Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

Repository-based System Management Using Conary

Michael K. Johnson, Erik W. Troan, Matthew S. Wilson
Specifix, Inc.
`ols2004@specifixinc.com`

Abstract

There have been few advances in software packaging systems since the creation of dpkg and RPM. Conary is being developed to provide a fresh approach to Open Source Software management and provisioning, one that applies new ideas from distributed software version control tools such as GNU arch and Monotone. Rather than concentrating on package files, Conary provides an architecture built around distributed repositories and change sets, and includes features designed to make branching and tracking Linux distributions simple operations.

The rise of distributions such as Fedora and Gentoo has moved the development of Linux distributions from small, tightly-connected groups to widely-dispersed groups of informal collaborators. These changes have brought to light many shortcomings of the dominant packaging metaphor. By providing version trees distributed across Internet-based software repositories, Conary allows these casual groupings of contributors to work together much more effectively than they can today.

1 Packaging Limitations

Traditional package management systems (such as RPM and dpkg) provided a major improvement over the previous regime of installing from source or binary tar archives. However, they suffer from a few shortcomings, and some of these shortcomings are felt more acutely as the Internet and the Open Source communities have developed and expanded. The authors' experience with the shortcomings of current package management systems strongly motivated Conary's design.

1.1 Branching

Traditional package management systems use simple version numbers to allow the different package versions to be sorted into "older" and "newer" packages, adding concepts such as **epochs** to work around version numbers that do not follow the packaging system's ideas of how they are ordered. While the concepts of "newer" and "older" seem simple, they break down when multiple streams of development are maintained simultaneously using the package model. For example, a single version of a set of sources can yield different binary packages for different versions of a Linux distribution. A simple linear sorting of version numbers cannot represent this situation, as neither of those binary packages is newer than the other; the packages simply apply to different contexts.

1.2 Package Repository Limitations

Traditional package management systems provide no facilities for coordinating work between independent repositories.

- Repositories have version clashes; the same version-release string means different things in different repositories. Repositories can even have name clashes—the same name in two different repositories might not mean the same thing.
- There is no way to identify which distribution, let alone which version of the distribution, a package is intended and built for.

For example, is the `aalib-1.4.0-5.1fc2.fr` package newer than the `aalib-1.4.0-0-fdr.0.8.rc5.2` package? One is from the freshrpms repository, and the other is from the fedora.us repository. Which package should users apply to their systems? Does it depend on which version of which distribution they have? How are the two packages related? Are they related at all?

This is not really a problem in a disconnected world. However, when you install packages from multiple sources, it can be hard to tell how to update them—or even what it means to update a package. You have to rely on your memory of where you fetched a package from in order even to look in the right repository. Once you look there, it is not necessarily obvious which packages are intended for the particular version of the distribution you have installed. Automated tools for fetching packages from multiple repositories have increased the number of independent package repositories over the past few years, making the confusion more and more evident.

The automated tools helped exacerbate this problem (although they did not create it); they have not been able to solve it because the packages do not carry enough information to allow the automated tools to do so.

1.3 Source Disconnected from Binaries

Traditional package management does not closely associate source code with the packages created from it. The binary package may include a hint about a filename to search for to find the source code that was used to build the package, but there is no formal link contained in the packages to the actual code used to build the packages.

Many repositories carry only the most recent versions of packages. Therefore, even if you know which repository you got a package from, you may not be able to access the source for the binary packages you have downloaded because it may have been removed when the repository was upgraded to a new version. (Some tools help ameliorate this problem by offering to download the source code with binaries from repositories that carry the source code in a related directory, but this is only a convention and is limited.)

1.4 Namespace Arbitrary and Unmanaged

Traditional package management does not provide a globally unique mechanism for avoiding package name, version, and release number collisions; all collision-avoidance is done by convention and is generally successful only when the scope is sufficiently limited. Package dependencies (as opposed to file dependencies) suffer from this; they are generally valid only within the closed scope of a single distribution; they generally have no global validity.

It can also be difficult for users to find the right packages for their systems. Both SUSE and Fedora provide RPMs for version 1.2.8 of the iptables utility; if a user found release 101 from

SUSE and thought it was a good idea to apply it to Fedora Core 2, they would quite likely break their systems.

1.5 Build Configuration

Traditional packaging systems have a granular definition of architecture, not reflecting the true variety of architectures available. They try to reduce the possibilities to common cases (`i386`, `i486`, `i586`, `i686`, `x86_64`, etc.) when, in reality, there are many more variables. But to build packages for many combinations means storing a new version of the entire package for every combination built, and then requires the ability to differentiate between the packages and choose the right one. While some conventions have been loosely established in some user communities, most of the time customization has required individual users to rebuild from source code, whether they want to or not.

In addition, most packaging systems build their source code in an inflexible way; it is not easy to keep local modifications to the source code while still tracking changes made to the distribution (Gentoo is the most prominent exception to this rule).

1.6 Fragile Scripts

Traditional package management systems allow the packager to attach arbitrary shell scripts to packages as metadata. These scripts are run in response to package actions such as installation and removal. This approach creates several problems.

- Bugs in scripts are often catastrophic and require complicated workarounds in newer versions of packages. This can arbitrarily limit the ability to revert to old versions of packages.
- Most of the scripts are boilerplate that is copied from package to package. This increases the potential for error, both from faulty transcription (introducing new errors while copying) and from transcription of faults (preserving old errors while copying).
- Triggers (scripts contained in one package but run in response to an action done to a different package) introduce levels of complexity that defy reasonable QA efforts.
- Scripts cannot be customized to handle local system needs.
- Scripts embedded in traditional packages often fail when a package written for one distribution is installed on another distribution.

2 Introduction to Conary

Conary provides a fresh approach to open source software management and provisioning, one that applies new ideas from distributed configuration management tools such as GNU arch and monotone. Rather than concentrating on separate package files as RPM and dpkg do, Conary uses networked repositories containing a structured version hierarchy of all the files and organized sets of files in a distribution.

This new approach gives us exciting new features:

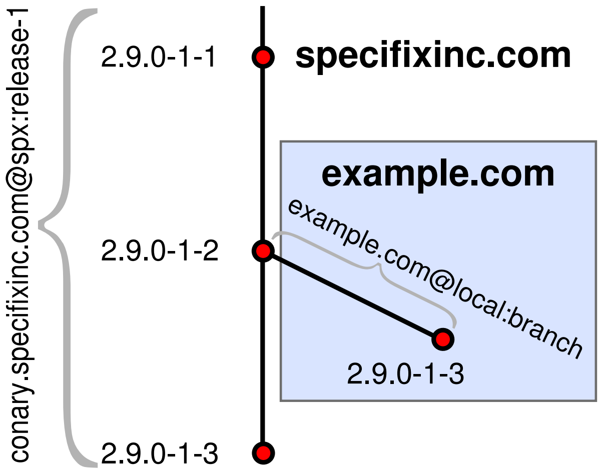
- Conary allows you to maintain and publish changes, both by allowing you to create new branches of development, and by helping track changes to existing branches of development while maintaining local changes.
- Conary intelligently preserves local changes on installed systems. An update

will not blindly obliterate changes that you have made on your local system.

- Conary can duplicate local changes made on one machine, installing those changes systematically on other machines, thereby easing provisioning of large sets of similar or identical systems.

3 Distributed Version Tree

Conary keeps track of versions in a tree structure, much like a source code control system. The difference between Conary and many source code control systems is that Conary does not need all the branches of a tree to be kept in a single place. For example, if Specifix maintains a kernel at `specifixinc.com`, and you, working for `example.com`, want to maintain a branch from that kernel, your branch could be stored on your machines, with the root of that branch connected to the tree stored on Specifix's machines.



3.1 Repository

Conary stores everything in a **distributed repository**, instead of in package files. The repository is a network-accessible database that contains files for multiple packages, and multiple versions of these packages, on multiple development branches. Nothing is ever removed from the repository once it has been added. In simple terms, Conary is like a source control system married to a package system.

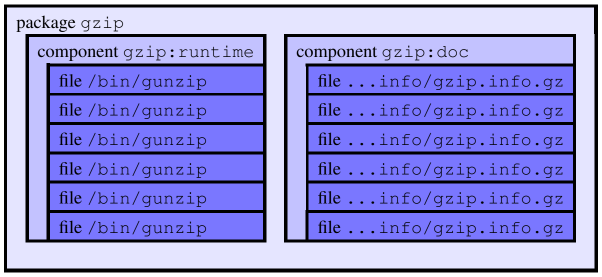
3.2 Files

When Conary stores a file in the repository, it tracks it by a unique file identifier rather than by name. Among other things, this allows Conary to track changes to file names—the file name is merely one piece of metadata associated with the file, just like the ownership, permission, timestamp, and contents. If you think of the repository as a filesystem, the file identifier is like an inode number.

3.3 Troves, Packages, and Components

When you build software with Conary, it collects the files into **components**, and then collects the components into one or more **packages**. Components and packages are both called **troves**. A trove is (generically) a collection of files or other troves.

A package does not directly contain files; a package references components, and the components reference files. Every component's name is constructed from the name of its container package, a `:` character, and a suffix describing the component. Conary has several standard component suffixes: `:source`, `:runtime`, `:devel`, `:docs`, and so forth. Conary automatically assigns files to components during the build process, but you can overrule its assignments and create arbitrary component suffixes as appropriate.


package gzip
component gzip:runtime
file /bin/gunzip
file /bin/gunzip
file /bin/gunzip
file /bin/gunzip
file /bin/gunzip
file /bin/gunzip
component gzip:doc
file ...info/gzip.info.gz
file ...info/gzip.info.gz
file ...info/gzip.info.gz
file ...info/gzip.info.gz
file ...info/gzip.info.gz
file ...info/gzip.info.gz

One component, with the suffix `:source`, holds all source files (archives, patches, and build instructions); the other components hold files to be installed. The `:source` component is not included in any package, since several different packages can be built from the same source component. For example, the `mozilla:source` component builds the packages `mozilla`, `mozilla-mail`, `mozilla-chat`, and so forth. The version structure in Conary's repositories always tells exactly which source component was used to build any other component.

3.4 Labels and Versions

Conary uses strongly descriptive strings to compose the version and branch structure. The amount of description makes them quite long, so Conary hides as much of the string as possible for normal use. Conary version strings act somewhat like domain names, in that for normal use you need only a short portion. For example, the version `/conary.specifixinc.com@spx:trunk/2.2.3-4-2` can usually be referred to and displayed as `2.2.3-4-2`. The entire version string uniquely identifies both the source of a package and its intended context. These longer names are globally unique, preventing any confusion.

Let's dissect the version string `/conary.specifixinc.com@spx:trunk/2.2.3-4-2`. The first part, `conary.specifixinc.com@spx:trunk`, is a **label**. It holds three pieces of information:

- **The repository host name:** `conary.specifixinc.com`
- **Namespace:** `spx` A high-level context specifier that allows branch names to be reused by independent groups. Specifix will maintain a registry of namespace identifiers to prevent conflicts. Use `local` for branches that will never need to be shared with other organizations.
- **Branch name:** `trunk` This is the only portion of the label that is essentially arbitrary; and will be defined by the owner of the namespace it is part of.

The next part, `2.2.3-4-2`, contains the more traditional version information.

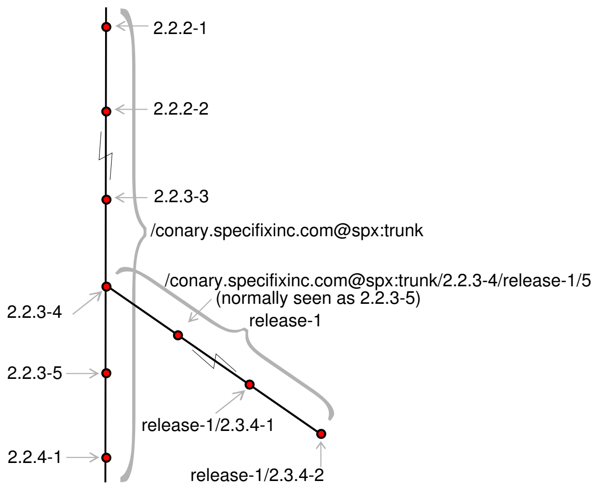
- **Upstream version string:** `2.2.3` This is the version number or string assigned by the upstream maintainer: Conary never interprets this string in any way; the only check it does is whether it is the same or different. It is there primarily to present useful information to the user. Conary never tries to determine whether one upstream version is "newer" or "older" than another. It makes these decisions based on the ordering specified by the repository's version tree.
- **Conary revision:** `4-2` This pair is composed from:
  - **Source build serial number:** `4` Incremented each time a version of the sources with the same upstream version string is checked in. It is similar to the release number used by traditional packaging systems.
  - **Binary build serial number:** `2` How many times this particular source package has been built. This number is not provided for source

packages, because it is meaningless in that context.

Conary describes branch structure by appending version strings, separated by a `/` character. The first step to make a release is to create a branch that specifies what is in the release. Let's create the `release-1` branch off the trunk: `/conary.specifixinc.com@spx:trunk/2.2.3-4/release-1` (note that because we are branching the source, there is no binary build number).

In this branch, `release-1` is a label. The label inherits the repository and namespace of the node it branches from; in this case, the full label is `conary.specifixinc.com@spx:release-1`

The first change that is committed to this branch can be specified in somewhat shortened form as `/conary.specifixinc.com@spx:trunk/2.2.3-4/release-1/5` Because the upstream version is the same as the node from which the branch descends, the upstream version may be omitted, and only the Conary version provided. Users will normally see this version expressed as `2.2.3-5`, so this string, still long even when it has been shortened by elision, will not degrade the user experience.



2.2.2-1
2.2.2-2
2.2.3-3
/conary.specifixinc.com@spx:trunk
/conary.specifixinc.com@spx:trunk/2.2.3-4/release-1/5
(normally seen as 2.2.3-5)
release-1
2.2.3-4
2.2.3-5
release-1/2.3.4-1
2.2.4-1
release-1/2.3.4-2

Labels also have an unusual property: a single label can reference *multiple* branches. To demonstrate why this is useful, let's look at the glib library. Like many other libraries, glib is designed to allow more than one version to be installed on the system at once. Older programs require glib 1.2; newer programs require glib 2. All new releases of glib 1.2 are compatible with programs written and compiled for older versions of glib 1.2; all new releases of glib 2 are compatible with programs written and compiled for older versions of glib 2. They are not, however, compatible with each other; a program compiled for glib 1.2 will certainly not run with glib 2. Therefore, a complete system requires that glib 1.2 and glib 2 both be installed.

Packaging systems often solve this problem by naming the packages differently, putting part of the version number into the name of the package (i.e. `glib` and `glib2`). This works, but it dilutes the revision history that the repository model provides.

By contrast, Conary solves this problem by allowing labels to apply to more than one branch. To see how, we will start by "going back in time" and looking at the version string for glib on the trunk with only glib 1.2 packaged: `/conary.specifixinc.com@spx:trunk/1.2.10-19-3`

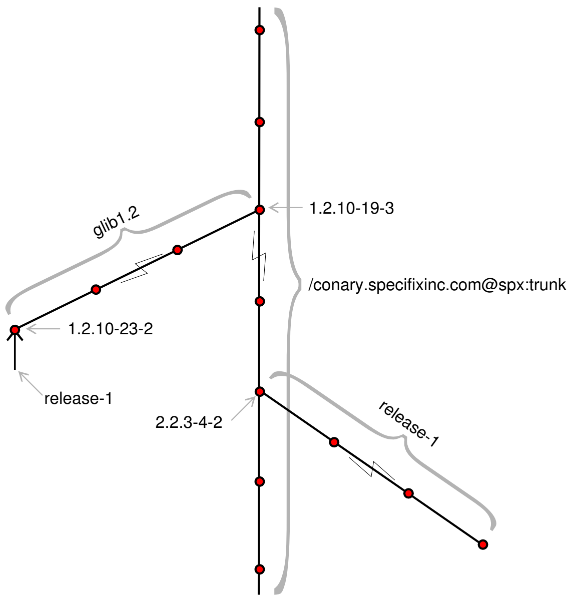
Now, we want to add glib 2 to the repository. We want to have a branch for continuing maintenance of maintain glib 1.2, though, so let's create that first: `/conary.specifixinc.com@spx:trunk/1.2.10-19-3/glib1.2`

Now, we upgrade the trunk to glib 2: `/conary.specifixinc.com@spx:trunk/2.2.3-1-1`

Having maintained both glib 1.2 and glib 2 for a while, we decide that we want to make our first release. We will label every package in the release, including two versions of glib: `/conary.specifixinc.com@spx:trunk/2.2.3-4-2/release-1/4-2` and `/conary.specifixinc.com@spx:trunk/1.2.10-19-3/glib1.2/23-2/release-1/23-2`

The label `conary.specifixinc.com@spx:release-1` now specifies *both* versions of glib. Therefore, if you install `glib conary.specifixinc.com@spx:release-1`, you will get both versions of glib.

Normally, the label to install will be set by installation scripts, and Conary will automatically install both versions of glib. Of course, updates will be applied only when there is a change; an update to glib 1.2 does not affect glib 2. In other words, it "just works" without you having to worry about it.



3.5 Shadows

The most powerful way to manage local changes is (of course) to build changes from source code. Conary makes this possible in two ways. One way is a simple branch, just like you would do with any source code control software. Unfortunately, this is not always the best solution.

Imagine a stock 2.6 Linux kernel packaged in Conary, being maintained on the `/linux26` branch (we have omitted the repository host name and namespace identifier from the label for brevity) of the `kernel:source` package, currently at version `2.6.5-1` (note that because it is a source package, there is no binary build number). You have one patch that you want to add relative to that version, and then you wish to track that maintenance branch, keeping your own change up to date with the maintenance branch, and building new versions as you go.

If you create a new branch from `/linux26/2.6.5-1`, say `/linux26/2.6.5-1/mybranch`, all the work you do is relative to that one version. Creating a new branch does not help you, because the new branch goes off in its own direction from one point in development, rather than tracking changes. Therefore, when the new version `/linux26/2.6.6-1` is committed to the repository, the only way to represent that version in your branch would be to manually compare the changes and apply them all, bring your patch up to date, and commit your changes to your branch. This is time-consuming, and the branch structure does not represent what is really happening in that case.

Conary introduces a new concept: a **shadow**. A shadow acts primarily as a repository for local changes to a tree. A shadow tracks changes relative to a particular upstream version string and source build serial number. Therefore, you cannot change the upstream version of the package—though you can apply any patch you like. (In order to change the upstream version of the package, you would need to create a branch rather than a shadow.) The name of a shadow is the name of the branch with `//shadowname` appended; for example, `/branch//shadow`. The whole branch is shadowed, so if `/branch/1.2.3-3` and `/branch//shadow` exist, then so does `/branch//shadow/1.2.3-3`, regardless of whether `/branch/1.2.3-3` existed at the time the shadow was created. Similarly, if `/branch/1.2.3-3/rel1/1.2.3-3` exists, then so does `/branch//shadow/1.2.3-3/rel1/1.2.3-3`.

Both `/branch/1.2.3-3` and `/branch//shadow/1.2.3-3` refer to exactly the same contents. Changes are represented with a dotted source build serial number, so the first change to `/branch/1.2.3-3` that you check in on the `/branch//shadow` shadow will be called `/branch//shadow/1.2.3-3.1`.

So, to track changes to the `/linux26` branch of the `kernel:source` package, you create the `mypatch` shadow of the `/linux26` branch, `/linux26//mypatch`, and therefore `/linux26//mypatch/2.6.5-1` now exists. Commit a patch to the shadow, and `/linux26//mypatch/2.6.5-1.1` exists. Later, when the `linux26` branch is updated to version `2.6.6-1`, you merely need to update your shadow, modify the patch to apply to the new kernel source code if necessary, and commit the your new changes to the shadow, where they will be named `/linux26//mypatch/2.6.6-1.1`. You can use the shadow branch name `/linux26//mypatch` just like you can use the branch name `/linux26`; you can install that branch, and `conary update` will use the same rules to find the latest version on the shadow that it uses to find the latest version on the branch.

3.6 Flavors

Conary has a unified approach to handling multiple architectures and modified configurations. It has a very fine-grained view of architecture and configuration. Architectures are viewed as an instruction set, including settings for optional capabilities. Configuration is set with system-wide flags. Each separate architecture/configuration combination built is called a **flavor.**

Using flavors, the same source package can be built multiple times with different architecture and configuration settings. For example, it could be built once for `x86` with `i686` and `SSE2` enabled, and once for `x86` with `i686` enabled but `SSE2` disabled. Each of those architecture builds could be done twice, once with `PAM` enabled, and once with `PAM` disabled. All these versions, built from exactly the same sources, are stored together in the repository.

At install time, Conary picks the most appropriate flavor of a component to install for the local machine and configuration (unless you override Conary's choice, of course). Furthermore, if two flavors of a component do not have overlapping files, and both are compatible with the local machine and configuration, both can be installed. For example, library files for the `i386` family are kept in `/lib` and `/usr/lib`, but for `x86_64` they are kept in `/lib64` and `/usr/lib64`, so there is no reason that they should not both be installed, and since the AMD64 platform can run both, it is convenient to have them both installed.

4 Changesets

Just as source code control systems use patch files to describe the differences between two versions of a file, Conary uses **changesets** to describe the differences between versions of troves and files. These changesets include information on how files have changed, as well as how the troves that reference those files have changed.

These changesets are often transient objects; they are created as part of an operation and disappear when that operation has completed. They can also be stored in files, however, which allows them to be distributed like the packages produced by a classical package management system.

Applying changesets rather than installing new versions of packages allows Conary to update only the parts of a package that have changed, rather than blindly reinstalling every file in the package.

Besides saving space and bandwidth, representing updates as changes has another advantage: it allows merging. Conary intelligently merges changes not only to file contents, but also to file metadata such as permissions.

This capability is very useful if you wish to maintain a branch or shadow of a package—for example, keeping current with vendor maintenance of a package, while adding a couple of patches to meet local needs.

Conary also keeps track of local changes in essentially the same way, preserving them. When, for example, you add a few lines to a configuration file on an installed system, and then a new version of a package is released with changes to that configuration file, Conary can merge the two unless there is a direct conflict (unusual but possible). If you change a file's permission bits, those changes will be preserved across upgrades.

Conary supports two types of change sets:

- The differences between two versions in a repository
- The complete contents of a version in a repository (logically, this is the difference between nothing at all and that version)

In the first case, where Conary is calculating the differences between two different versions, the result is a **relative changeset**. In the second case, where Conary is encoding the entire content of the version, the result is an **absolute changeset**. (If you use an absolute changeset to upgrade to the version provided in the absolute changeset, Conary internally converts the changeset to a relative changeset, thereby preserving your local changes.) Absolute changesets are convenient ways of distributing versions of troves and files to users who have various versions of those items already installed on their systems. In practice, they can be distributed just like package files created by traditional package management systems.

Conary can do two things with one of these changesets. It can update a system, either directly from a changeset file, or by asking the repository to provide a changeset and then applying that changeset. It can also store existing changesets in a repository. This capability will be used in the future to provide repository mirroring, and it can also be used to move changes from one repository to a branch in a different repository.

4.1 Representing Local Changes

Conary can also generate a **local changeset** that is a relative changeset showing the difference between the repository and the local system for the version of a trove that is installed. You can distribute a local changeset to another machine in two ways:

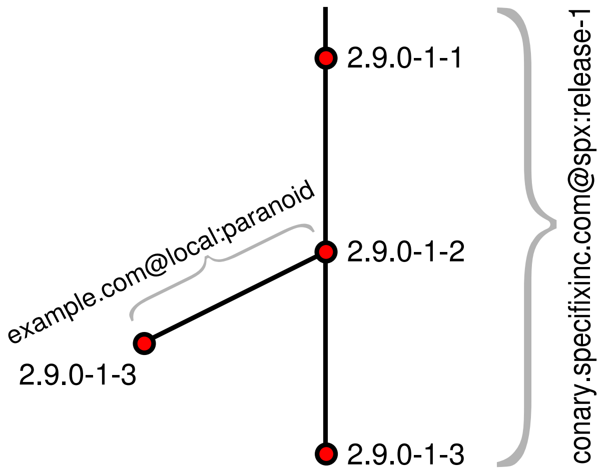
- You can distribute it to other machines with the same version of the trove in question installed.

- You can commit the local changeset to a branch of a repository, and then update to that branch on target machines.

There is an important distinction between the two cases. In the first case, the machine that applies the changeset will act as if those changes had been made by the system's administrator; since those changes are not in a repository they are not versioned. In the second case, however, the machine gets those changes by updating the trove to the branch that contains those changes, and it can continue to track changes from that branch.

For example, assume that you have machines with troves from branches labeled `conary.specifixinc.com@spx:rel1` installed, and you have some local changes that you want to distribute to a group of machines. Let's say that after updating to version `2.9.0-1-2` of `tmpwatch`, you want to change the permissions of the /usr/sbin/tmpwatch binary because you are paranoid: `chmod 100 /usr/sbin/tmpwatch` Now, you record that change in a local changeset; that changeset is relative to `2.9.0-1-2`, and describes your local changes.

You then commit your local changeset to the `conary.example.com@local:paranoid` branch in your local repository. Now, on all the machines in the group, you can update `tmpwatch conary.example.com@local:paranoid`. Each machine will now look in the `conary.example.com` repository on the `paranoid` branch if you simply run `conary update tmpwatch`. This means that if you make further changes to the `tmpwatch` package, you can commit those changes to the `paranoid` branch on the `conary.example.com` repository, and each of the machines will update to the latest version you have committed to that branch. Every time a new version of `tmpwatch` is released on the `conary.specifixinc.com@spx:rel1` branch, you will have to apply the changeset to the `conary.example.com@local:paranoid` branch before the machines with your `paranoid` branch installed will update their copies of `tmpwatch`.


2.9.0-1-1
2.9.0-1-2
2.9.0-1-3
example.com@local:paranoid
2.9.0-1-3
conary.specifixinc.com@spx:release-1

If rather than maintaining a branch, you merely want to distribute some changes that are local to the group of machines, you do not want to commit the local changeset to the repository. Instead, you want to copy the changeset file (let's call it paranoid.ccs) to each machine and run `conary localcommit paranoid.ccs` on each machine. Now, your change to permissions applies to each system, but `conary update tmpwatch` will still look at `conary.specifixinc.com@spx:rel1` and Conary will apply updates to `tmpwatch` from `conary.specifixinc.com@spx:rel1` without additional work required on your part, and it will preserve the change to the permissions of the /usr/sbin/tmpwatch binary on each machine.

Both ways of managing local change are useful. Committing local changesets to a repository is best for systems with entirely centralized management policy, where all system changes must be cleared by some central agency, whereas distributing local changesets is best when individual systems are expected to autonomously update themselves asynchronously.

4.2 Merging

When Conary updates a system, it does not blindly obliterate all changes that have been made on the local system. Instead, it does a three-way merge between the currently installed version of a a file as originally installed, that file on the local system, and the version of the file being installed. If an attribute of the file was not changed on the local system, that attribute's value is set from the new version of the package. Similarly, if the attribute did not change between versions of the package, the attribute from the local system is preserved. The only time conflicts occur is if both the new value and the local value of the attribute have changed; in that case a warning is given and the administrator needs to resolve the conflict.

For configuration files, Conary creates and applies context diffs. This preserves changes using the the widely-understood diff/patch process.

4.3 Efficiency

Conary is more efficient than traditional packaging systems in several ways.

- By utilizing relative changesets whenever possible, Conary uses less bandwidth.
- By modifying only changed files on updates, Conary uses less time to do updates, particularly for large packages with small changes.
- By using a versioned repository, Conary saves space because unchanged files are stored once for the whole repository, instead of once in each version of each package.
- By enabling distributed repositories, Conary
  - saves the time it takes to maintain a modified copy of an entire repository, and
  - saves the space it takes to store complete copies of an entire repository.

4.4 Rollbacks

Because Conary updates systems by applying changesets, and because it is able to follow changes on the local system intrinsically, it easily supports **rollbacks**. If requested, Conary can store an inverse changeset that represents each **transaction** (a set of trove updates that maintains system consistency, including any dependencies) that it commits to the local system. If the update creates or causes problems, the administrator can ask Conary to install the changeset that represents the rollback.

Because rollbacks can affect each other, they are strictly stacked; you can (in effect) go backward through time, but you cannot browse. You have to apply the most recent rollback before you apply the next most recent rollback, and so forth.

This might seem like a great inconvenience, but it is not. Because Conary maintains local changes vigorously, including merging changes to configuration files, and because all the old versions you might have installed before are still in the repositories they came from, you can "update" to older versions of troves and get practically the same effect as rolling back your upgrade from that older version.

Applying rollbacks can be more convenient when you know that you want to roll back the previous few transactions and restore the system to the state it was in, say, two hours ago. However, if you want to be selective, "upgrading" to an older version is actually more convenient than it would be to try to select a rollback transaction that contains the change you have in mind.

5 Other Concepts

5.1 Dynamic Tags

In place of the fragile script metadata provided by traditional package management systems, Conary introduces a concept called **dynamic tags**. Files managed by Conary can have sets of arbitrary text tags that describe them. Some of these tags are defined by Conary (for example, `shlib` is reserved to describe shared library files that cause Conary to update /etc/ld.so.conf and run `ldconfig`), and others can be more arbitrary. (In order to allow tag semantics to be shared between repositories, it is likely that Specifix will host a global tag registry in the future.)

By convention, a tag is a noun or noun phrase describing the file; it is not a description of what to do to the file. That is, *file* is-a *tag*. For example, a shared library is tagged as `shlib` instead of as `ldconfig`. Similarly, an info file is tagged as `info-file`, not as `install-info`.

Conary can be explicitly directed to apply a tag to a file, and it can also automatically apply tags to files based on a **tag description** file. A tag description file provides the name of the tag, a set of regular expressions that determine which files the tag applies to, the path of the **tag handler** program that Conary runs to process changes involving tagged files, and a list of actions that the handler cares about. Conary then calls the handler at appropriate times to handle the changes involving the tagged files.

Actions include changes involving either the tagged files or the tag handlers. Conary will pass in lists of affected files whenever it makes sense, and will coalesce actions rather than running all possible actions once for every file or component installed.

The current list of possible actions is:

- Tagged files have been installed or updated; Conary provides a list of all installed or updated tagged files.
- Tagged files are going to be removed; Conary provides a list of all tagged files to be removed.
- Tagged files have been removed; Conary provides a list of filenames that were removed.
- The tag handler itself has been installed or updated; Conary provides a list of all tagged files already installed on the system.
- The tag handler itself will be removed; Conary provides a list of all the tagged files already installed on the system to facilitate cleanup.

Because the tag description files list the actions they handle, the tag handler API can be expanded easily while maintaining backward compatibility with old handlers.

Avoiding duplication between packages by writing scripts once instead of many times avoids bugs in scripts. Practically speaking, it avoids whole classes of common bugs that cause package upgrades to break installed software, and even more importantly from a provisioning standpoint, bugs that would cause rollbacks to fail. It makes it much easier to fix bugs when they do occur, without any need for "trigger" scripts that are often needed to work around script bugs in traditional package management. It also allows components to be installed across distributions—as long as they agree on the semantics for the tags, the actions taken for any particular tag will be correct for the distribution on which the package is being installed.

Calling tag handlers when they have been updated makes recovery from bugs in older versions of tag handlers relatively benign; Conary needs to install only a single new tag handler with the capability to recover from the effects of the bug. Older versions of packages with tagged files will use the new, fixed tag handler, which allows you to revert those packages to older versions as desired, without fear of re-introducing bugs created by old versions of scripts.

Furthermore, storing the scripts as files in the filesystem instead of as metadata in a package database means:

- they can be modified to suit local system peculiarities, and those modifications will be tracked just like other configuration file modifications;
- they are easier for system administrators to inspect; and
- they are more readily available for system administrators to use for custom tasks.

5.2 Groups and Filesets

There are two other kinds of troves that we did not discuss when we introduced the trove concept: groups and filesets.

Filesets are troves that contain only files, but those files come from components in the repository. They allow custom re-arrangements of any set of files in the repository. (They have no analog at all in the classical package model.) Each fileset's name is prefixed with `fileset-`, and that prefix is reserved for filesets only.

Filesets are useful primarily for creating small embedded systems. With traditional packaging systems, you are essentially limited to installing a system, then creating an archive containing only the files you want; this limits the options for upgrading the system. With Conary, you can instead create a fileset that references the files, and you can update that fileset whenever the components on which it is based are updated, and use Conary to update even very thin embedded images.

The desire to be able to create working filesets was a large motive for using file-specific metadata instead of trove-specific metadata wherever possible. For example, files in filesets maintain their tags, which means that exactly the right actions will be taken for the fileset. If Conary had package scripts like traditional package managers, it would be impossible to automatically determine which parts (if any) of the script should be included in the fileset. (As already discussed, scripts have other problems that tags solve; this is just another one of the architectural reasons that tags are preferable to scripts.)

Groups are troves that contain any other kind of trove, and the troves are found in the repository. (The task lists used by apt are similar to groups, as are the components used by anaconda, the Red Hat installation program.) Each group's name is prefixed with `group-`, and that prefix is reserved for groups only.

Groups are useful for any situation in which you want to create a group of components that should be versioned and managed together. Groups are versioned like any trove, including packages and components. Also, a group references only specific versions of troves. Therefore, if you install a precise version of a group, you know exactly which versions of the included components are installed; if you update a group, you know exactly which versions of the included components have been updated.

If you have a group installed and you then erase a component of the group without changing the group itself, the local changeset for the group will show the removal of that component from the group. This makes groups a powerful mechanism administrators can use to easily browse the state of installed systems.

The relationship between all four kinds of troves is illustrated as follows:

Troves		built from	
		source	repository
contain	files	component	fileset
	troves	package*	group

*packages contain only components

Groups and filesets are built from `:source` components just like packages. The contents of a group or fileset is specified as plain text in a source file; then the group or fileset is built just like a package.

This means that groups and filesets can be branched and shadowed just like packages can. So if you have a local branch with only one modified package on it, and then you want to create a branch of the whole distribution containing your package, you can branch the group that represents the whole distribution, changing only one line to point to your locally changed file. You do not have to have a full local branch of any of the other packages or components.

Furthermore, when the distribution from which you have branched is updated, your modification to the group can easily follow the updates, so you can keep your distribution in sync without having to copy all the packages and components.

6 Further Work

An alpha release of Conary is now available from `http://www.specifixinc.com`, along with a Linux distribution built with Conary. While these releases allow users and developers to begin making use of Conary's features, there is significant work remaining.

The shadow design discussed in this paper has not yet been implemented.

Conary does not yet resolve dependencies. Although some dependency information is already generated and tracked on a per-file basis, no effort is made to ensure that those dependencies are resolved when components are installed.

As Conary and Conary-based distributions become more popular, there will be a need for both repository caches and repository mirrors. While some preliminary design work has been done for each of these, no implementation work has begun.

The implementation of flavors is preliminary, especially in regards to configuration settings. While limited testing has been done with troves built for varying architectures and Specifix's build scripts implement some configuration settings, Conary does not yet properly select the flavor to install on a system.

Conclusion

Conary was designed to address many of the limitations of the traditional packaging metaphor. The enormous growth in the Linux developer base over the past decade has shown that packaging systems do not scale well to multiple repositories with conflicting content, and can make it difficult for large numbers of developers to coordinate package releases.

Conary provides flexible branching, which enables it to find both binaries and sources anywhere on the Internet, and allows the local administrator to preserve local changes and create local development branches of those packages. By providing a name space separator as part of the branch names, Conary allows many groups to use the same tool while building a single distributed version tree, without any formal collaboration between the groups.

Innovations such as shadows and versioning groups of packages and files (allowing those container objects themselves to be branched and shadowed) significantly reduce the difficulty of maintaining customized Linux distributions. Instead of being forced to accept complete responsibility for all aspects of the distribution, developers can now concentrate on maintaining just their changes. Those changes are represented in a concise way that can track upstream changes to the entire distribution.

Conary is designed to enable a loosely-coupled, Internet-based collaborative approach to building Linux distributions. By making branching and shadowing inexpensive operations that can change almost any aspect of a Linux system, we hope members of the Linux community will be able to build the Linux distribution they want, rather than use one that is merely close enough.