « Gentoo Portage Snapshot Signing Key » :

```
$ gpg --recv-keys C9189250
$ gpg --list-keys C9189250
pub   4096R/96D8BF6D 2011-11-25 [expire :2015-11-24]
uid                  Gentoo Portage Snapshot Signing Key (Automated Signing Key)
sub   4096R/C9189250 2011-11-25 [expire :2015-11-24]
```

3. Télécharger la dernière version du snapshot de Portage :

```
$ lftp ftp.free.fr/mirrors/ftp.gentoo.org/snapshots
> get portage-latest.tar.xz
> get portage-latest.tar.xz.gpgsig
```

4. Vérifier qu'elle n'a pas été altéré :

```
$ gpg --verify portage-latest.tar.xz.gpgsig portage-latest.tar.xz
```

### A.3.2 Identifier les fichiers nécessaires

Identifier et importer les fichiers nécessaires dans *clip-int* (cf. 1.1.1) :
- version des *ebuild* voulues (dernière version stable : variable *KEYWORDS* contenant *x86*) ;
- fichiers *MISC* (`ChangeLog`, `Manifest`, `metadata.xml`) ;
- fichiers supplémentaires nécessaires (fichiers *AUX* situés dans *FILESDIR*) ;
- dépendances (DEPEND et RDEPEND) nécessaires (drapeaux *USE* utilisés, notamment pour *clip*, *clip-livecd* et *clip-devstation*).

### A.3.3 Récupérer les sources

Lister et récupérer les fichiers correspondant au SRC_URI de l'ebuild (ne pas tout lister !) :

```
$ lftp ftp.free.fr/mirrors/ftp.gentoo.org/distfiles
> ls foo*
> get foo-1.2.3.tar.gz
```

### A.3.4 Récupérer les dépendances

Les dépendances sont indiquées dans le fichier *ebuild* par deux directives :
- `DEPEND` indique les paquetages nécessaires à la compilation ;
- `RDEPEND` indique les paquetages nécessaires à l'exécution.

La vérification de ces paquetages se fait en parcourant l'arborescence de `clip-int`, maintenue par Subversion.

## A.4 Créer l'arborescence de développement du paquetage dans *clip-int*

### A.4.1 Mettre les sources dans le bon `distfiles`


Si le paquetage ne doit être utilisé que par *clip-devstation*, utiliser uniquement le dossier `clip-int/distfiles-dev`.

Placer l'archive source dans `/opt/clip-int/distfiles` et `/opt/clip-int/distfiles-dev` :

```
$ clip-cpdistfile [-d] ~/paquetages/xournal-0.4.5.tar.gz
```

### A.4.2 Créer l'arborescence *ebuild*

Créer l'arborescence *ebuild* dans le répertoire qui convient :

```
# mv ~/app-text/xournal/ portage-overlay/app-text/
```

Règles pour déterminer le bon répertoire :
- `portage/` pour les paquetages portés tels quels, sans modification ;
- `portage-overlay/` pour les paquetages requiérant des modifications ;
- `portage-overlay-clip/` pour les applications spécifiques à CLIP ;
- `portage-overlay-dev/` pour les applications Gentoo destinées aux postes de développement.

### A.4.3 Vérifier les distfiles

La vérification des distfiles téléchargés se fait ainsi :
- copier le Manifest (non modifié) de portage ;
- vérifier la signature PGP du Manifest ;
- faire un `ebuild foo-1.2.3.ebuild manifest` pour s'assurer que les distfiles correspondent bien à leurs checksums.

### A.4.4 Modifier le champ de description de l'*ebuild*

Concernant les paquetages destinés à apparaitre en paquetages optionel CLIP, le fichier *ebuild* doit être modifié afin que les variables `DESCRIPTION_FR` et `CATEGORY_FR` soit positionnée :

```
DESCRIPTION="bind tools :dig, nslookup, host, nsupdate, dnssec-keygen"
DESCRIPTION_FR="Outils de test DNS :dig, nslookup, host, nsupdate"
CATEGORY_FR="éRseau"
```

### A.4.5 Supprimer les dépendances à certaines bibliothèques

Tout d'abord, on distingue deux distributions (3.4.1) : la distribution *clip* est destinée aux cages essentielles (socle, update, etc.) et la distribution *rm* est destinée aux cages utilisateur.
Ensuite, on distingue deux types de paquetages : les paquetages *primaires* essentiels au fonctionnement du poste (comme *dev-libs/openssl* ou *app-arch/dpkg*) et les paquetages *secondaires*, purement optionnels.
Un dernier point : les paquetages sont regroupés au sein de groupes voire de sous-groupes.

**Savoir si un paquetage est primaire ou secondaire**

Les paquetages primaires sont marqués `DEB_PRIORITY=Required` dans le fichier `/opt/clip-int/specs/clip-rm/rm.spec.xml` ou `/opt/clip-int/specs/clip-rm/clip.spec.xml`.

**Ce qu'on ne veut pas**

Le paquetage secondaire `net-dns/bind-tools` dépend du paquetage `dev-libs/openssl`. Si le paquetage `bind-tools` est mis à jour, on ne veut pas que celà entraine la mise à jour de `openssl` car la mise à jour des paquetages primaires est bloquée (on ne peut mettre à jour les paquetages primaires qu'à certains moments de la vie du système).

Dans le fichier *ebuild* :

- le drapeau `clip-deps` permet de supprimer les dépendances à des paquetages primaires
- le drapeau `core-deps` permet la même chose mais uniquement pour la distribution *clip*
- le drapeau `rm-deps` permet la même chose mais uniquement pour la distribution *rm*

Lors d'une mise à jour, il est préalablement nécessaire d'identifier les anciennes modifications de l'*ebuild* relatives à CLIP (cf. 1.3 et 1.2).

Vérifier que des patches s'appliquent bien ne consiste pas seulement à constater qu'il n'y a pas d'erreur lors de leur application, mais de comprendre leur implication et les évolutions du delta de la nouvelle version du code a patcher.

**Comment procéder ?**

Ensuite, il faut modifier les directives `DEPEND` et `RDEPEND`. Dans l'exemple ci-dessous, le paquetage dépend de `dev-libs/openssl` primaire uniquement dans *clip*, et de `sys-libs/glibc`, primaire dans *clip* et dans *rm* :

```
DEPEND="ssl? ( dev-libs/openssl )
      xml? ( dev-libs/libxml2 )
      idn? (
             || ( sys-libs/glibc dev-libs/libiconv )
                  net-dns/idnkit)"
```

Or il faut supprimer les dépendances à ces paquetages primaires ; on obtient :

```
DEPEND="!core-deps? ( ssl? ( dev-libs/openssl ) )
        xml? ( dev-libs/libxml2 )
        !clip-deps? ( idn? (
                              || ( sys-libs/glibc dev-libs/libiconv )
                                   net-dns/idnkit) )"
```

Les indicateurs `!` et `?` veulent dire que si `core-deps` n'est pas défini (grâce à `IUSE`), alors la dépendance est acceptée.

Il faut terminer en éditant la directive `IUSE` pour rajouter les drapeaux dont on a besoin (attention à l'espace de concaténation) :

```
IUSE="doc idn ipv6 ssl urandom xml"
IUSE+=" clip-deps core-deps"
```

Afin de faciliter les futures mises à jour, il est important de modifier l'ebuild de manière à avoir un « diff propre ».

### A.4.6 Créer le fichier `ClipChangeLog`

Le ChangeLog (cf. 1.1.2 et B.2) permet de suivre les modifications qui ont étés apportés à un paquetage et leurs raisons (e.g. bug, sécurié, fonctionalité...). Il faut le mettre à jour à chaque changement de l'ebuild, nécessitant une montée de version ou non.

### A.4.7 Mettre à jour le fichier `Manifest`

La modification d'un fichier du répertoire où est l'ebuild doit être suivie d'une mise à jour du fichier Manifest :

```
# ebuild xournal-0.4.2-r1.ebuild manifest
```

Toute modification ayant un impact sur l'installation d'un paquetage nécessite une incrémentation de release ou de version.

## A.5 Tester la compilation et l'installation du paquetage dans clip-sdk

Cette étape est facultative. Il s'agit de la compilation et de l'installation du paquetage dans le clip-sdk :

```
# emerge -av xournal
```

L'étape précédente n'est pas entièrement facultative car, par exemple, pour compiler et packager Xaw3d, il faut `xmkmf` du paquetage `x11-misc/imake`. Si ce dernier n'est pas installé, la commande `clip-compile` échouera. L'appel à emerge permet de résoudre simplement ce problème.

En cas d'échec, la commande ci-dessous montre les informations de dépendance :

```
$ qdepends -d xournal
```

Pour faire un `emerge` sans utiliser certaines fonctionnalités :

```
$ USE="-cups" emerge -av xournal
```

Si jamais il y a besoin de changer le fichier `Manifest` :

```
$ ebuild xournal-0.4.2-r1.ebuild manifest
```

Le répertoire de travail est défini par la variable *PORTAGE_TMPDIR* du fichier `/etc/make.conf`, par défaut `/var/tmp/portage` ou, si la machine a assez de mémoire, il est conseillé d'utiliser `/run/shm/portage`.

## A.6 Créer le paquetage Debian

La compilation et la construction du paquetage comprends plusieurs étapes. Tout d'abord, éditer les dépendances des *nouveaux* paquetages dans `/opt/clip-int/specs/clip-rm/rm.spec.xml`.

Puis construire le paquetage (*.deb*) :

```
# clip-compile clip-rm/rm -pkgn app-text/xournal
```

Afin de faciliter la création récurrente de paquetages, il est conseillé d'utiliser la commande *clip-make build* et un fichiers listant les paquetages nécessaires (Cf. `/opt/clip-int/pkglist/*.conf`).

Examiner le contenu du paquetage :

```
# dpkg -c ~/build/debs/rm/xournal_0.4.2.1-r1_i386.deb
```

Si des fichiers apparaissent ailleurs que dans `/usr/local`, il est nécessaire de faire un patch (statique, ou dynamique avec la commande *sed* par exemple, dans la fonction `src_prepare` de l'*ebuild*).

## A.7 Créer un patch

Pour obtenir l'arborescence de travail :

```
# ebuild xournal-0.4.2-r1.ebuild prepare
# cd /var/tmp/portage/app-text/xournal-0.4.2-r1/work/xournal-0.4.2-r1
```

Création du patch :

Afin de comprendre facilement la fonction du patch, il faut suivre la règle de nommage suivante : *${PN}-<version>-<fonction>.patch* (cf. 1.4).

```
# diff -urpN xournal-0.4.2-r1.orig/ xournal-0.4.2-r1 > xournal-0.4.2-description.patch
# cp xournal-0.4.2-description.patch /opt/clip-int/portage-overlay/app-text/files/
```

Il est conseillé d'utiliser l'outil *quilt* pour créer, tester et faire évoluer les patchs plus facilement.

Éditer le fichier *ebuild* pour ajouter une directive afin d'appliquer le patch :

```
src_prepare() {
    [...]
    epatch "${FILESDIR}/${PN}-1.2.3-figparserstack.patch" #297379
    epatch "${FILESDIR}/${PN}-1.2.3-spelling.patch"
    [...]
}
```

Puis reconstruire le paquetage :

```
# ebuild xournal-0.4.2.1-r1.ebuild manifest
# clip-compile clip-rm/rm -pkgn app-text/xournal
```

### A.7.1 Modifier un chemin incorrect

**Autotools**
Un chemin qui pointe hors de `/usr/local` peut être corrigé à plusieurs niveaux. Un petit rappel sur les *GNU autotools* :

1. `automake` lit `Makefile.am` et génère `Makefile.in`.
2. `autoconf` se base notamment sur `configure.ac` pour générer `configure`.
3. Finalement, `configure` se base sur `Makefile.in` pour générer le fichier `Makefile`.

**Xmkmf**
Si le paquetage utilise `Imakefile` et `xmkmf`, le fichier *ebuild* contient parfois une directive spéciale permettant de spécifier les chemins sans avoir à créer de patch :

```
sed_Imakefile() {
  # see fig2dev/Imakefile for details
  vars2subs="BINDIR=${CPREFIX :-/usr}/bin
      MANDIR=${CPREFIX :-/usr}/share/man/man$(MANSUFFIX)
  ...
}

src_prepare() {
  ...
  sed_Imakefile fig2dev/Imakefile fig2dev/dev/Imakefile
  ...
}
```

Il ne faut pas modifier en dur les chemins utilisés par l'*ebuild* mais utiliser les variables adéquates : `${CPREFIX:-/usr}`. La variable CPREFIX aura la valeur `/usr` pour les paquetage coeur (*clip-core* et *rm-core*) et la valeur `/usr/local` pour les autres paquetages (*clip-apps* et *rm-apps*).

Dans ce cas, ne pas hésiter à fouiller dans les fichiers `Imakefile` pour rechercher les macros à insérer dans la directive ci-dessus ou à patcher. Pour savoir quelles sont les macros, lancer la commande `xmkmf` génère un fichier `Makefile`. Les macros utiles sont dedans :

```
# ebuild xfig-3.2.5b.ebuild prepare
# cd /var/tmp/portage/media-gfx/xfig-3.2.5b/work/xfig.3.2.5b
# xmkmf
# less Imakefile
# less Makefile
```

## A.8 Tester le paquetage sur CLIP

Pour tester le paquetage, il ne reste plus qu'à l'installer manuellement sur CLIP, à partir d'une clef usb avec la commande `dpkg -i`.

Il faut pour celà que le poste soit « instrumenté », et notamment que les réductions de capacité dans /etc/init.d/reducecap soient commentées (Cf. [CLIP 4002]).

Pour monter la clef usb, on suppose qu'elle est sur `/dev/sdb` :

```
# mount /dev/sdb1 /mnt/usb
```

Déposer le paquetage dans la cage *rm_b* :

```
# cp /mnt/usb/xournal_0.4.2.1-r1_i386.deb /vservers/rm_b/update_priv/var/tmp
```

Pour installer le paquetage, il faut entrer dans la cage `/update` situé dans le *vserver* de *rm_b* :

```
# vsctl rm_b enter -c /update
```

On peut ensuite installer l'archive :

```
# dpkg -i /var/tmp/xournal_0.4.2.1-r1_i386.deb
```

Voilà, l'application peut maintenant être testée !

Ne pas oublier d'enlever le paquetage (ou de réinstaller la version antérieure) par la suite :

```
# dpkg --purge xournal
```

### A.9 Mettre à jour Subversion

Une fois que tout fonctionne, il faut ajouter et commiter l'arborescence *ebuild* dans Subversion. Tous les changements inter-dépendants doivent être commités en même temps (pour rester cohérent, faciliter la maintenance et les éventuels backports) :

Ne pas oublier d'avoir un `Manifest` à jour avant de commiter.

```
$ cd /opt/clip-int
$ svn up
$ svn add portage-overlay/app-text/xournal
$ svn rm distfiles{,-dev}/xournal-0.4.4.tar.gz
$ svn add distfiles{,-dev}/xournal-0.4.5.tar.gz
$ svn st
$ svn ci portage-overlay/app-text/xournal distfiles{,-dev}/xournal-0.4.{4,5}.tar.gz
```

Lors d'une mise à jour, il est nécessaire de supprimer les anciennes versions et fichiers devenu inutiles (`svn rm` pour les fichiers *AUX* et *DIST*).

Il y a donc au minimum 5 chemins à spécifier pour une mise à jour : dossier de l'*ebuild*, anciens et nouveaux *distfiles* et leurs liens symboliques dans *distfiles-dev*.

Il ne reste plus qu'à respecter les conventions de message de commit (cf. B.1).

## B Conventions typographiques

### B.1 Message de *commit*

Le message de *commit* doit respecter l'expression rationnelle « `^ticket \d+: .+` » afin que les changements soient liés au ticket correspondant.

La description en **français** (sans accents à cause du Bugzilla) doit permettre de comprendre les modifications qui ont étés apportées sans nécessiter de regarder le contenu du commit.

Il est recommendé d'indiquer sur la première ligne, à la suite de la référence au ticket, un titre court et d'apporter des précisions à la suite (en laissant une ligne vide).

Quelques exemples courants :

– Dans le cas d'un commit de tag il faut seulement indiquer le nom du tag (e.g. « *ticket XXXX : foo-bar-1.2.3* »). Attention toutefois à ce que ce commit soit uniquement une copie Subversion (faire `svn up` avant `svn cp`).

– Pour une mise à jour d'un ebuild non modifié, typiquement lors d'une mise à jour d'un logiciel de *clip-dev*, il peut être suffisant d'indiquer « *ticket XXX : Mise à jour <nom-ebuild> dans portage* » étant donné que la description des modifications doit déjà être présente dans les commits précédents (dans *clip-dev*) liés au même ticket.

### B.2 Message de `ChangeLog`

Le contenu des fichiers `ChangeLog` et `ClipChangeLog` (cf. 1.1.2) doit être en **anglais** et respecter les conventions Gentoo [DEVREL] (nom, version et date de l'ebuild, nom des fichiers ajoutés, modifiés ou supprimés, description, texte sur 80 colones...).

La commande `clip-bump` permet d'apporter les modifications minimales nécessaire pour mettre à jour un ebuild et inscrire un squelette de ChangeLog dans le bon fichier.
Par défaut, c'est une modification seule de l'*ebuild* (changement de *release* ; *rbump*) mais on peut préciser un saut de version simple (option `-n`) ou spécifique (option `-v 1.2.3`).

## C Les droits sur CLIP

### C.1 VeriCtl

Certains paquetages fournissent des applications ayant par exemple besoin d'ouvrir des sockets réseau (ce qu'un binaire n'a pas le droit de faire par défaut sur CLIP). C'est l'interface *VeriCtl* (cf. 2.4) qui fournit ce droit là.

Pour ajouter le droit, modifier l'*ebuild* et ajouter les entrées suivantes :

```
inherit verictl2

[...]

pkg_predeb() {
    use clip-rm && doverictld2 "${CPREFIX :-/usr}/bin/host" e - - - c
}
```

Attention à ne pas oublier `inherit verictl2` au début de l'*ebuild* !
Les *flags* de `doverictld2` sont (très) résumés ci-dessous :

```
doverictld2 "${CPREFIX :-/usr}/bin/host" e - - - c ccsd
            ^                            ^ ^ ^ ^ ^ ^
            |                            | | | | | +--- algo
            |                            | | | | +--- privs (c  :CLSM_PRIV_NETCLIENT)
            |                            | | | +--- cap_i
            |                            | | +--- cap_p
            |                            | +--- cap_e
            +--- name                    +--- flags (e  :VRX_FLAG_EXE)
```

Par exemple, pour le paquetage `arping` l'entrée suivante est utilisée :

```
doverictld2 "${CPREFIX :-/usr}/sbin/arping2" er 'NET_ADMIN|NET_RAW' - 'NET_ADMIN|
    NET_RAW' csN
```

## Références

[BUSYBOX] *Busybox*. http://www.busybox.net/about.html.

[CLIP 1101] Documentation CLIP, 1101, *Génération de paquetages*, ANSSI.

[CLIP 1103] Documentation CLIP, 1103, *Environnement de développement*, ANSSI.

[CLIP 1201] Documentation CLIP, 1201, *Patch CLIP-LSM*, ANSSI.

[CLIP 1203] Documentation CLIP, 1203, *Patch Grsecurity*, ANSSI.

[CLIP 1301] Documentation CLIP, 1301, *Séquences de démarrage et d'arrêt*, ANSSI.

[CLIP 1302] Documentation CLIP, 1302, *Fonctions d'authentification CLIP*, ANSSI.

[CLIP 1304] Documentation CLIP, 1304, *Cages CLIP*, ANSSI.

[CLIP 1401] Documentation CLIP, 1401, *Cages RM*, ANSSI.

[CLIP 3101] Documentation CLIP, 3101, *Liste de configuration*, ANSSI.

[CLIP 4002] Documentation CLIP, 4002, *Outils et debug*, ANSSI.

[DEVREL] *Gentoo Developer Handbook*. http://www.gentoo.org/proj/fr/devrel/handbook/handbook.xml.