tach - Simplified RPM package management

Gregory J. Kuchyt - <kuchyt25@potsdam.edu>

May 20, 2006

Abstract

The RPM package management system greatly simplifies the complexities inherent with managing program installations in the Unix world. RPM allows administrators to depart from the traditional sequence of '`configure, make, make install`' that is required with most source based installations of a program. It is, however, the common user who sees the greatest benefit from a package management system like RPM. RPM attempts to level the playing field with other more popular operating systems by providing a higher level method of installing and uninstalling programs. RPM provides the notion of dependencies, the idea that a package provides and requires resources. In order for a package to be installed its required resources must be satisified, likewise for a package to be uninstalled there cannot be a package that requires its provisions. The failing of RPM is that it will only report on the resource that prevents a given action, and not suggest what package is able to provide, or still requires a given resource. I propose to write an application that utilizes the RPM API and provides higher level functionality such as autuomatic resolution and installation of dependencies and "one command" updating of all system packages. I plan to build this application to utilize the same service and channel model that the now defunct Red-Carpet package manager utilized.

Contents

0.1 Introduction

0.1.1 RPM - RPM package manager

RPM is a package management system written for *NIX based operating systems, with a predominant hold in GNU/Linux distributions (e.g. Fedora Core, SuSE). RPM files, referred to as packages, contain pre-compiled binaries and other files a program needs to function. This avoids the tedious `configure; make; make install` process associated with source distributions of programs. In addition to the package's files and binaries, packages are bundled with additional meta-information about the program, such as descriptions and version information. This meta-information is stored in a data structure referred to as the RPM header[1]. One of the major portions of the header is dedicated to storing package dependency information. Dependency information is used in transactions, removal or installation processes, to verify that all the requirements of a package are installed in the case of installation, or that no installed packages will be affected by the removal of a package in the case of uninstallation. When a package is installed, the contents of its RPM header are parsed and entered into a database referred to as the RPM database. The RPM database contains indexed lists of commonly desired information for all installed RPMs such as files installed, dependency information, and package names. RPM has library bindings in C, Perl, and Python.

0.1.1.1 Dependency and Versioning resolution

RPM has a concept of package versions meaning it is able to detect which of two packages is newer based upon their version naming information. It is important to note that versioning is inherently dependent upon the person responsible for packaging the RPM. In this sense, RPM trusts that the packager has correctly adhered to the notion of versioning (i.e. updates increment versioning identifiers). For example, `tach-beta1-2` is newer than `tach-beta1-1` when these two version strings are passed to the RPM version comparison libraries.

Once RPM has been instructed to install packages, whether updates or not, it must resolve dependencies. RPM systematically checks all the packages' "requires" dependencies in the install transaction against the "provides" dependencies in the RPM database. If all the dependencies are met, the packages are installed, if not, error information about the unresolved dependencies is returned and the packages are not installed. The same process essentially occurs upon package removal, except no package is removed if the removal will break dependencies for other installed packages.

0.1.2 Red-Carpet

Ximian's Red-Carpet is arguably the hallmark for package managers. Red-Carpet was a software layer written on top of the RPM libraries that provided a simplified interface to the RPM sub-system. It provided both ease of use

and enterprise level features such as the notion of channels, federated package management, and automated package installation and updating. While other package managers for RPM such as Yum, Pup, and up2date exist, none brought to the table the usability of Red-Carpet. Mainly this is because the aforementioned package managers try too much to cater towards the typical user, and therefore curtail their feature-sets in the name of usability. Red-Carpet was able to maintain a simple level of usability and still provide a rich feature set unparalleled by any RPM package manager. Sadly, Novell's acquisition of Ximian has resulted in Red-Carpet being assimilated into Novell's Zenworks infrastructure.

0.1.2.1 Red-Carpet infrastructure model

Red-Carpet used the traditional client-server model, thus allowing for distributed package management capabilities. A Red-Carpet server running the Red-Carpet daemon (`rcd`) accepted connections from a client, and would download meta-information and packages from a package repository, or service provider. A service provider contains XML meta-information that describes itself, its channels and their target distributions and architectures, and each channels packages'. Communication with a service provider was done via HTTP, and client server communication was done through XMLRPC.

0.1.2.2 Red-Carpet features beyond RPM

- Channels with a knowledge of packages outside of those currently installed.
- The ability to search channels for packages that match a search string i.e. searching for any package that contains 'mp3' in its name
- Querying packages for descriptions of what they do and what features they provide.
- Automatic resolution of package dependencies during installation and removal.
- Automatic installation of available updates
- Distributed package management

0.1.2.3 torque

As mentioned previously, Red-Carpet required XML meta-information in order to function properly. This XML meta-information clearly needs to be generated from some source that is able to read RPM files and has a knowledge of channels and service information. Ximian's Red-Carpet Enterprise was their product which did exactly all of this. Red-Carpet Enterprise carried a hefty price-tag, for a product that did nothing entirely all that special. Thus, in the summer of 2003 I wrote an open-source equivalent, torque, to Red-Carpet Enterprise that used a MySQL back-end to store channel and package information used in XML generation.

0.1.2.4 tach

Due to the void left by Red-Carpet's disappearance from the package manager field, the purpose of this project is to create a Red-Carpet clone to build upon for future development of an enterprise class package manager. While Red-Carpet was written in both C and Python, I decided to write tach exclusively in Python for portability. Since RPM can be compiled on any *NIX, it would be nice for tach to be easily installed alongside RPM. The growing popularity of Python has caused more and more distributions to at least include Python, if not pre-install it from the base. Thus Python is a prime candidate language to write a portable application in.

0.2 Objectives

0.2.1 Alpha release objectives

- Provide command functionality comparable with Red-Carpet
- Provide elementary communication between a client application and a server application
- Provide elementary error reporting and handling

0.2.2 Beta release objectives

- Provide advanced error handling and feedback with extensive logging facilities, errors should not crash the daemon. Therefore extensive use of exceptions should be implemented.
- Detailed help information for supplied commands
- Major bugs in sub-systems resolved
- Smallest possible footprint on system resources

0.2.3 Production release objectives

- Bugs in major sub-systems (i.e. caching and dependency resolution) resolved
- Full featured application protocol that provides real-time feedback capabilities
- Security flaws in protocol and code resolved to a reasonable degree of application security
- Interface agnostic framework that will be equally easy to write a graphical application to use the same libraries as the command line client

0.2.4 Post-production objectives

- Gnome-applet, based on my previous rcd-applet[3], that allows users to manage tach via an applet.
- Graphical interface that allows package management through tach libraries

0.3 Design

0.3.1 Client-server network architecture

Tach uses a traditional client-server network architecture. A daemon (tachd) listens on tcp port 8585 for incoming connections. Upon receiving a connection and a message, which is potentially a command, tachd processes the message and decides whether or not it is a valid command. If it is, it executes the command and returns status information, if the message is not a valid command an error message is returned. When necessary, either by user input or an internal decision, tachd will communicate with service providers via HTTP to obtain data.

Currently, the client-server architecture is very primitive. It is essentially text-based, as commands are sent verbatim from the client input. Once received, the command is processed to see whether it is valid, if not an error is returned. The reason for the protocol's lack of functionality is that it is still in development, as this is currently a beta release. An objective in future beta releases is a full featured application protocol for tach that will allow for features such as bi-directional real-time communication. Currently, a command is issued and received by tachd. The command is executed and status information is returned to the client only after the command has completed (whether successfully or not). Real time communication will be useful to the end-user so they are informed of what is happening, and so they can send information back to the daemon to allow for real-time decision making. This is useful, for example, when a user wishes to remove a package and that package requires the removal of other packages. The user can be notified of this requirement and send back a decision of whether or not to remove them. In addition, the application protocol should provide the ability to authenticate users, allowing distributed management.

0.3.2 Modular infrastructure

In order for a package management system to be feasible in an enterprise setting, it must be scalable. Thus a modular design is necessary to facilitate ease of expansion and expansion in unforeseen directions. The current model is a modular design where commands and sub-systems are essentially pluggable modules. These modules can be swapped in and out in real-time, however that functionality is not yet provided in tach. The biggest benefit of this design structure is the scalability afforded. Features can be added and removed at will without

recompiling anything. As well, tach could easily be re-tooled to work with other package management systems by simply replacing the command modules with new ones. This greatly increases the scalability of tach.

In addition, a modular infrastructure allows components of sub-systems to be rewritten without causing a rewrite of the entire programming. Since all functionality, such as logging, caching, and commands, is pushed out to modules, it is easy to rewrite a sub-system without changing the API calls for that module. For example, it would be trivial for tach to use a logging module that connected to a database as long as the current API was adhered to. This is a powerful ability that allows tach to be tailored to meet a variety of needs.

0.3.3 Meta-info caching

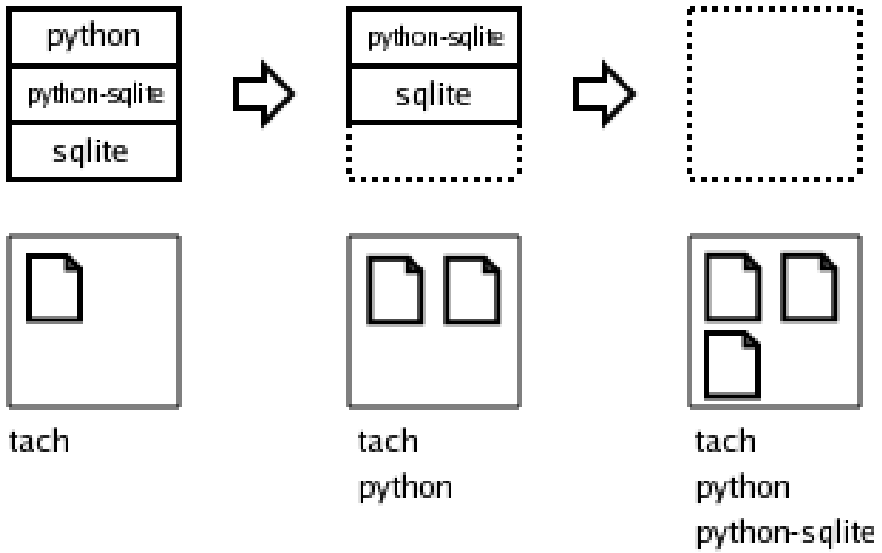
Once XML meta-information is obtained from a service provider it is cached locally to enable quick look-ups of information. If caching was not implemented information would have to be found by re-parsing the XML meta-files. With some of these files having hundreds of thousands of lines, this processes takes a considerable amount of time (about 45 seconds). Thus it only makes sense to parse these files once and cache the information while parsing. Caching is done locally using a SQLite database. SQLite implements SQL in a flat file database. The advantage of SQLite over other SQL databases is that no network daemon is required, thus lowering the collateral requirements to run tach. All package information is cached along with dependency information for each package.

0.3.4 Dependency resolution

One of the most important features of tach is the automatic resolution of dependencies when adding and removing packages. For example, if a user wants to install the `rhn-applet` package, which requires the `up2date-gnome` and `up2date` packages, tach will automatically install all three packages. Initially, the dependency resolution system was built from scratch, using the cache information to resolve all dependencies before consulting the rpm transaction. This approach proved to be problematic since every dependency was identified and linked to its providing package, then this list of packages was checked to see which were installed. This process was time consuming and prone to errors. Instead, this system was replaced with a system that pushes most of the dependency resolution off to the RPM libraries. When unresolved dependencies exist, the local cache is consulted for the packages that provide those dependencies. This method is quicker and simpler, thus less prone to errors.

The basic theory taken with dependency resolution is the notion of a dependency graph. When a package is requested to be installed or deleted it is added to a transaction and the transaction is checked for its dependency resolution status. The return from this check is a list of all unresolved dependencies. This list is then iterated over and for each dependency the cache is queried for the name of the package that provides the unresolved dependency. That package is added to the transaction and once all the dependencies have been iterated

through, the transaction is asked to check dependencies again to account for new dependencies from the added packages. We can assert that each package added to a transaction should cause a change in the dependency graph. If a change does not occur after an iterative check, we can conclude that the dependencies are unresolvable and an exception is thrown. Otherwise, we know that eventually the graph will be reduced completely and all dependencies are met.



0.3.5 Error handling

Error handling is implemented using Python's exception capabilities. This allows tach to handle errors without crashing the daemon, a feature essential to any real world use. Currently there are two classes of exceptions that tach uses; `tachRuntimeError` and `tachRuntimeArgError`. The former being an exception for any unexpected event during network communication or command execution. The latter being an exception thrown when a command receives incorrect arguments.

0.4 Analysis

Tach is currently in beta stage, with only a partial set of the beta release objectives complete. The two largest issues with the current version of tach are the lack of a full featured application protocol and a problem with the way dependencies are cached. Currently, multiple packages may require a dependency, but the logic of the caching system does not allow for multiple packages to provide the same dependency. This is in part due to the inability of the current application protocol to provide this functionality. The problem with the multiple dependency providers occasionally creates issues where a package is not able to

correctly fulfill its dependencies as a result of this limitation. The inter-related nature of these two issues means that their resolution will come at the same time.

0.5 Closing comments

While Linux and, in general, the *NIXes provide advanced features and usability, the general usability can leave something to be desired. Package management in particular is an area where ease of use is needed in order to bridge the usability gap. Tach provides a simple high level interface to the RPM package management system allowing users to take advantage of its benefits without having to learn the overhead. The end result is a package management system that an end-user can use without having very much experience with Linux or RPM. Combined with an eventual graphical interface, tach could become a strong package manager for both enterprise and novice users.

0.6 Glossary

- **dependency** - A dependency is a resource (i.e. file, shared object library, package name) that a package requires or provides. Dependencies must be fully resolved, meaning no dependencies are unmet, in order for a package to be installed by a transaction.
- **header** - A data structure that is part of an RPM file that contains meta-information including a listing of all files contained and the dependency information for the RPM file.
- **manifest** - A manifest is synonymous with **packageinfo**
- **packageinfo** - A package info file is a gzipped XML file that contains XML representation of each package header in a given channel.
- **transaction** - The temporal process responsible for the installation or removal of packages.

Bibliography

[1] http://www.rpm.org/max-rpm/s1-rpm-file-format-rpm-file-format.html

[2] http://torque.sourceforge.net

[3] http://www2.potsdam.edu/kuchyt25/index.cgi?page=projects.rcdapplet