Third-party software management under BSD

Andrew Pantyukhin <infofarmer@FreeBSD.org>

Preface

When I set out to write this paper in July 2006 I was a FreeBSD ports committer, determined to find something new in OpenBSD ports, NetBSD pkgsrc, as well as in a number of software management systems for Linux. I was hoping to find a way for the BSD community to exchange ideas with each other and to learn a lot from our Linux colleagues. Now, three months later, I'm still a FreeBSD ports committer, and I'm still hoping for us to work together, but I sure have gone a long way, longer than I ever expected to. The thing is, software management is developing so rapidly, you can never expect anything from it until you go and see for yourself what's happening.

In July I was pretty sure what I am going to write about, but a few weeks after I started the research, I was abashed by the affluence of information and I knew it was impossible just to describe solutions and discuss their implementation. In this paper, in addition to some factual background, in a clumsy, but purposefully informal and easy-going way, I try no more but to convey my own impressions from my venture into the world of package management. Note also, that my goal is not only to learn and observe, but also to take an active part in this vital world, to make people from distant projects notice and meet each other and hopefully to open new channels for communication.

Introduction

Operating systems come bundled with software. As removable media grows in size, leaving developers, trying to fill it up with code, far behind, we can fit more and more on a CD, DVD, Blu-ray Disc and what not. But while it seems to many end-users that somewhere there's a perfect combination of tools to cater to all their needs, they fail to see some simple points, exposing this illusion:

* However huge data storage is and however fast it grows, the number of software projects is overwhelming. With over 120 thousand projects at SourceForge alone, and many similar repositories amassing dozens of thousands more, it is absolutely clear why we just cannot jam everything into one distribution and present it to somebody other than a football-field-sized data center owner.
* We can greet the user with gigabytes of the most popular software in the world, and many Linux distributions do just that. But in our naturally heterogeneous IT world, there's always a great deal of unsolved problems. And once some piece of software answers a need, users want it. They won't wait until the next version of the whole distro, they won't wait until the packagers actually notice the new tool, the want it here and now.
* We can't pretend every user has enough resources to install a multigigabyte chunk of software just like that. There's embedded market where you need to enjoy your life on a shoestring, there are users with legacy hardware, there are users multibooting in 10 different systems, there are virtual private servers - and in each case any piece of software can be required, but it's not possible to install all the software at once.

Hence packages. Traditionally, package management can be integrated into packages themselves, into the operating environment. The first way is decentralized by design, and popular among commercial closed-source software vendors. They don't like to conform to cheaply-advertised standards or to wait for anyone to accept their package into a repository, so they just bundle their programs with installers, and sometimes deinstallers, and make it available as an executable. That's the way most packages come on proprietary desktop operating systems and many proprietary packages on other systems, and unfortunately that's the way to give your system administrator nightmares. The other way usually involves some guidelines and package developers, or packagers, have to take them into account in order to build a conforming package. Such packages are usually easy to install, deinstall and upgrade through a common interface.

History

Package management in UNIX

I'm not nearly old enough to recite my memoirs about how I got to try the first package management tool ever, and in fact those times went into oblivion years ago. Before package management existed, as we know it now, developers preferred spending their time troubleshooting installation issues to thinking about deinstallation. This approach became deeply rooted, and remains so to this day, in the Windows world. Back then a user usually had to get a file archive, extract it, optionally hack it and compile it, and install it. Surprisingly, today some administrators, especially those dealing with more obscure proprietary systems, regard this routine as quite straight and normal. Many operating systems came bundled with all the software you were supposed to ever need.

That's the way *BSD systems went, coming with rich userlands so that user might have a chance to never think about anything third-party. That's the way early Linux distributions were - it was all OS developers' job to decide what's important, compile it, integrate it and give you a nice versatile bundle.

But of course it couldn't stay that way for long. Eventually Unix got its System V (Solaris) PKG format and users started seeing binary packages, which they didn't have to hack or compile, or even extract. A simple pkgadd command would "transfer" the package to their system, and pkgrm would remove it. pkginfo and half a dozen extra tools were also there to constitute one of the first Package Management Systems (PMS).

Package management in BSD

In august 1993 Jordan Hubbard committed his package install suite, and almost exactly a year later he presented us with his new ports make macros, also known as bsd.port.mk. NetBSD imported the pkg tools in summer 1997 and later that year they took adopted the ports technology under codename pkgsrc (because the word "port" already meant a hardware architecture port in NetBSD). OpenBSD inherited pkg_install suite and ports from NetBSD, where pkg tools were rewritten in Perl by Marc Espie in 2003.

The FreeBSD ports system was just a facility to ease building binary ports, a collection of macros written in make, which later became a vital part of all three major BSD OSes.

Package management in Linux

Year 1993 welcomed Slackware, Debian, RedHat and Bogus distributions to the scene - and each came with its own PMS. By mid 1994 there were Slackware packages, Bogus'es modestly-named PMS, RedHat's RPP and Debian dpkg. RedHat later developed a new system called RPM, which are, together with Debian packages the two most popular PMS for Linux today.

When Gentoo Linux 1.0 was released in 2002, it came with a system called Portage, which was based on FreeBSD ports, but was powered by bash and python instead of make and shell. With over 11000 of official packages, it is the second largest centralized repository after that of Debian.

Today

Today there are dozens of systems, allowing to manage software on Unix-like systems in one way or another. Some divide them into binary-based, where you only deal with binary packages, source-based, where you install everything by compiling from source and hybrid, where you can do a little bit of both. In fact, all systems deal with source code at some point in time and all systems deal with binaries when the software is installed, so the real difference is in how they manage to compile software, install it and remove it.

Early PMS did not provide much help in compiling the code. More often than not, you were required to compile it by hand, move binaries to a special place - and use some tool to archive it along with some metadata into a package. That wasn't very pleasant job, especially if you consider the wealth of open source software and the frequency of updates. Ports makefiles, RPM recipes, Debian control files, Portage ebuild - are all there now to ease the task by automation and modularization of common actions.

You can hardly imagine building thousands of packages by hand, if you take into account that you have to do that for several versions of an OS, multiplied by several hardware architectures. Today in the FreeBSD ports system less than a megabyte of uncompressed core make macros make it possible for the other 375 megabytes of package-specific code perform this task with excellence, compiling over 15000 pieces of software, which amount to tens of gigabytes of non-bloatware source code, into packages.

BSD ports and especially Portage have very advanced macro systems, while RPM and dpkg mostly utilize separate packages to perform common actions. All these systems deal with pristine sources, i.e. they store all the information needed for the original upstream source code to be compiled into a package. Lots and lots of portability issues have resulted in many macros, which effectively unburdened thousands of software developers, and let us compile code written without much portability in mind without any showstopping trouble.

Of course, compilation is only one part of the whole picture. We can't just throw binaries at users, we have to make installed software easily available. For plain old console apps it just means placing binaries in a PATH-exposed directory. For daemons, we have to help user stop and start them at reboot automatically. For X11 apps

we usually have to install some Gnome- or KDE-specific files. And things just get more complicated when it comes to web- and sql-based software, and other modern software usage paradigms, like virtualization, clustering, and so on. Some of these issues are solved in many PMS, others are not even planned to be alleviated or even not recognized as problems, but believed to be there to entertain system administrators.

We'll now look at some popular contemporary PMS, at issues and solutions, at what users and porters expect from the infrastructure, and we will try to understand why there is so little collaboration between very similar projects and how people can start working together.

FreeBSD ports

Most of us know how FreeBSD ports system works in general. It's written in make and shell with every port having its own Makefile along with some other files, like patches and checksums, but the way we see it as multiple files in multiple multilevel directories is only a matter of organization. In fact, we could have had everything fit into a handful of makefiles and specify what port we're going to operate on every time we invoke make. There are countless ways to organize these hundreds of megabytes of code, and shell and make being comparatively simple languages, we've seen snippets ported from one to the other and back.

Core ports macros are located in a special Mk directory. They can be used by ports directly, or through the main macro package, bsd.port.mk, also located there, by setting special USE_XXX flags. A number of additional macros is located throughout the ports tree. In theory, you can create a port without using a single macro package, but macros ease the task immensely. You would have to program all the actions manually, from fetching, building and installing to creating a standard package. While most actions can be redefined, no port ever required to redefine all of them (there are over a hundred actions defined in just bsd.port.mk).

Thanks to macros most of the work is already done for you. In many simple cases, all a porter has to do is to write down a name, version, and download urls for a piece of software, along with a short description and list of files it installs - and the port is all ready. You can install it, remove it, make a package and submit it for inclusion into the official ports tree. But you only begin to experience the power of the ports system when you have some trouble with an app. You can solve most problems with a couple of tweaks, but there are hard nuts, when you spend hours trying to figure out what to patch and why does it segfaults at start. There's always room for automation, though. Many porters find themselves doing the same hacks over and over again, - and only reluctant to automate them because it's not that easy. Portage has a well thought-out eclass system to encourage streamlining all kinds of hacks, we'll look at it later.

NetBSD pkgsrc

Many users think that OpenBSD and FreeBSD ports are very similar, because they are both "ports", and NetBSD pkgsrc is something alike, but still different, because it sounds very different. In fact, like we said a bit earlier, pkgsrc would probably have been called ports if only the word "port" had not already had an entirely different meaning in the NetBSD project. It's a challenge to find out whether it was OpenBSD or NetBSD who has done more work on ports, but at first sight pkgsrc feels more like FreeBSD ports than OpenBSD ports do. It is probably because OpenBSD guys had rewritten the pkg_install suite from scratch (and renamed it to pkg_add to avoid a directory name clash during the transition). Along the way, they introduced many improvements into the infrastructure, as we'll see in a minute.

Now NetBSD still uses the original pkg_install suite, although John Kohl has contributed to the code and the refinements made it to FreeBSD. pkgsrc also got many interesting features, to name a few random ones:

* Licensing notion - ports refer to license names, which are located in a separate common directory. A user can then restrict ports to a subset of known licensing options.
* print-PLIST target - simple, but nice automatic plist generation tool, it uses "find -newer" and some awk/mtree magic
* Good documentation - pkgsrc.txt is a comprehensive guide for users, porters and developers
* Buildlink3 - symlinks required libs and headers into WRKDIR at pre-build
* builtin.mk - decide if system or installed lib should be used
* pkginstall framework - some common tasks for install/deinstall scripts have been automated, like user/group creation, dealing with config file
* pkg options framework - options have been reworked to allow for easy customization
* more flexible subst framework
* <vanity>policy-prodded unique dist_subdirs for rerolled distfiles</vanity>

OpenBSD ports

Like I just mentioned, OpenBSD ports infrastructure seems to have changed a lot since it was inherited from FreeBSD. The fact is it might have experienced much less development than pkgsrc has, but the changes affected it in a more visible way. And that's what any infrastructure should probably be aiming at: little changes in the core producing much positive effect in the consumers.

* Fake build environment - when you install a port, it is first installed into wrkdir (fake root), then package is built and only then it is installed
* Immaculate documentation - many comments made it from makefiles into manpages, many concepts are now described in dedicated manpages
* Options reworked into flavors, a little less flexible, but a lot cleaner mechanism
* Multi-packages - building several packages from a port the smart way
* Packages with different options or different subpackages in a multi-package have different file names
* You can act on several ports in a go, grouped by package name, category or maintainer
* Locking-supported parallel builds
* Built-in updating support

The whole pkg_install suite has been rewritten in Perl, and became arguably a lot smarter. I won't discuss it right now, but the ultimate target is to integrate most package management tasks into the base system.

Other Worlds

Many of us remember that there's much more to operating systems than BSD, some even recognize the word Linux when they hear it. Apart from BSD ports there are three big package-management players in the Unix-like world: RPM (RedHat, SuSE), dpkg (Debian, Ubuntu) and a rising star named Portage (Gentoo). And there are dozens of other most diversified approaches, which I won't discuss in detail, but will mention when I talk about some interesting feature.

RPM is probably the best-known package format in the world. It is associated with a package manager of the same name. RPM can run on most Unix-like systems and has been employed as the built-in manager in many Linux distributions. Binary RPM packages are built from source ones, which usually contain pristine sources, patches and a spec file, much like a BSD port's makefile. There is no central repository of macros, so packagers are restricted to RPM's built-in functionality. Binary packages from one system are usually unusable on another, or even on a different version of the same. Unfortunately, source packages usually obey the same rule, which limits RPM in its success as a universal package manager. When vendors publish packages, they usually have to provide one for each OS it is supposed to run on. There are efforts under way, most prominently Linux Core Consortium, which is behind Linux Standard Base, to alleviate the problem of incompatible packages.

Dpkg approach, also known for it's .deb packages, is a lot like RPM, but thanks to rigorous packaging practices has much fewer compatibility issues. Binary packages from one Debian-based system usually run on another one. Lately there have been some issues with Ubuntu, the most popular Debian derivative, about package compatibility. We can only hope that Ian Murdock will do everything he can to prevent RPM chaos from coming into Debian family (he's also working on LSB), while we discuss some other dpkg highlights. Documentation is extensive and quite impressive, leaving no room for questions from a novice, but the thing packagers profit the most from is probably debhelper suite, and lately Common Debian Build System (CDBS). Debhelper is a collection of tools which can be called from rules files - makefiles controlling how package is built. CDBS is a collection of macros packages, much like dot.mk files in BSD infrastructures. They can be included into rules files to use predefined targets and other handy make macros. CDBS builds on debhelper, and together they can bring packagers more convenience then ports currently do.

Last but not least is the youngest, most vigorous system named Portage, as a tribute to BSD ports. Its original developer, Daniel Robbins, took a foray into FreeBSD and later used the impressions he got to design a new PMS in Bash and Python, which is now the heart of Gentoo Linux. He did a great job at studying what other systems did, so he laid out a pretty slick design and implemented it successfully. Somewhat like RPM, Portage uses Bash scripts, named ebuilds, to control the building process. To provide debhelper and CDBS functionality, he designed a system of eclasses, also Bash scripts, which are a lot more fun to use than make macro packages. All in all, Portage does not introduce any revolutionary practices in PMS world, except for bringing home the source-based PMS can be a success on Linux, but its straightforward design and the power of Bash at its core attracted many developers and made it grow as fast as no one could expect.

Why Bother?

So there are BSD ports, Linux packages and a lot of other systems. Maybe we could take a look and learn something new, but at any rate, we should probably try to save our individuality and leave other projects well enough alone, diversity is good, right? Well, the problem is that no package management of today can cope with users' demands. Whatever OS one uses, she always meets some mishaps and shortcomings. First of all, there is enormous amount of open source projects. Whenever we tell a user "you'll find everything you need right here in our collection" we are lying. He'll be lucky to find a few most popular percent of currently available software, and he'll be very lucky to find that most of them are up to date and usable. And by only exposing the most popular programs, we are actually raising barriers for them to become popular in the first place. And by saying "you don't need that and that anyway" we begin to dictate opinion.

And the problems are not just in the numbers. It's a topic for another pile of papers, whether it's right or wrong to present users with a zillion of useless tools, whether diversity on its own is vile or virtuous. But there is so much more to both qualitative and quantitative metrics describing the way PMS serves its purpose. In a minute we'll start looking at some issues and solutions, and will hopefully discover that no project alone can embrace even a list of problems it would want to solve. Sometimes users are so loyal they mistake shortcomings of the systems they use for the way things should be, or even consider them advantageous. For instance, those who use binary-centric systems exclusively often frown upon source-based ones, because they are unaware of the problems which they can solve. And the other way around.

The interesting thing about packaging is that we all use the same software. At the operating system level, all we might care about is interoperability standards, implementation can work in ten different ways under the hood. But when it comes to third-party software - we're actually using the very same source code on all the different platforms. So while developers might pride on their distinctiveness and isolationism, packagers just can't do that. Be it FreeBSD, Linux or Mac OS, we should look for ways to work together, or we'll end up thinking that we're doing great when in fact we're suffocating both our users and software vendors. And the current situation of three BSD's working on three separate ports systems is just inconceivable. We're so close together we could fall on each other - and yet we find it much more comfortable to tweak things on our own.

We shall consider how to meet each other halfway later on, and now let's look at what's bothering us, what other and what other projects are having fun about.

What's up?

Scalability - Package Building

One of the main problems in any PMS is package building. Most porters acknowledge this, and the FreeBSD portmgr team could probably right an epic about it. Basically, FreeBSD package building cluster is a bunch of donated boxes. When building the whole tree takes desperate amounts of time, we ask for more hardware resources - and sometimes we even get them. As a result we've got one of the most up-to-date PMS trees out there and one of the most outdated package collections. Most Linux distributions don't seem to have these problems, but in reality they are just cheating. Fedora builds only the core, official packages, plus a generous amount of extras, - and lets users go find all the software they need anywhere else. Portage only builds at release times. Debian allows porters to build and upload packages themselves. BSD ports might have something to learn from each of them.

Firstly, traditionally we always try to build the whole tree, but we really don't have to. When it comes to a point when we just can't handle more package building, we either don't accept new ports or don't build them. Whichever is lesser evil is a hard question, but while we can handle a lot more code in our VCS there's no reason not to allow it to be added.

Secondly, also by tradition, we keep package building centralized. Centralization is always a two-edged sword. It keeps us from wasting coding efforts on redundant solutions, i.e. encourages collaboration, but it also demands non-trivial hardware resources when it comes to hungry tasks. At this point we can't just let porters build packages themselves and upload them, because it's a commonplace to customize build environments, but it's possible to automate standards-compliant builds, and in a way less painful than tinderbox to set up. Once porters can build standard packages, it can be automated. Everyone takes the ports she maintains and builds them on whatever archs she can, pkg_add's them, tries to run, uploads. Once the building part is automated, we can distribute tasks among both porters and non-porters. And distribution of hardware-hungry tasks seems to always solve the

problem. Of course, there are security ramifications to be thought about, but in general, we have to trust people. Package signatures will be a must, though.

System Resources: Using, Keeping Track

In a way, every PMS solves a problem of managing system resources, like disk space, file namespaces, user names, etc. It's just few people put it this way. When we think about a program which requires a specific user name, we imagine a script to create it at install, remove it at deinstall, spit out some warnings if the user already exists in our system and so on. Why don't we call it a resource and acknowledge that the app needs it. We might have one and we might not; some resources, like user names, can be shared between a number of different programs; some, like a TCP port at a specific IP address, usually can't. Whatever we should call a resource depends on our imagination.

To reiterate, among the things that can be actually spent or saved or wasted, programs usually require:

* disk space - this is ignored much too often, but it's far too important. A PMS of the future should probably provide a means of package-specific run-time disk space quotas, which are requested at pre-install time and prevent programs from filling up /var with logs and other similar issues. A user should also be able to view requirements of the packages she has installed, is installing or is planning to install, so that she can decide on her hard disk layout, or what to share via NFS, or numerous other points of administrative design.
* directory/file namespaces - facing a problem of having multiple instances of the same packages (of one or several different versions) installed at the same time, we should think about naming problems.
* user/group names/ids - many programs require a dedicated user name (and for security reasons we might want to encourage it where it is optional), some share it with other programs (e.g. many web apps share user/group with web server programs), but there's always a problem when it comes to adding/removing user accounts. There are ways to run a program under whatever user we like, so we should avoid hardcoding user id's or specific user name.
* TCP/UDP ports - we are accustomed to seeing ports as some hardcoded property of a program. In fact, almost any network-enable program provides a way to specify what port it should use. And we should leverage it in order to automate installation of several similar apps on one box.
* CPU, RAM, disk throughput, number of processes, number of open files, etc. - of course it would be cool if we could distribute performance based on priorities, soft/hard limits or otherwise between all the packages we have installed. Unfortunately, few operating systems have enough built-in functionality to implement that. Of course, we could employ some clever wrapper scripts or other hacks, but an efficient solution would still require OS support.

There's more to Resources

Now that we've seen how packages consume resources, why don't we allow packages to provide resources? Databases, virtual hosts, pixel-based on-screen real-estate, client connections to some persistent antivirus engine - you name it. Is it possible to automate it in a safe way? - Why not? And who could possibly do it in a better way than the maintainers themselves, who usually know more about their particular piece of software than most other users do. Of course, there are security issues to consider, but in fact many administrators choose less secure configurations in favor of more complicated ones - just because they haven't got enough time or zest. Apache runs chrooted on OpenBSD by default, it's not a port, but that's an accomplishment all the same. I doubt half of FreeBSD users chroot Apache by hand, in spite of all the security benefits. And what does it take porters to automate this setup? Probably less than it would take a new Apache user to do it the first time.

Of course, flexibility issues arise when porters try to make mandatory decisions for users. Well, it usually only takes one "if" clause to make some action optional. Moreover, porters should try to allow for many common choices. Let users prefer Postgres to MySQL, or database on another host to local one, and let applications take that preferences into account.

Customization

Resource management is tightly coupled with a more general problem of software customization, from setting preferences to applying useful functionality-enhancing patches. I must have installed phpmyadmin for a hundred times and almost each time I had to edit the configuration file to make the very same change - enable cookie-based authentication and set a blowfish secret. It would probably take less than an hour to implement some "with_" variable and automate the whole process. Many other web apps offer generous web-based installation wizards, but they always ask almost the very same questions. What would it take to let user say "I've got this database on this host with these admin credentials, please manage db/user creation for me"?

Sometimes programs require particular settings tweaked in other programs. A well-known example is php.ini settings. Should we make user deal with it herself or should we outline requirements and automate all the necessary tweaks if some super-manual-override mode has not been enabled in make.conf?

User interface

Most PMS have a unified interface to perform all the tasks related to software management. Here the simplicity of management contradicts flexibility and complexity of operation. I've always liked the way VCS clients deal with the problem. One main program, comprehensive easy-to-use help system, orthogonal switches, dozens of completely different functions performed by intuitive concise incantations. OpenBSD has taken pkg_install suite there (by rewriting it in Perl from scratch), Portage has emerge, Debian - apt. For a long time now FreeBSD has relied on portupgrade. Doug Barton has been working on a new tool called portmaster, written in shell, so that it can be integrated into the base system. But we have still to see a tool to let us customize ports. The way users are asked to set options now is strange to say the least.

Choosing what (not) to install

An easy way to say, what she wants to be installed, what she considers OK to be installed and what she doesn't want to be installed at all. At any given time, the PMS should know which of the installed packages are actually required by the user and which are installed as requirements for other packages. Sometimes it's important to be able to mark some packages not to be installed in any case. For example, a user might not want xorg libraries on a server with constrained resources - or just to keep system clean for that matter - and she would prefer some graphic app failing to install instead of having a bunch of heavy-weight packages installed.

Where do the old versions go?

FreeBSD ports pride upon being one of the most current repositories of open-source software in the industry, without having too much stability hassle. This makes it possible for all kinds of users to stay on the edge. But a lot of users require much more than just that. There are countless situations when an earlier version of a program is required. Most PMS, including BSD ports, try to solve this problem by providing several major versions as separate branches of a package. But what if a user requires an earlier version on a branch? Currently the only two solutions are to hunt for old packages or to downgrade the whole PMS. Both are good ways to mess up your whole system.

Portage has a lazy, but a better way to deal with it. They keep several versions of ebuilds (counterparts of our Makefiles) in directory of many ports. It's not very CVS-friendly, and they have to maintain each of the ebuilds, but it works.

Multiple problems arise when we talk about multiversioned ports and packages. To introduce full support into packages, we would have to redesign the whole concept of package dependencies. For the time being we might be better off leaving packages well enough alone, and leave them depending on a single version of each required package. The versions might be explicitly specified, designated as default in the dependency itself, or just the latest ones.

Metadata storage, as well as distfile storage are of particular interest. With metadata (makefiles, distfiles, patches and so on) we might go the Portage way and keep different versions in separate files. A more efficient solution might be to keep them on different branches in out VCS. As for distfiles - we may choose to drop support of unavailable versions, or, much better, mirror older distfiles. Of course, just to mirror them would put a substantial strain on disk space resources of our mirrors. Therefore, we should consider a possibility of keeping distfiles on vendor branches, also in out VCS. By all means, this repository may be separate from the one where we keep FreeBSD, but in fact it won't create much pollution in a change-set based VCS. Every update is just one changeset. As for digests, we'll have to keep per-file ones in addition to per-distfile ones. A successful solution will probably require extensive checkout capabilities, so that users could get a .tar.bz2 archive via http or ftp, containing all the sources of a particular version. It's not impossible, though again places additional load on the mirrors. On the other hand, per-file digests will make it possible to choose new compression algorithms in a trouble-free way. Some hosts might choose to offer LZMA-compressed checkouts, which will help users cut down on their traffic.

Repository-based PMS is not news. rPath, Inc. presented a system named Conary back in 2004. Conary implements a new vision of package management, proudly called software provisioning. It is based on the

concepts of dVCS, peering far into the future. There's even a Linux distro called Foresight based on Conary (not to mention rPath's own Linux of the same name). Unfortunately, Conary is not very active right now, but it has already generated a wealth of documentation for us to learn from.

Fetching sources

Speaking of distfiles, there are more ways today to get them than just fetch(1). People make software available in form of RCS files, anonymous VCS access, p2p shares and metalinks. We make porters think about that, and provide traditional archives via http and ftp links. Some distfiles can not be downloaded automatically because of licensing restrictions. In such cases we usually weed out the lazy users by telling them go to such and such url, register and download a file with such and such name. Instead we could present him with a text browser window and even a preregistered bugmenot-like account. Not that we should encourage the use of non-free software, but we don't make life much easier for users when we strongly discourage that.

Incremental distfiles

Some users still have very slow and/or expensive bandwidth. Many of them look at the rate our openoffice port is updated at and wish they could always have the current version, but they just can't afford downloading 300 Mb several times a month. But what if users could update their distfiles incrementally? A bzip2-compressed diff between OOo 2.0.4 rc1 and rc3 is about 1 Mb, which could result in 300 times less traffic consumed for the upgrade. And we already have a solution which takes care of the ports collection itself - portsnap. It's not an easy task to marry portsnap's concepts to distfile updates, and again, we have the problem of keeping the distfiles in a versioned environment. We even have a highly efficient bsdiff binary diff solution from Colin Percival, and some room for its improvement in a doctoral thesis by the same author, in case we decide to version-control closed-source or non-textual data.

Functionality providers

Many PMS (like Debian and Portage) implement so-called virtual packages, where several programs with similar functionality (e.g. mail clients, or web servers) are united into one, "provide" the virtual package. Several "providers" can be installed at the same time, one of them presented to the user via a uniform wrapper script, or a symlink (e.g. type "mail" and one of providers - whichever priority is highest - will be launched). Not only is this a user-friendly way to present some functionality, but also a convenient paradigm when it comes to other programs depending on some kind of facility, e.g. a web server or an MTA.

Multiple instances of the same program

Portage has a feature called slots, where multiple versions (branches) of the same package with different slot numbers defined can coexist on one system. FreeBSD also has this feature in form of version-suffixed port and package names. A little bit earlier we were talking about how cool it would be to have access to all versions of an app at once. Indeed, this is especially true in high-availability environments where you can't afford downtime and should test every new version before deploying it. While a separate sandbox is always advisable, why not just allow to install the new version on existing system without deleting the old one? This way a roll-back will only take a few seconds. Moreover, no matter what app we're talking about in most cases you'll be able to provide users with access to both versions at the same time.

Tobias Oetiker, the man behind the ubiquitous rrdtool and mrtg, has once been challenged with package management across 400 Unix workstations. Of course, he developed his own system named SEPP and his users were happy ever since. The fact is that whenever an upgrade took place, they always could launch the old version of a program. And before each upgrade they were given a chance to try out the new version. In fact they could keep all the versions they wanted for as long as they liked. Nowadays this has many security implications, but we'll talk about security later and now let's just say there's Debian which almost always applies security patches to older versions, so it's not impossible in practice. SEPP installs programs into versioned directories. Inter-package dependencies are supported, but Tobias recommends keeping everything a program requires in one package. It's impractical in most cases, but in some cases this approach can be beneficial. There are wrapper stub scripts and symlinks making software available to users, keeping statistics and providing for some additional run-time configuration.

There are other systems that keep packages installed in separate subdirectories. Keeping them versioned solves many issues, such as file namespaces we were talking about. There is system integration to think about - manpages, rc.d scripts can all be made versioned, but require non-trivial design decisions to be made. Last but not

least, if we allow several versions of a package to be installed simultaneously, why not allow same versions to be installed in the same fashion. It would not require much more - just a special instance identity to augment versioning and avoid name clashes. If we couple this functionality with resource management, running three daemons of the same version but with different patches applied will become a hassle-free operation.

As for running several instances of the very same binary - it can be also be achieved by launching it with different options. This has a benefit of run-time configuration as opposed to build-time in multiple instancing with different binaries. Unfortunately, some apps have hard coded values. They will have to be either patched to be run-time configurable or configured at compile-time only.

Movable packages

Whether installed into private subdirectories or not, it would be great to allow user to relocate a package installation from one place to another without disrupting it. The problem is reconfiguring it at run-time so that it is not surprised by new paths.

More run-time customization

Post-installation resource management is very important in dynamic environments. You have a web server listening on port 80, you install a new version and it listens on port 81. Once you verify that the new version works OK, you have to be able to exchange resources between the servers. Apart from port numbers, there might be different document root paths, database users and so on. Of course, we can make the user to just reinstall the daemon, but while a run-time reconfiguration should only take a few seconds, a reinstallation might require much more time.

Non-privileged package management

Non-root installation is advertised as the Holy Grail in the new breed of user-friendly clickety-clack systems, such as Klik and Zero Install. Any PMS could benefit from this functionality. Hacky solutions can be based on running PMS in a chrooted environment, but a real solution should introduce a notion of user installs into the core of PMS. Ideally users should be able to choose what to depend on: only system-wide packages, only user-installed ones or both. And run-time relocation and customization facilities should be able to make a user-specific package system-wide and the other way around.

Smart techniques could be employed to watch if more than one users install the same package and save disk space in some way. An even smarter, but a lot more security-encumbering solution would be to install all packages in system wide locations, but mark them available per-user and deal with per-user customizations. This would save space by default. VDS and shell providers will appreciate this kind of functionality.

Click!

One simple feature most users come to appreciate very much is the ability to do powerful clicks. I.e. you see a nice icon on a website, you click on it - and the next thing you know is a full-blown application installed and launched on your computer. The security implications will probably make some people pant, but smart design should yield some decent insulation. The matter is too many new packaging systems attract users with this kind of features. Even Microsoft gave in to the tide and announced its ClickOnce solution where there's no setup.exe, but only a mouse, a click and a working application.

Appliances

rPath, the company behind the aforementioned Conary system, advertises a concept of software appliances. Their marketing materials are quite vague, but the idea is simple. Instead of distributing your application alone, trying to make sure that it will work in many environments, you can marry it to an operating system, and distribute it as one package, guaranteed to work on most hardware configurations. rPath provides solutions for bare hardware based on their own Linux distro, virtual appliances to be used on virtual hardware and hardware appliances which is software appliance bundled with a computer. If we're talking about FreeBSD, we can extend this concept to jail appliances. You plug them in - and they just work. And you can plug a lot of them in a single system. Sounds exciting and in fact not staggeringly difficult to implement.

distcc/ccache

Portage and pkgsrc have built-in support for both distcc and ccache, two solutions to speed up the builds. Problem number one is getting ports to respect the designated compiler. Two is looking for issues that inevitably happen due to parallel compilation. Many users also report problems with ccache, apparently results of configuration issues. Built-in support means hassle-free automation, so all configuration problems should be sorted out with all kinds of environments in mind. Problem three is distcc on heterogeneous systems, i.e. setting up an old box running FreeBSD 4.x/i386 do all the building on a fast 6.x/amd64 system, or even on machines running another OS. This brings us to cross-compilation.

Cross-compilation

It has been accepted as a fact that whether distcc is involved or not, cross-building packages is not an easy task. Nevertheless, Krister Walfridsson, presented a new concept at EuroBSDCon 2004 and implemented it in NetBSD pkgsrc in 2005. His idea is to substitute the calls to some programs during the build process with calls to emulated programs. Granted, this depends on NetBSD emulation framework, but a similar solution might be feasible on FreeBSD.

Security

Vulnerability and exposure tracking is one of the most underdeveloped processes today. There are literally dozens of commercial, community and governmental security trackers, aggregators and copycats, but they are trying so hard to keep at a distance from each other that there's no reliable source of security-related information. Fortunately there is the CVE dictionary backed by the Mitre Corporation and the U.S. Department of Homeland Security. Most of the time it provides us with useful references so that we can say "a-ha, we are talking about the same issue". But neither does it provide comprehensive information about each particular issue, nor does it cover them all.

There is still no centralized community-based security database and PMS need it bad. Until such a facility appears, we'll continue maintaining our project-specific databases, which is not a completely lost, but mostly a wasted effort. When the database comes, each project can choose how to use it. You can either put references to fixed issues directly into packages or you can maintain a database with very simple entries: a reference to the issue in the central DB and affected packages. No descriptions, no reference hunting - these are centralized. But until version numbers and package names will become standardized, which I doubt will ever happen, PMS will have to maintain thin compatibility layers.

Collaboration

Education

PMS developers should take an active interest in other projects. That starts with learning about them. There are not numerous enough to bury you under piles of white papers, manuals and guides. Of roughly about a hundred projects only a couple of dozen have extensive documentation. The thing is not only that we shouldn't reinvent the wheel, but that some decisions we are going to make might have already proven to be ruinous in other systems. Also, whatever project we consider, its developers should recognize that the bulk of users are happy with other tools. Each PMS is only known to a fraction of users, which also means that most development and advances are happening outside. While some isolation seems to rectify our own in-house practices, which are so dear to us, it can only hurt by filtering all the most important outsider information. Recognizing the need for more sources of input is often a non-technical problem of developers' attitudes.

Spirit

This is a complicated topic. It's known for a fact that every good engineer has an itch to implement everything to his own liking, not out of vanity, but perhaps because you can only accept imperfections when you're the one that's responsible for them. Anyway, most PMS founders at one point in time felt that they would be better off writing something new from scratch or forking off a seemingly stagnant project, than talking to the developers of existing systems and trying to do something together. While this has unerringly resulted in most wonderful diversity, we're still at the point of having no solution to cater to even basic user demands.

So in order to move forward developers should probably accept, at least temporarily, that (1) it's not the time to start a brand-new project that is doomed to follow in others' steps and only stand out thanks to a shiny website or a few catchy taglines; (2) it's not the time to burn a bridge and fork, this will lead to either a suffocating dead-end where there are no interested users and developers or to another bridge left burning by yet another generation of

successors; and (3) it's not the time to keep isolation and work on your own. Learning from each other's mistakes and successes is good, but without interactivity little progress will be made.

We can't claim abilities to change how people are, so we'll just have to find people in other projects that are willing to communicate, hence -

Communication

A decade ago a newsgroup, a mailing-list, or some kind of other centralized communication method would probably do it. Today, it's hard to put such conventional limitations on the processes inside wide-scale highly-distributed communities. We have seen task forces, working groups, standardization committees proposing brilliant guidelines which were ignored altogether, because people are just too busy. Package management does not tolerate stagnation. With hundreds of updates out each day, we're like some news clerks, running around without looking sideways. But paying attention to what's happening over the hedge does not only help us find solutions in our everyday routine, it also makes us want to respond, to take part in foreign discussions. If we accept that we're part of the same process, subscribe to each other's mailing-lists, make comments in blogs, contribute to bug-tracking systems - and, most importantly, make acquaintances, get to know our colleagues by names - then it will truly be communication.