From 1.x to 11.x

# THE ONGOING EVOLUTION OF THE PBI Format

Since its very early days PC-BSD has used a unique form of package management, known as PBI or

**PUSH BUTTON INSTALLER**

BY KRIS MOORE

While this PBI format has changed and evolved with each major release of PC-BSD, the basic concept and principles have remained the same: to provide a way for applications to be installed on a system, in a self-contained manner, without introducing messy dependency resolution issues. In principle, this means that a user can safely upgrade or downgrade an application, such as Firefox, without having to worry about changes being made to the packages which make up the rest of the system, such as X, GTK, KDE, and others.

Figure 1 provides a simplified version of what a typical dependency-driven package system can look like. This is the model most common to both FreeBSD and Linux systems. While it has some benefits, such as reduced disk space, it also introduces a large element of complexity for any updating system to cope with. When a user wants to initiate an upgrade of a package, often it requires the package management system to resolve the upgrade of a tangled web of dependencies, which could easily touch hundreds of packages in a typical desktop application. In the example of Firefox, this may leave a new user perplexed as to why bits of seemingly "unrelated" packages have to be changed, such as GTK, Gnome, and others. Assuming the upgrade of all the packages is done properly, we are still left with another potential problem, that of new bugs or regressions. While an experienced computing user may be able to find a workaround to these problems, a more casual user will be left wondering why a simple update to their web-browser means that some else in their desktop stopped working.

Figure 2 shows a simplified version of how the PBI system interacts with the other software installed on the system. In this case, each application has been distributed as a "bundle," which is installed into its own directory, such as /usr/pbi/firefox, without affecting the existing package layout on the system. This makes the process of upgrading an application as simple as updating the contents of the bundle, either as a binary differential update or a complete bundle replacement. By eliminating the reliance on dependencies, the user is now able to add, remove, and update applications at will, knowing that at worst, only that particular application will be affected.

In PC-BSD, moving applications to this model has greatly improved system reliability and consistency. However, it has presented a number of technical hurdles to overcome. The first hurdle was dealing with the system bloat that comes with having many copies of the same files installed in different PBI bundles. In PC-BSD 9, this was addressed with the implementation of a "hash" directory. This directory is created and managed by a PBI daemon, which tracks shared binaries and libraries between PBI bundles.

In Figure 3, two PBI bundles both include an identical copy of libfoo.so.1. In this case, the PBI daemon moves the library into the hash-directory, with the file's checksum attached. It then creates a hard-link of the file back into each PBI bundle.

During the upgrade of a PBI bundle, the daemon again tracks the bundle's shared binaries

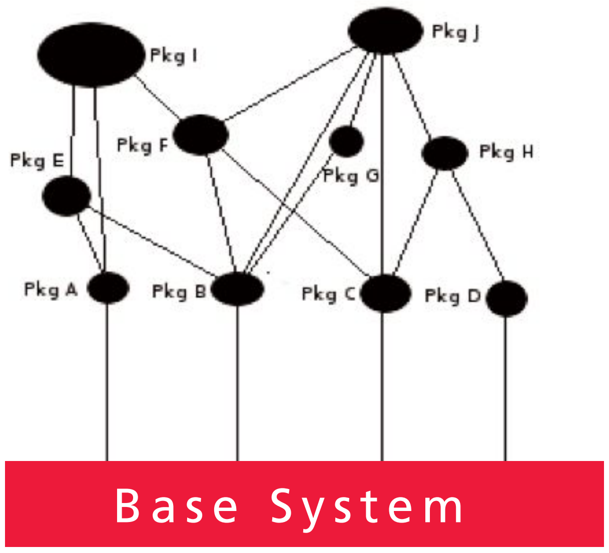

**Fig. 1.** A Typical Dependency-driven Package System.

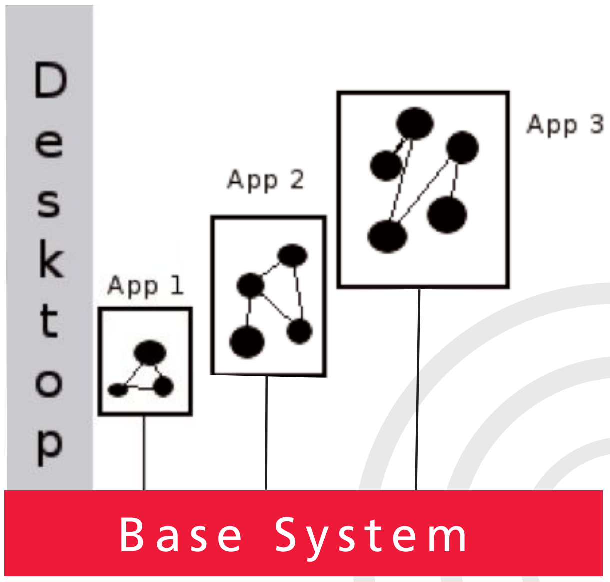

**Fig. 2.** PBI Application Bundles.

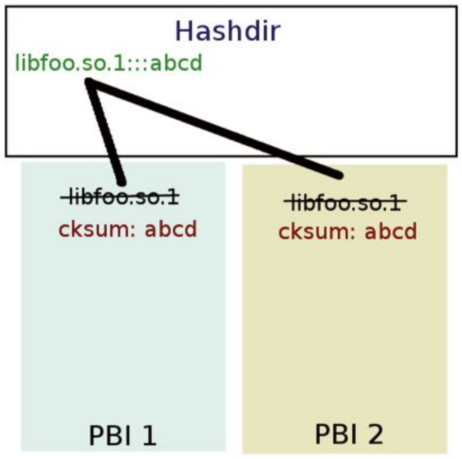

**Fig. 3.** Identical Files Linked to the Hash-directory.

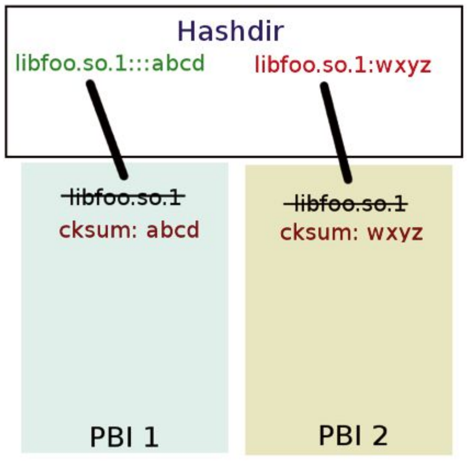

**Fig. 4.** Installing or Updating a Bundle with New Libraries.

and libraries. As seen in Figure 4, when a different version of an existing library is detected, the hash-directory is populated with a copy of the new library, which is again hard-linked back into the existing PBI bundle. The PBI daemon continues to monitor this directory, and when a file's hard-link counter drops to one instance, it is removed.

For the recently released PC-BSD 10.0, the PBI system has again undergone some refinement. For all previous editions of PBIs, applications were specially compiled from the ports tree using a custom LOCALBASE setting. This was done in order to force applications and libraries to behave in a self-contained manner, by having them look for their relevant data in the /usr/pbi/<appname> directory instead of the default system "/usr/local" directory. While this provided a working solution to self-containment, it presented some unique challenges. First, it exposed a number of bugs in ports and applications which expected their data to live in the system directories. Fixing these issues could range from simple to complex and proved to be very time consuming. Second, since packages were compiled with a custom LOCALBASE, each PBI needed to be compiled from scratch. This meant builders had to spend many cycles re-compiling the same libraries just to get some different location variables linked into the resulting binaries, libraries, and configuration files.

During the 9.x life-cycle, a number of enhancements were introduced to ease this building burden, such as ccache support, package caching, and more. However, going into the 10.0 release process we knew that for our PBI repositories to scale in size, we would have to fundamentally fix this problem. With the new pkgng and poudriere utilities making it easy to build the entire ports tree from scratch, we began looking at ways in which we could build PBIs from a single pkgng repository, but still keep the "self-contained" principles intact. The idea of being able to simply assemble a PBI from packages vs source building had the potential of reducing the build time for each PBI from hours down to a few minutes. However, this still left us with the run-time issues to solve. We experimented with many different options from using jails to running string replacement on binaries, but none of them proved a viable option. When looking into the idea of using jails, I thought: If jails can be used to virtualize an entire FreeBSD system, would it be possible to do some sort of "jail-lite," which only virtualizes the contents of "/usr/local," which is where our packages live? I had also seen several jail implementations that used some tricks with nullfs mounts to share a single FreeBSD world between multiple jails.

These ideas all came together when I began playing with the jailme utility in the ports tree. Instead of using calls to execute applications inside a jail, I modified the utility to do some nullfs mounting, replaced the calls to the jail with chroot, and created a "virtual" /usr/local space for a PBI to execute. And thus the idea of "PBI containers" was born. This immediately granted us the benefits that we sought: the ability to use a single set of packages, compiled with the standard /usr/local LOCABASE for assembling any number of PBI files. Additionally, it greatly reduces the complexity required for running applications, since we don't have to "force" applications to only load libraries from their own /usr/pbi directory. When the PBI is executed, it will only have access to its own files located in /usr/local.

This virtual /usr/local is accomplished is by using a wrapper binary in the front of the PBIs target executables. At launch of a target PBI, this wrapper will first check if the "virtual" container environment has been created. If not, it will proceed to perform some nullfs mounts, creating a replica of the running system in /usr/pbi/.mounts/<app>. Then the PBIs own /usr/local replacement will be mounted into this directory, replacing the systems version. With the virtual container now created, the wrapper binary will then re-create the containers ldconfig hints files in /var/run, preparing the new /usr/local directory for execution. Lastly the wrapper will chroot into the environment and execute the target application as called. This entire process can be done in usually a second or less, and only needs to be done the first time a PBI is run, making subsequent execution nearly instantaneous. The container environment stays active until the system is rebooted or the PBI is removed.

For applications that need to run commands outside the container, some callback mechanisms were added to allow running other PBIs or commands in the system's /usr/local space. These are mapped to the xdg-open and openwith commands and are typically used to open a file with user-specified application. For example, this would be used in the case of clicking a URL in some mail application and having that URL be passed along to the users default web browser.

With the runtime problems now solved, we went back and began to look at the benefits to the build infrastructure that this change bought us. The PC-BSD PBI repository for the 9.x series had grown to well over a thousand PBIs, and doing a complete rebuild from scratch easily takes 3-4 weeks on a modern server. This change reduced the build time of the same set of applications to only 48 hours!

With the release of PC-BSD 10.0 now complete, thought is already being given to the next stage of PBI evolution for version 11.0 or possibly 12.0. With the library de-duplication issue solved, and now the implementation of PBI containers, next on the roadmap is finding ways to reduce the download size of PBIs. Since PBIs are entirely self-contained, the total download size for each PBI is very large, often with the same files and libraries contained in each PBI file. While the de-duplication which occurs post-install fixes this issue for installed applications, we also want to find ways to reduce the total download size of a PBI file. This not only will save much downloading time for end-users, but also can reduce the total PBI storage footprint on our servers.

One possible solution is to make the PBI simply a "meta-file." This file would provide desktop and mime information, as well as a list of the packages that comprise the PBI. During the installation, this package list can be parsed and used as a blueprint for reassembling the PBI into a container, using cached pkgng packages. Since many of the packages used by PBIs are identical, this allows us to only fetch the missing packages needed for a specific PBI to be installed. Once the packages are all cached on disk, the installer can then assemble them back into an installed PBI. This method would greatly shrink the data being sent over the wire, but also opens up interesting ideas such as making PBIs customizable, by adjusting the particular packages that are reassembled into the container. This can all be done while preserving the integrity of the main system packages, which comprise a desktop or server installation, and other PBI containers. For users who still want the "offline" experience of copying a single PBI file to USB and installing it on a non-connected system, we can provide mechanisms to assemble the various meta-data and packages into a single file for installation.

While the PBI format has evolved greatly over the years, we are confident that it will continue to improve with each major release. All the while staying true to the vision of making the installation and upgrade of applications as simple and risk-free as possible. Users who wish to participate in this discussion and development are encouraged to get in touch with us on the PC-BSD developers mailing list. ●

---

**Kris Moore is the founder and lead developer of the PC-BSD project. He is also the co-host of the popular BSDNow video podcast. When not at home programming, he travels around the world giving talks and tutorials on various BSD-related topics at Linux and BSD conferences alike. He currently lives in Tennesee (USA) with his wife and five children and enjoys playing bass guitar and video gaming in his (very limited) spare time.**

## Further Reading on the PBI Format:

http://wiki.pcbsd.org/index.php/PBI_Manager/10.0

http://wiki.pcbsd.org/index.php/PBI9_Format

http://www.bsdcan.org/2008/schedule/events/81.en.html

http://bsdtalk.blogspot.com/2008/02/bsdtalk141-pbi4-with-kris-moore.html

http://2011.eurobsdcon.org/talks.html#moore