HEP/Nix Packages.

BV

[2015-08-15 Sat 09:16]

1 Nix in a Nut Shell

Nix is the package manager used by NixOS and can be used entirely from user space on POSIX'sh OSes. Its basic design is unique in the realm of software packaging (there is a similar, derived implementation called Guix which differs only in the configuration language used).

To begin to understand Nix, it is useful to compare and contrast it to systems which are already familiar to some in the HEP community which is done in the following sections.

1.1 The Nix Package Store

In contrast with other systems, Nix does not interleave the files of a package under a single-rooted file-system hierarchy such as the FHS (ie, the usual `/usr/{lib,bin,include}`) like Debian GNU/Linux nor does it produce isolated bundles which leverage a base file-system hierarchy such as in Mac OS X.

The Nix *package store* contains the directly installed file contents of Nix packages. It is somewhat similar to the "product database" of UPS, the module area of Environment Modules and the LCGCMT "externals" area. Some common features include:

- a well defined root directory containing the files of the installed packages.
- a per-package directory under this root named with identifiers used to later build a working environment.
- an FHS-like layout within each package directory.

The Nix *package store* differs from these other systems in some important ways:

- There is a **single directory root** common among all compatible installations. The prevailing convention is to place the store under `/nix/`. This must be a mount, not just a symlink. Other root directory conventions may be chosen but all packages must be built for that mount point.
- While a package's directory is named with the package name, version and other human-oriented identifiers it also contains a cryptographic hash formed from the dependencies required to build that package.
- A package directory is self-contained in that all files associated with the installation of that package reside in the directory (the others allow for, and in the case of UPS in practice, violate this).

These unique aspects of the Nix package store allow for a number of beneficial features:

- Packages can be shared in binary form with robust dependency resolution.
- Binary executable and library files do not need environment variables to locate dependencies due to the common path.
- Packages are precisely reproducible from source and binary can be validated. This allows for a wide pool of package builders to share binary packages with a wide community of users. This additional level of verification and validity is fully lacking in any other packaging system known to be used in HEP.

1.2 User Environment

Another unique aspect of Nix packages is how a user environment is produced. To speak generally, a user environment is created as some aggregation or view of a larger package store. This aggregation can be performed in a number of ways and typically relies partly on environment variables and the file system.

In the case of conventional GNU/Linux or Mac OS X user environments, the default environment is provided by the FHS-like directory hierarchy typically rooted in `/usr` and brief settings of a small number of environment

variables such as `PATH` to find executable files and relying on conventions "baked" into the loader to find libraries.

The packaging commonly used in HEP layers on top of this OS-default aggregation which relies heavily both on environment variable settings and some file system organization as described in the previous section. The environment settings typically extend the standard `PATH`-like variables, typically with one additional component for every package or project being aggregated. Many package require their own `PATH`-like variables (eg `PYTHONPATH`) or single-location variables (eg ROOT's `ROOTSYS`) to be extended or set. Also typical is to set one or more "standard" variables for every single package "just in case they might be needed" (eg CMT and UPS set a `<PACKAGE>_DIR` variable).

Heavily relying on environment variable-based aggregation is problematic for a few reasons:

- it is ephemeral, existing only in the shell session.
- its construction is ill-defined, depending on what setup scripts were run and in what order.
- its often confusing to users.
- it can lead to inconsistencies, particularly during development.

Nix takes a more file-system-oriented approach and NixOS is entirely based on this approach such that the entire OS-level environment is subsumed. With Nix, a user may have one or more "profiles". Each profile provides the files for the working environment and which are aggregated into a single directory that follows an FSH-like convention. The file-system hierarchy in the profile is recorded through symbolic links into the Nix *package store*.

This file-system aggregation has a number of benefits:

- the profile directory provides a tangible record of the user environment.
- user environment variables require minimal modification (single entry added to `PATH`-like variables).
- profiles can be populated in an indirect manner allowing site-wide or group-wide release definitions (eg, defining through a "pro", "dev" etc pattern, or through release definitions).
- atomic upgrades and rollback patterns are trivial.

- profile directories at a site may be automatically queried by site administrators to determine what packages are actually in use and which can be safely purged.
- profiles are amenable to `chroot` or similar containment.

1.3 Package Definition

Another way in which Nix is unique among the systems in use in HEP is that it is comprehensive in providing for configuration management, build automation, package distribution and installation and user environment management. None of the other systems managed, by themselves, cover this necessary ground.

Nix package definitions are short text files in the Nix packaging language. They provide parameters interpreted by functionality built in to Nix or provide any special shell commands needed for less common build methods. These specifications are highly portable due to leveraging a well characterized build environment - that of all the other Nix packages. This allows for a huge amount of shared effort not obtainable by other systems. For example, one individual or small group can maintain the Nix package specification for ROOT and the entire world can benefit from its use.

2 Impedance mismatches between Nix and HEP

At the heart of Nix is the *package store* and its contents are produced based on the other packages it already contains. Because of the hashing mechanism the location of this store is "baked" into the package names and binaries are built against their dependencies located under the *package store*.

The implication of this is that any package can be considered part of a "package realm" defined by the mount point. An apparently trivial change of this mount point requires rebuilding all packages. The benefit of accepting this limitation is that packages need not be /relocatable/[1] and that binaries built by one individual can be shared and need not be built by any other individual using the same mount point.

[1] The term "relocatable" is used in some systems like UPS. There, it takes a weak definition as the relocation is only possible through setting user environment variables. Nix provides a PatchELF tool with which one can produce binary executable and library files which can be truly relocated. It works by rewriting the path strings that are compiled into the binaries. If rewritten to their relocated locations the environment setting that must otherwise be modified need not be.

For NixOS installations this limitation is invisible as the system itself relies on the *package store* to be found at the `/nix` root. However, essentially all HEP computer installations do not run NixOS and thus a common Nix *package store* mount must be created in context of whatever native policies exist. The main issues with this are:

- many HEP users do not have administrator privileges on the computer systems they use.
- HEP clusters (and the wider Grid) have local policies driven by various forces and negotiations are needed to established a shared `/nix` mount.

There are two known solutions to this issue.

The first is to give up on a global *package store* mount point and leave it to each installation to pick one and simply spend the CPU time to populate the store by building packages from source. Some increased coverage could be had by exploiting the global AFS namespace. CVMFS is becoming more prevalent and with it another common convention for mount points is possible. Both provide a delivery mechanism although that is redundant with the one that Nix also provides.

The other solution is to use the functionality of PRoot which is a user-level `chroot` container mechanism. It is a light weight way to effectively produce a "virtual" mount in the context of a single job. The globally `/nix` mount can then be provided even if it may reside in a user's home directory, or a group location or otherwise not directly mounted. This virtual mount is achieved through some intercepted system calls and so some small performance overhead is suffered. Quantifying this overhead would be a necessary task to accepting this otherwise good solution.

2.1 Issues with existing HEP build systems

Some software is resistant to producing Nix packages. Because Nix is designed to build from source any software that is difficult to build from source is resistant to using in a Nix based system.

Of some small importance to HEP is proprietary, binary-only packages. To deal with this, the "source" of the build is the provided binaries. Nix has developed PatchELF to rewrite compiled-in paths to match the *package store* path, allowing the result to be truly relocatable.

More problematic are existing large, important software projects which have grown intricately intertwined with local computing infrastructure and installation patterns. An example is the `UPS/cetbuilttools` based software

largely developed at Fermilab for US HEP Intensity Frontier experiments. It is effectively not packages are not possible to build these package from "first principles" and instead they require, down to their low-level package build configuration and source code, intricate intertwining with the end-user environment management system (UPS). As a consequence, they are built for a small number of OS platforms and can only be built for a Nix-based system after some significant effort and buy-in by Fermilab.

3 Plan for a Prototype

To determine if Nix can be used in a practical sense in HEP, some prototyping is the first plan of action. The pat to this is:

1. Use the PRoot approach to provide `/nix`.
2. Work on package specification for common HEP packages, starting with ROOT[2].
3. Include releases of experiment-level package.
4. Develop methods and helper tools to develop on experiment packages.

4 See also

- notes on existing ROOT5 packaging
- notes creating new ROOT6 packaging

4.1 Links

- `http://christopherpoole.github.io/My-workflow-for-adding-packages-to-nixpkgs/`

[2]Someone has already packaged a recent ROOT 5 for Nix but its build happens to be failing at the time of writing.