Introduction to Conary

Conary: Git for system management

Conary is a system software provisioning and management tool that brings concepts from distributed source code control systems such as Git and Mercurial to system management. Conary provides differential update, rollback, configuration management, staging/promotion, entitlement, replication, dependency management, introspection, attribution/lineage, repeatable build, and layered platform/system definition. Unlike most package-based software management tools that depend on archive files as their primary mechanism of distribution, Conary provides networked repositories containing structured version hierarchies of all the files and organized sets of files in software products.

Conary has three main components: software repository, system management, and software build. The system management component manages the state of an individual system (based on the contents of a Conary repository), the build component automates building software and collections of software into a Conary repository, and the Conary repository is a web application that stores versions of software and collections of software. This paper introduces the repository and system management components of Conary.

Conary models the intended state of a system, such that it can recreate the same state systematically on other machines, enabling precise staging ("dev/test/prod") of changes through a deployment process, and easing provisioning of large sets of similar or identical systems. A model can be maintained on a target system or packaged into a Conary repository. Conary also intelligently preserves intentional local changes on installed systems, such that an update will not blindly obliterate local changes such as changes to configuration files.

Conary is also the core technology of a family of tools that further automate the software build and management process.

1. Repository

Conary stores all software and collections of software in **repositories**, instead of in package files. A Conary 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 a repository once it has been added. In simple terms, Conary is like a source control system married to a software management system.

1.1 Software Version Management

Conary keeps track of software versions in a tree structure, much like a source code control system. (Technically, it is a directed acyclic graph of versions.) 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 repository; instead, branches can be **distributed** between multiple repositories. Like Git, one repository can contain a branch from another repository; unlike Git, those branches are maintained by reference rather than as a copy. Conary's collections of software can and normally do reference software in multiple repositories.

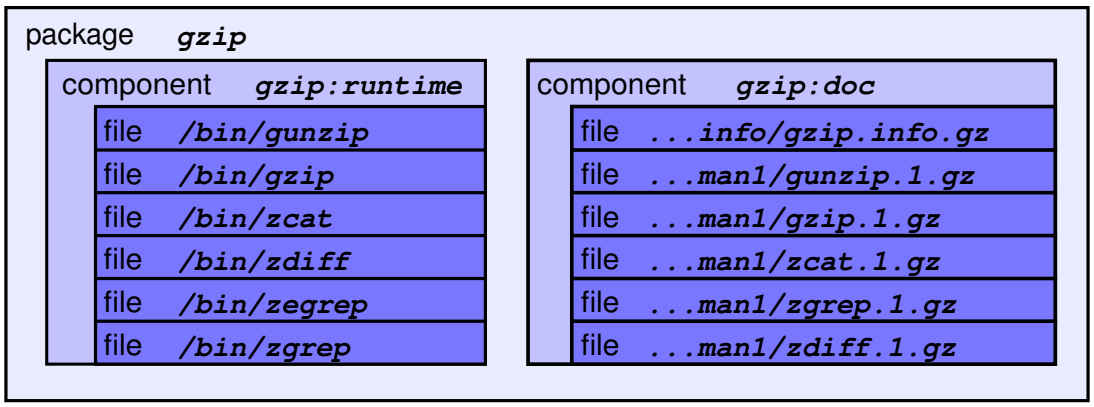
1.2 Troves, Packages, and Components

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 generically discribes any collection of files or other troves in Conary.

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

For example, when the gzip package is built, several components are built, including `gzip:runtime` and `gzip:doc`. The following diagram shows their relationships.



One unique 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, but Conary keeps a reference to exactly which source component was used to build any binary component.

1.3 Groups

Groups are collections of software components of any kind (any kind of Conary “trove”). In most software management systems, lists of component names are bound to specific versions at the time an action is invoked on an end system. Conary instead considers binding names to versions as a compilation task comparable to compiling product source code to binaries. Like packages, Conary groups are built from `:source` components containing text. This text is compiled to produce the version bindings as a build action.

Groups are normally nested to represent the normal relationships of software components. Nested groups can be built from a single definition at once, or separately. For example, an operating system platform could be built as a single large group containing sub-groups representing major functionality divisions such as “web server” or “desktop environment” or “developer workstation”, all from a single definition of the entire group structure, built in a single build action. A product could then be built from different sources that merely reference the relevant parts of the operating system platform by the names of the relevant groups, and also includes the product software, built and managed independently of the operating system platform.

Systems are managed primarily in terms of groups that define sets of software components that work together.

Names prefixed with `group-` are reserved and required for groups.

1.4 Other reserved trove names

Factories are build automation components that can introspect sources to build packages and groups. Names prefixed with `factory-` are reserved and required for factories.

Filesets are troves that contain only files, but those files come from other components already in the repository. Names prefixed with `fileset-` are reserved and required for filesets. Filesets are very rarely used because they are not useful enough to be worth the work to create them; they were originally intended to be useful for small embedded systems where they could have been.

1.5 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 by default hides as much of the string as possible for normal use. Like domain names, in normal use you need only a short portion of a Conary version string. For example, the version `/conary.example.com@plat:1/2.2.3-4-2` can usually be referred to and displayed as `2.2.3-4-2`. The entire version string is globally unique, and identifies both the source of a package and its intended context.

Let's dissect the version string **`/conary.example.com@plat:1/2.2.3-4-2`**.

The first part, **`conary.example.com@plat:1`**, is a **label**. The label holds:

- **The repository host name: `conary.example.com`** is generally a fully-qualified domain name that can be resolved.
- **Branch name: `plat:1`** is a context identifier made up of a **namespace** and a **tag**:
    - **Namespace: `plat`** A high-level context specifier that by convention communicates relationship between otherwise independent branch names. (The **`local`** label is reserved for internal Conary purposes.)
    - **Tag: `1`** 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. It usually represents a major version and usually product stage; related tags might be **`1-devel`** and **`1-qa`** representing the development and QA stages of product development, respectively.

The next part, **`2.2.3-4-2`**, is called the **revision** and contains the more traditional version information.

- **Upstream version string: `2.2.3`** This is the version (number or string) of the source code, whatever its origin. Conary merely checks whether this upstream version exists already (to see which source count to use; see below), that it starts with a numeric character (to distinguish versions from labels when abbreviating versions), and that the `-` character is not in it (because the `-` character separates the upstream version string from the next data element). The upstream version string 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. Instead, the ordering specified by the repository's version tree determines what Conary thinks is older or newer; the most recent commit to the label is the newest.
- **Source count:** `4` Incremented each time a new revision of the source component containing the same upstream version string is checked in to the Conary repository. It is similar to the release

number used by traditional packaging systems.

- **Build count:** 2 How many times the source component that this component comes from has been built into a corresponding binary package. Source components have no build count because they are the input to the build process.

1.6 Shadows

Conary introduced the concept of a branch that tracks rather than diverges from its parent as a **shadow**. This is similar to the Git concept of a "tracking branch", where the developer can choose to merge changes into the shadow (tracking branch) as well as to push changes from the shadow (tracking branch) to the parent. Unlike Git, Conary stores the relationship between the parent branch and the shadow in the version of the shadow. The name of a shadow is the name of the parent with `//shadowname` appended; for example, `/parent//shadow`. (Of course, `/parent` will really be something like `/conary.example.com@plat:1` and `//shadow` will really be something like `//child.example.com@example:myshadow`)

Both `/parent/1.2.3-3` and `/parent//shadow/1.2.3-3` refer to exactly the same contents. Changes are represented with a "dotted" source count, so the first change to `/parent/1.2.3-3` that you check in on the `/parent//shadow` shadow will be called `/parent//shadow/1.2.3-3.1`. When you build binaries, you will have versions like `/parent//shadow/1.2.3-3.1-1.1` where the build count has also been "dotted".

If you update to a new upstream source version on your shadow without merging that version from the parent, `0` is used as a placeholder for the parent source count, since the source count is 1-based and is never `0` otherwise. Therefore, if you check in version 1.2.4 on this shadow, you will get `/parent//shadow/1.2.4-0.1` as your version. The same thing happens for build count; if the source version `/parent/1.2.4-1` exists, but the build version `/parent/1.2.4-1-1` does not exist when you build on your shadow, you will get versions that look like `/parent//shadow/1.2.4-1.1-0.1`

1.7 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, and per-package flags for configuration that are very package-specific. A **flavor** is an architecture/configuration combination.

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 for authentication, and once with pam disabled. All these versions, built from exactly the same sources, are stored together in the repository, and given the same version string.

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 x86_64 platform can run both, it is convenient to

have them both installed.

2. System Management

2.1 System Model

To manage a system, Conary stores a file called a **system model** that is a plain text description of the desired state of the system. It looks like a log of operations, but it isn't: it is a description of operations that could be used to get to the desired state. Packages installed only to satisfy dependencies are not included in that description, and Conary updates to newer versions by changing the versions in the model, processing the model to describe the newly-desired state of the system, and then making whatever changes are needed to synchronize the system to that newly-desired state.

2.2 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.

Changesets are transient objects; they are created as part of an operation and disappear when that operation has completed. Copies of changesets can be stored in files for inspection.

Applying changesets rather than installing new versions of packages allows Conary to update only the parts of a package that have changed, apply changes to configuration files, and even respect file metadata changes such as permissions.

This capability is very useful if you wish to maintain a 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. For example, when 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 except in the rare case of conflict. If you change a file's permission bits, those changes will be preserved across upgrades.

Conary supports two types of change sets:

- A **relative changeset** expresses the differences between two versions in a repository
- An **absolute changeset** expresses the complete contents of a version in a repository (logically, this is the difference between nothing at all and that version). 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.

2.3 Merging Changes

When Conary updates a system, it does not blindly obliterate all changes that have been made on the local system. Instead, it merges attribute changes between the currently installed version of a file as originally installed, that file as it exists on the local system, and the new 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. Conflicts occur only if both the new value

and the local value of the attribute have changed; in that case a warning is given and the administrator must resolve the conflict.

For **configuration** files, Conary creates and applies context diffs. This preserves changes using the the widely-understood diff/patch process. For other files, Conary supports different content merge behavior based on the file type: **transient** files unconditionally replace file content on the system with content from the repository; **initial contents** files unconditionally retain content from the system, and other file types by default abort update operations on conflict.

2.4 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**. Conary stores 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, Conary can apply the changeset that represents the rollback.

Rollbacks are meant only for precisely undoing Conary operations, not for version downgrades. Rollback operations are strictly stacked; you can (in effect) go backward through time, but you cannot browse and selectively revert parts of history. You have to apply the most recent rollback before you apply the next most recent rollback, and so forth. Applying a rollback cannot itself be rolled back. 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 downgrade by “updating” to older versions of troves. This is similar to rolling back your upgrade from that older version, except that the operation of downgrading can itself be rolled back.

2.5 Dynamic Tags

Traditional software package management systems allow the packager to attach arbitrary scripts or programs 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 can be catastrophic and require complicated workarounds in newer versions of packages. This can arbitrarily limit the ability to revert to old versions of packages.
- Scripts are often 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).
- Some systems have triggers (scripts contained in one package but run in response to an action done to a different package), which introduce combinatoric complexity that defies reasonable QA efforts.
- Scripts cannot be customized to handle local system needs.
- Scripts embedded in traditional packages often fail when a software unit is installed in a context that was not explicitly tested for.

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 on Linux causes Conary to update the /etc/ld.so.conf file if necessary, and run the “`ldconfig`” command), and others can be arbitrary.

Tag names are intended to be shared between repositories and distributions as much as is reasonably possible. When tag names are shared, tags will not introduce arbitrary incompatibilities in packaging. If one distribution needs something special done for any particular type of file, it should modify or replace the tag handler for that tag, but should leave the tag name the same.

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 or tag description have been installed or updated; Conary provides a list of all tagged files already installed on the system.
- The tag handler or tag description 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.

Writing scripts once instead of many times avoids bugs in scripts. 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 have often been needed to work around script bugs in traditional package management. It also allows components to be installed across software 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.

Conary also provides **trove scripts** that are attached to specific versions of troves and function essentially like the scripts in traditional package management systems. They are usually added to groups, particularly to implement workarounds for bugs. However, they have all the downsides of their analogs in traditional package management systems, in particular making rollbacks less reliable.

3. Efficiency

Conary has several efficiency benefits over traditional software package management systems.

- 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 based on content-addressable storage, 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.

Conary stores several orders of magnitude more dependency information about files and components than traditional software package management systems, in order to more confidently and completely ensure that all the dependencies in a system are resolved. This means it uses significantly more memory and processor time than traditional software package management to calculate dependency closure for system installation and update operations.

Conclusion

Conary was originally designed to address many of the limitations of the traditional packaging metaphor. The enormous growth in the Linux developer base over the past two decades demonstrated 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. Beyond Linux, growing software complexity has exposed weaknesses in classic software management paradigms.

Conary brings the concepts behind “fork me on github” for distributed development to software system management.

Copyright © SAS Institute Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.