The redesign of pkg_install for pkgsrc

Jörg Sonnenberger

October 15, 2006

Pkgsrc is a framework for building third party software on a variety of systems. It is the system of choice on DragonFly and NetBSD.

Pkgsrc was originally derived from FreeBSD ports and many features were added to that foundation. One central component is "pkg_install", a collection of small programs to install and remove packages and other related tasks. While it has been extended over time, the original code base is still mostly present, together with a number of limitations.

During Google's Summer of Code 2006 program this component was rewritten to better fulfill the needs of pkgsrc:

- Integrated archive handling.
- Full specifications of file formats and algorithms.
- Versioned, extensible meta data.
- Better integration of the install framework.

In the paper a comparison of the old approaches, the new solution and the rationale, as well as the state of integration in pkgsrc and of the conversion tools are given.

1 Introduction

The NetBSD Packages Collection or "pkgsrc" is a framework for building third party software. Over the years it was extended to support not only NetBSD, but a great variety of Operating Systems, ranging from Apple's MacOS X to Interix (Microsoft Services for Unix). Beside NetBSD, pkgsrc is the system of choice on DragonFly.

The pkgsrc infrastructure is originally derived from FreeBSD's ports framework. Many features like the wrapper system and buildlink were added over the years. One specific piece is "pkg_install", a collection of small programs to install and remove packages and manage related tasks. While it has been extended over time, the original code base is still mostly present.

Several problems have shown up with different severity, like

- use of external programs for the extraction of packages,
- use of a temporary directory during extraction, followed by moving/copying every file to the real location,
- missing documentation of file formats and precise syntax,
- redundancy of installation/deinstallation scripts,
- advanced updating facilities,
- incoherencies between packages built from source and those installed via binary packages,
- difficult interaction with high-level tools.

The Google's Summer of Code 2006 project provided an opportunity to work on redesigning "pkg_install" to fix most, if not all of the aforementioned problems.

This paper discusses the results in comparison with the older approaches and looks at the state of integration into the pkgsrc system.

2 Package metadata

2.1 Package patterns

The ability to match package names is needed in a number of situations. This includes dependencies and conflicts, but also checks for security vulnerabilities.

In pkgsrc four different pattern types are currently used:

- Plain package names form exact matches.
- Dewey patterns like "gdm>=2.14<2.14.8" consist of the package base name and relation operations. Version numbers are parsed according to a complicated rule set modeled after common practice.

- Fnmatch patterns allow shell-like wildcards ("pear-5.0.[0-9]*") and are most commonly used to match any version of a package.
- Csh-style alternatives ("sun-{jre,jdk}<1.3.1.0.2") are expanded to elementary patterns. If any of those matches, the alternative itself is matching.

All four types have at least one major limitation. Plain matches are actively discouraged, since they can't even deal with local patch versions ("estd-0.5nb1"), making them almost useless.

Csh-style alternatives are needed to handle multiple packages providing common functionality like ghostscript-afpl and ghostscript-gnu.

Dewey patterns are the most expressive pattern, but can't represent a match to all versions. Matching e.g. all sub-versions of 4.3 is a problem as well, since release candidates and patch level complicate the matter. Dewey pattern only work well, when upper or lower bound are precise. The old implementation also had some interesting validation bugs, e.g. "php<5>4" is matched by "php-4".

Fnmatch patterns have the downside of matching more than intended. If there's ever a PHP module which name starts with a digit, the common "php-[0-9]*" pattern for the PHP interpreter itself would match the PHP module as well.

The situation is complicated further as multiple patterns are sometimes used to reduce the number of matching packages. A dependency on PHP 4.x for example introduces at least two patterns: "php-4.4.*" to match the API and "php>=4.4.1nb3" to specify the ABI. The evaluation order by "pkg_add" for missing dependencies is critical. When the second pattern is evaluated first, PHP 5 would be installed and the first pattern would be unsatisfiable as PHP 4 and PHP 5 conflict with each other.

To reduce this mess a way to unify the four styles was needed. One more desirable criterion exists, which wasn't satisfied by the existing rules. "pkg_add" has to choose a package, when more than one package matches a pattern. As long as they have the same base name, a built-in rule is used (see PHP 4/5 earlier) which selects the highest available version. There's no deterministic rule for csh-style alternatives though. User interaction can be used to resolve such conflicts, but they are often either undesirable or unavailable (e.g. automatic package installation during bulk builds). The order should therefore follow explicitly from the pattern.

As most of the patterns in pkgsrc follow the Dewey-style it was useful to keep it as base. The generalized version consists of a package base name and zero or more operator/version pairs. Zero operators provide a full wildcard match and each pair is processed in order as long as they match. This means the incorrectly parsed pattern “php<5>4” now is valid and behaves as expected. Beside the normal relational operators “<”, “<=” and so on, “˜” is introduced as prefix match. “php˜4.4” matches “php-4.4”, but also “php-4.4pl1” and “php-4.4rc1”. Finally multiple of this simple patterns can be joined using “|” to form alternatives. Ordering of two matches is done by the first matching alternative first and by ordering the versions themselves if they match the same one.

While the given rules allow easy merging of two basic patterns, it gets more complicated, when alternatives are involved. As this is not typically used in pkgsrc (yet), the problems are left unresolved for now and will be revisited later. A possible solution is to consider a package version as matching only if it matches all requirements.

2.2 Dependencies, conflicts and compatibility

Packages often need other packages to function properly, e.g. because they are dynamically linked against them or call a program from them. In a similar way, some packages can’t work when installed at the same file. Historically two packages has to be marked as conflicting, when the package content overlapped, as the “pkg_add’ program didn’t handle it as failure.

Another use case of patterns are explicit compatibility hints. In pkgsrc the buildlink framework knows two kinds of dependencies – for ABI and API. The latter are the classic way to describe that a certain (minimal) version is needed by a package, e.g. because a new functionality was added in it. ABI dependencies are more complicated though. As dependencies are normally open-ended (all later versions match), it is hard to describe properly when the interface is compatible.

To solve this a package can explicitly specify what it is compatible to. So instead of requiring “libfoo>=1.0”, an exact match can be used by packages depending on libfoo. The maintainer of libfoo is now responsible for specifying what the oldest compatible version is. This can be used for ABIs as well as module interfaces in scripting languages like Python. Support for maintaining the compatibility list based on ELF “sonames” or libtool archives is planned.

2.3 Package lists

The heart of a package are the files within. The package list (plist for short) contains all the files in the package, which are supposed to be “static”. For each file a checksum is stored and it can be used to detect undesired modifications. The old plist format also contains modifiers to remove directories on removal and execute single line commands. The functionality to specify permissions or ownership existed, but was never used.

The old plists had three major issues:

- It contains some package metadata, but not all. The ability to execute commands was mentioned already. Another example is that dependencies and conflicts are listed in the plist. The on-line description, the full description, install and deinstall scripts, the package maintainer and all the other information are stored separately though.

  Checksums have been added as afterthought using special comments.

- Commands don't belong into a plist, that's what the install/deinstall scripts are for. Firstly, it increases the number of places to audit and secondly, it also provides a different environment.

- Handling of shared directories is flawed as it is often impossible or very unpractical to factor out a base package to "own" the shared directories. In the past most common directories have been created using mtree from a template and were considered sticky (e.g. never to be removed).

For the new "pkg_install", @exec and @unexec are no longer supported by unanimous consent. All non-plist rated information have been moved and the other statements have been made local to each entry. A field for checksums has been added as well as a field to tag entries to belong to specific classes. The latter allows special scripts to run on the tagged entries e.g. to register a font with fontconfig or add a texinfo page to the local index.

The second important change is the classification of entries. Inspired by the Solaris package tools, other types of plist entries beside simple files are support.

Configuration files are first-class entries. When the file does not exist at install time, it is copied from a template or created as empty file (e.g. for logfiles). On removal, the management tools can either keep it as is, remove it on user request or archive it for later use.

Similar to configuration files, volatile files have a template. They are not archived or even checked for modification, but instead assumed to be modified by the package at vim. This is useful for fixed indices like texindex's info/dir file.

Beside files directories can be contained in the plist as well. As the new "pkg_add" creates them on demand and "pkg_delete" removes them when no other package is referring to them, this is seldom needed. It is needed when empty directories should be part of a package or when special permissions are required.

Two special kinds of directories are also supported. Configuration directories can contain only configuration files and directories as entries and are a way to mark a whole directory hierarchy as containing only configuration files. They are supposed to be handled as whole (e.g. archived). Exclusive directories place a directory under the sole control of a package. No further plist entries are allowed and the system doesn't care about the content. The package is responsible for removing the content at deinstall time. This makes it possible to properly handle e.g. shared-mime-info's share/mime.

Last but not least are symbolic and hard links recorded. The former should not change its target and the latter might be converted down to a symlink if necessary, e.g. when target and plist entry are not on the same filesystem.

2.4 Essential and non-essential metadata

Some of the data attached to a package has been mentioned already – the package name, the list of dependencies and conflicts, the plist. Other items are:

- The prefix a package is installed to and which it is supposed to stay in with some exceptions,
- How to reach the maintainer of the package.
- The OS version and architecture the package was built.
- The license(s) it can be distributed under.
- The short and long descriptions, both in English and local languages.

All this data can be classified as essential or as non-essential. The former category describes what directly affects "pkg_install" and the basic user experience. Having translated descriptions is nice to have, but the English version will always be authoritative and required. Just because a field is essential doesn't mean that it has to be present though. A typical example is the license field which will be missing for most packages, but is critical for determining whether a package can be distributed.

The separation between both classes is useful as it reflects the need of correctly managing and preserving the meaning of a field. As the list of metadata will change in the future, backwards-compatibility will be needed. At the very least it must cover all the essential fields and those have be updated as easily as possible. To achieve this, each field has strong validation rules, which are relaxed for the non-essential metadata.

2.5 Package format

The old "pkg_install" just compressed tar archives containing all files in the plist and normally one file for each of short and long description, the plist, install and deinstall script, size infos. The latter set is also the metadata kept in the package database (typically /var/db/pkg or .pkg in the prefix).

For a typical installation this easily takes a few thousand inodes. To avoid the associated overhead, a format to keep them in one file was needed which doesn't compromise the extensibility. Two generic markup languages were considered, namely XML and YAML. Since white-space handling in XML is awful and YAML is also much human-friendlier, it was preferred by the author.

The serialized package content uses a shallow hierarchy which emphasizes the importance of the various fields. The package itself and the plist entries are explicitly tagged and thereby also versioned. This allows the package tools to easily detect and convert older versions when necessary.

Binary packages are still (compressed) tar archives. The content is different though. In the top level directory, there's an index file containing the serialized package description (as above). This is also required to be first entry of the archive. Signatures will be stored as second entry, but as no light-weight gpg verifying exists and X.509 certificates don't play nicely with the (current) setup of pkgsrc bulk builds, this is not finalized yet. After the index file the normal files from the plist are stored in plist order. The files are stored with the relative path under a directory named like the package. All other plist entries are synthesized during extraction.

Enforcing a strict order on the packages makes it possible to extract a tarball with minimal buffering and read the content without having to process more than the index size (up-rounded to compression blocks). The construct of using a subdirectory for the actual file allows later bundling of multiple packages into a single archive, with minimal changes.

3 The programming interface

The implementation of "pkg_install" consists of a library core and small bindings on top. The core consists of four major components: the pattern related functions, the package-related functions, the plist-related functions and the package database functions.

3.1 Pattern functions

The pattern API provide simple accessor functions for easy access in common situations. Both matching a pattern against a package name and ordering two package names with regard to a pattern are supported. The allocation and freeing of resources are kept internal.

The convenience functions are wrappers for the full implementation. Parsing of a package name or pattern is a separate task to allow later reuse. Functions to extract to the base package name or the list of matched base package names for a pattern are provided. Those are useful e.g. for a bulk build as they can reduce the quadratic runtime in the number of packages and patterns to linearly.

3.2 Package functions

The package functions deal with in-memory package description and related functions. Functions to create one from scratch or destroy it with freeing all associated resources are provided as well as functions to get or set the metadata. Multi-value fields can be read either using a temporary array or an iterator interface.

The finished package description can be validated either for basic compliance or for the full package conformance. Descriptions which pass the latter can be serialized using a callback interface. In the same way package descriptions can be read back and parsed. A function to create a binary package from a package descriptions and the files relative to given prefix completes the interface.

Errors are classified depending on whether they are input-related or internal. For internal errors like failing memory allocations or violations of the API contract, the program can provide a callback which is called with the current package descriptions, a failure code and optional context-depending arguments. The callback is expected to terminate the application, otherwise it is abort(3)ed. For input-related and other "soft" errors, a different concept is used. The error callback has the same arguments, but can return a value to decide whether or not the processing should continued. This is a ternary value–on error the processing can continue as long as it makes sense to diagnose further problems, but the initial error is sticky. Alternatively the processing will directly bail out. The callbacks are provided on package creation or parsing, it is not yet intended to modify them.

3.3 Plist functions

The plist API allows the addition and removal of individual entries. The interface is strongly typed and each type has independent accessor functions. The implemented makes heavy use of the preprocessor to keep redundancy in code minimal. Similiar to the generic package interface, the plist access is mostly done using iterative callback interfaces.

3.4 Package database functions

The database functions are still in the progress of being revamped. The desired interface has three components:

- Functions to query the database. This should be generic enough to work with package repositories as well.
- Functions to regenerate all internal state like the hash databases of all files and the forest of packages and their relationship.
- Functions to modify the database as set of add/remove operations.

The first category is rudimentary implemented by providing an iterator interface over all packages. The requirement for generalization is important here as the same functions to decide whether a dependency is installed can be used to find the best match in a binary repository. Most query functions should work on binary package repositories as well as the package database.

The second category is implemented, but has to be moved from the standalone command into the library.

The third category is the most challenging. Single add and remove operations work, but impose a severe limitations. Updates of non-leaf packages would have to either remove all depending packages or leave the database temporarily in an inconsistent state. To solve this, complex updates should be done as sets of add and remove operations, which are atomic from the point of the package database.

The downside is that the logic for verifying whether all dependencies are resolved, no conflicts are present and the plists of all to-be-installed packages are non overlapping gets a lot more complicated. As the use of index databases is still necessary for installations with multiple hundred packages, the usage of memory to keep the changes in memory is increasing as well.

It is open whether it is possible and helpful to split such transactions into minimal blocks, which keep the database in a consistent state. It will not help when e.g. xorg-libs changes, but is useful for the generic “update-my-system” case.

4 Integration and conversion

4.1 Staged installation

The first step for the integration of the new "pkg_install" is the elimination of direct installation into the prefix. This makes it much simpler to ensure that all directories created are either requested by the administrator or handled by the framework.

Another important desire is to ensure consistent permissions as many packages don't use the pkgsrc INSTALL_* variables, but random combinations of cp, pax/tar and install.

Therefore the facilities to install into a subdirectory of the working directory were added. As pkgsrc already provided just-in-time su, it was desirable to allow full user package builds. Many packages just use default ownership for files and the aforementioned override directives can be used to provide the functionality even in the old "pkg_install". Some care had to be applied for packages which install setuid/setgid binaries as the access permissions are extracted by tar and the ownership is later changed by "pkg_add", removing the setuid/setgid bits as side effect.

4.2 Pattern conversion

The need to convert old patterns to the new style is an independent effort. Both for the integration and the conversion patterns have to be converted, but it can mostly be done on demand.

As written in section 2.1, esp. fnmatch patterns are often not precise. A perfect automatic conversion is therefore not possible, but the intent of most patterns can be accurately represented.

The conversion mechanism is based on type-specific rules. Csh-alternative style patterns are expanded, each expanded pattern is converted and the list joined with "|". Simple package names are converted by replacing the last hyphen with "==". Dewey patterns are unchanged as they are a subset of the new grammar. The edge cases are working as humans would expect them, so the change in functionality is justified.

The most difficult case is the conversion of fnmatch patterns. For those a number of heuristics are used. The pattern is matched against regular expressions representing common use in pkgsrc. For example, when "^(.*)-\[0-9]*$" is matched, it means that the patterns applies to any version of the captured first sub-expression. As such it is converted simply to the that sub-expression. Other cases which are handled automatically are "php-4.4.[0-9]*"

and “php-4.4.*”, which are converted to “php 4.4¿=4.4”. “php-4.4nb*” are “php-4.4nb[0-9]*” are converted to “php 4.4nb”.

The given rules can be used to convert all but 30 patterns used by packages in the “pkgsrc-2006Q2” branch and the rest are all somewhat bogus special cases. It is not clear, whether they will end as hard-coded special cases or are left for human intervention.

4.3 The new pkg flavour

In preparation for better support of multiple packaging systems Johnny Lam refactored the package installation and creation code over the last summer. This dramatically simplifies the initial efforts needed for integrating a different “pkg_install” implementation. Using compatibility wrappers for “pkg_info” and “pkg_admin”, the changes are concentrated to two places:

- mk/flavour/pkg or a copy thereof
- mk/pkginstall

The former code has to be modified to use the new calling conventions and use individual arguments for each dependency instead of a space-separated list.

The latter code provides the install/deinstall script framework. Most of the functionality has to conditionally tag corresponding items for the new “pkg_install’ instead of expanding the shell scripts directly. This will be done incrementally to allow better testing.

As the interface of the package management commands is not finalized, the implementation of this code is still a work-in-progress and not part of the pkgsrc tree.

4.4 Converting existing packages and installations

The creating of package descriptions for testing a new implementation is tiresome and with the implementation of “pkg_create” a shell script for converting existing packages was written. This script has been extended over time to stay in sync with the feature set of “pkg_create”.

The biggest missing item right now is the handling of old install scripts. Those fall in one of two categories. Either they are created from the install/deinstall script framework or they are custom rules for a specific package.

The first class is relatively easy to handle as the scripts create individual entries in the package tarball or package database. The metadata can be extracted from the bottom of each file to handle appropriately.

The second class is more involved as there are no fixed marker in the scripts to annotate the beginning or ending of the common fragments (which are already handled). A second problem is that the scripts work both as pre-installation and post-installation scripts and the calling convention has to be emulated. It is an open question how far and in which an automatic conversion can be successful.

5 Conclusion

The redesign of "pkg_install" allowed fixing many of the problems of the old implementations. The installation of packages can be done in-place. The toolchain itself is much more self-contained, typically not requiring external programs, but for additional features. Building blocks for better high-level update mechanisms are provided. The modular architecture will allow further improvements and extensions with minimal redundancy and in a straight-forward fashion.

As side-effect of this work, pkgsrc itself has been improved in a number of ways. During the development of the pattern conversion tools, many bogus dependencies have been fixed. The staged installation has been desired for years and allows catching up with OpenBSD's ports system in that area. Beside the ability to build packages entirely as normal user, it will allow pkgsrc to subpackages as well.