The Future of Continuous Integration in GNOME

Colin Walters
Red Hat, MA, USA
GNOME Project
walters@verbum.org

Germán Poo-Caamaño
University of Victoria, Canada
GNOME Project
gpoo@gnome.org

Daniel M. German
University of Victoria, Canada
dmg@uvic.ca

***Abstract*—In Free and Open Source Software (FOSS) projects based on Linux systems, the users usually install the software from distributions. The distributions act as intermediaries between software developers and users. Distributors collect the source code of the different projects and package them, ready to be installed by the users. Packages seems to work well for managing and distributing stable major and minor releases. It presents, however, various release management challenges for developers of projects with multiples dependencies not always available in the stable version of their systems. In projects like GNOME, composed of dozens of individual components, developers must build newer versions of the libraries and applications that their applications depend upon before working in their own projects. This process can be cumbersome for developers who are not programmers, such as user interaction designers or technical writers. In this paper we describe some of the problems that the current distribution model presents to do continuous integration, testing and deployment for developers in GNOME, and present ongoing work intended to address these problems that uses a git-like approach to the building and deployment of applications.**

***Index Terms*—Release Engineering, Continuous Integration, Free/Open Source Software, GNOME**

I. Introduction

GNOME[1] is a project whose mission is to build a free desktop. It is composed of several components or modules interrelated that are developed by different groups of people, both volunteers and paid developers. One of the challenges of building a desktop is to integrate and test the whole system, especially the fundamental pieces necessary to support it. To improve the user experience it is sometimes necessary to make changes at different levels of the application stack. Some parts of the stack are internal (such as libraries, window manager, user applications, etc.) and some others are external (for instance, 3D acceleration in drivers, 3D rendering by software, hardware presentation to user space, underlying intercommunication process, etc.). Changes that involve different levels of this stack require careful testing before releasing them to the end-users. This process can be cumbersome for new contributors, who need to deal with building the system **before** they can start contributing to a module of the project. In addition, contributors can be testers, documenters, user interaction designers or programmers; therefore, they might not have the skills to build software. While contributors need the latest version of libraries or applications for their development, they also need a stable system to perform other work.

[1]http://gnome.org

In [1], Michlmayr et al. revealed three different types of release management that occur in FOSS projects: development releases, major end-user stable releases, and minor releases (that update existing end-user releases). The major focus of release management in FOSS

has been in the last two, that in Linux systems is mainly done by distributors, leaving the first one up to the developers, who are considered experts.

Distributions, such as Ubuntu and SUSE, use packages to deliver software to their users. Unfortunately, the package model has restrictions for development. Packages are not created by distributions at the same pace as a software is being developed. This means that contributors cannot rely on their distributions to update the packages they are testing. They need to build the newer releases of packages that their system requires in order to have a system where they can develop and run the system being developed.

Even if packages could be created at the same pace of the software development, they have dependencies that might need to be updated and tested. Because it is software under development, eventually they could break a system and leave it unusable. Using the current packaging systems of distributions, it is not easy to roll back to a previous state. This situation is risky for contributors who only have one computer that they need to use for development and other work and cannot afford to have an unstable environment.

Another problem with the package model used by Linux distributions is the lack of a clear separation between the basic system and the applications. Every piece of software is a package (kernel, libraries, frameworks, end-user applications). Upgrading a package for testing might trigger a chain of undesirable upgrades, even if they are not strictly necessary. For example, a package that is being tested might declare that it requires a new version of another one (even if the older one is sufficient) and might trigger an upgrade to an unstable package, even though it is not needed. In Debian is possible to tag a package as *essential* to indicate that a package must be available to have an usable system [2], but there is no **safe** mechanism to "pin" a version so it is never upgraded (or downgraded). Hence a new version of the package might leave the system in an unusable state.

In this paper we address the consequences of the package model in the context of release management on projects like GNOME and we present OSTree, a continuous integration

978-1-4673-6441-6/13 © 2013 IEEE

RELENG 2013, San Francisco, CA, USA

system for GNOME that addresses these problems. We are interested in the following questions:

- How can be reduced the gap between the introduction and detection of regressions?
- How can the release management process be improved to allow any contributor to try the latest versions available without breaking their system, and allow her to easily switching between a testing and a development environment?

In section II, we describe two of the existing problems with the package model of distributing software in FOSS projects. In section III, we introduce the proposed solution that builds a system, keeps track of the binaries built after each commit and deploys systems for testing on a daily basis. In section IV, we enumerate the technical and political limitations of the OSTree approach. Section V provides an overview of the related work. The final section discusses future work and conclusions.

II. Background

The package model presents some problems for developers of an environment like GNOME, although it works well for distributors. GNOME follows a release schedule of 6 months for major releases. This allows distributors to pick a stable version of GNOME in a predictable timetable in order to integrate it into their systems, which are targeted to end-users. For projects like GNOME, the following problems arise as the projects get more complex: the gap between the introduction and detection of regressions grows, and using packages for testing a project becomes challenging. Each problem is described further below.

A. Gap Between Introduction and Detection of Regressions

Figure 1 illustrates a simplified timeline of a regression from its introduction to the moment that it is detected, after it has reached distribution's end-users. The developer perceives the system as changes in the version control repository. The packagers see the changes in the system as releases ("tarballs"[2]). The users see it as a new package that they install. It is then, when a user might notice a regression and report it back to the developers. This might happen weeks or months after the introduction of the regression.

In Figure 1, *users* include end-users as well as external developers that use the software. There might be some GNOME developers in that group, too. For instance, volunteers who maintain their projects with older versions of the libraries, so that they avoid to build a newer version of the stack because it could be time consuming.

Regression testing might not catch every potential situation and many defects might only manifest in real settings.

Most users of GNOME install it using the packages provided by their distributors. Therefore, one of the challenges to detect and fix regressions faster is to reduce the time between bug-introducing changes and the creation and distribution of packages.

[2]A tarball refers to a tar file, which is a collection of files and directories packed together preserving permissions and ownership of the files. This is the common way that FOSS projects release their software.

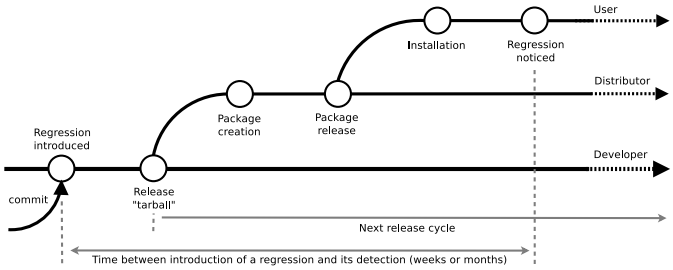

Figure 1. Time Between Introduction of a Regression and Its Detection

A solution to this problem is to ask users to download, build and install the system from tarballs, bypassing the distributions. However, this is not a practical solution. For many contributors building everything they need (including the different components of GNOME and any necessary dependencies) might be beyond their skills, and perhaps more critical, building the entire stack creates multiple potential points of failure. If one of the libraries does not install, or it is faulty, then the entire system might stop working. This is an undesirable outcome as it is explained in the next section.

B. Rolling Back to a Previous Working Version

Allowing multiple versions of a program, and to switch back and forth between them, would help developers to test continuously the latest development version of a program and still have an usable working environment. In GNOME, as in many FOSS projects, the developers are distributed in different locations. Most of them only have one computer that they use to contribute to the project and do any other, non-related work. Allowing users to install and switch between different versions of packages would be a way to reduce the risk of testing. Contributor might install the latest development packages they need; if a critical error is found, then they could easily switch back to the stable versions, without compromising the stability of their systems.

Packaging systems used by distributions like `rpm`, `dpkg` and `portage` can only install one version of a package at a time. The upgrade of a package is done by overwriting the old version with the newer one, and when there are dependencies among packages, they might have to be upgraded too. This process is not atomic: during the upgrade, some files belonging to the old version might conflict with the new one, leaving the system in an inconsistent state [3]. In addition, packaging systems rely on the version numbers assigned to packages, whose numbers must strictly increase over time (larger numbers imply newer releases). For official stable releases this system has proven to work well, mainly because packages, and their interactions, are well tested *before* they are deployed to users.

Unfortunately, this process was not created for the distribution of packages that are expected to be tested by users. If one of the packages breaks the system, it might not be trivial to

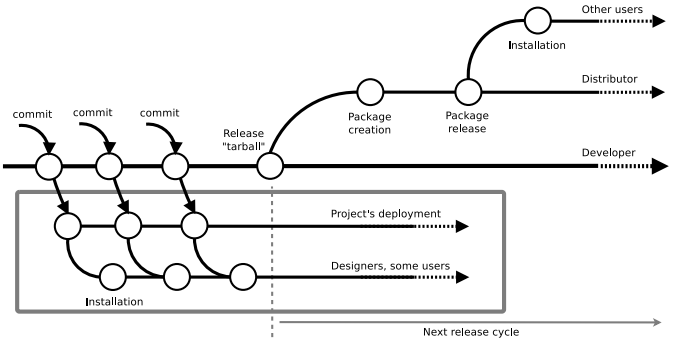

Figure 2. A Continuous Integration System for GNOME

rollback installations to return the system to a stable, reliable state.

III. OSTree as a Continuous Integration System

OSTree[3] is a tool for developing, building, and deploying Linux-based systems that aims to a release model as shown in figure 2. It sets aside the concept of package previously discussed. After a commit in a component being developed, the application and its dependencies are built and deployed in the entire project's deployment repository. From there, developers and experienced users can update theirs systems to run and test the current state of the whole project.

OSTree is separated into 3 conceptually independent parts:

1) Version control system designed for binaries;
2) Build system for applications; and
3) Deployment system for developing and building Linux-based operating systems.

OSTree relies on an external tool (Yocto [4]) to build the core that conforms the operating system. Basic programs and libraries are added to provide the minimal system necessary to build the application sets. This approach attempts to separate different type of software (core system and applications) in contrast to the package model found in Linux-based systems.

A. Version Control System for Binaries

OSTree replaces the package–centric installation model with a `git`-like version controlled repository of the entire filesystem tree, which can be cloned and updated on the client machine. A repository stores multiple and complete file system trees already built. As a consequence, the operations of installing and upgrading the system are equivalent to version control operations: installation is done by "cloning" the repository, and updates by "pulling" new changes from it. In addition, OSTree can retain multiple named versions of the tree and easily switch between any of them. That allows developers to maintain experimental builds, rollback to earlier versions to reproduce bugs, and ensure that the entire development team can experience the identical set of components.

[3]http://git.gnome.org/browse/ostree

The core of OSTree architecture is inspired by `git` [5]. As in `git`, OSTree has pack files, tree and commit objects. The underlying principle of OSTree is that the operating system (with basic programs and libraries) is read–only, that makes it possible parallel operating systems to be installed in a directory, sharing space by not copying the files that are identical among versions. Therefore, it is possible to be running GNOME from Debian stable and then download the latest version of GNOME running on latest Ubuntu into a folder and try it. Thus, the primary architectural difference from `git` is the focus on read–only checkouts.

Unlike virtualization, there is no extra consumption of resources and the user has access to all of her data as before (email, documents, etc.).

Finally, to avoid potential sources of conflict, each system counts with its own directory to write system data (logs, cache, locks, queues, etc.) during its operation. Therefore, these directories are not stored in the repository.

B. Build System Tool — ostbuild

`ostbuild` is a build tool that monitors the version control repositories of the software. Whenever it detects a change (commit), it builds a binary. This binary is tagged with the commit, and is stored in the OSTree repository. It only rebuilds changed components. OSTree makes possible for developers to track down exactly which commit caused a regression. The repositories to monitor are defined in a file that contains the definitions and options to build every application.

OSTree makes reverting regressions simple: a new commit that fixes a regression can be deployed from the repository. In the meantime, users can boot into a pre–regression operating system until a fix is available.

C. Operating System Management Tool — ostree

`ostree` is used to interact with the repository (in a manner similar to `git`). It has some features found in package distribution systems. For example, it provides triggers which are the analogous to the *postinst* scripts available in `rpm`. This feature allows `ostree` to run scripts after a checkout. The utilities can be necessary to update links and cache of shared libraries, icons, among others (similar to the way scripts are run after the installation of packages). In addition, `ostree` integrates a checkout into the boot process of a Linux system. This makes it possible to have multiple operating system versions installed, each of them booting in a restricted environment, only sharing the user data.

Table I lists its basic commands, which are categorized in regular and administrative commands. The former groups commands similar to `git`, whereas the later groups commands to deploy a system and require superuser privileges.

IV. Limitations

OSTree is currently functional, although still under development. It is being used to monitor and build over 200 `git` repositories, and is being used by some GNOME contributors to follow the development.

Table I
BASIC COMMANDS OF THE OSTREE TOOL

Regular Commands	
init	Initialize a new empty repository
pull	Pull a remote content into the local repository
checkout	Check out a commit into a file system tree
build	Build components, put the results in an ostree branch
resolve	Get the source code in and take a system's snapshot
ls	Print the contents of directories
cat	Show the content of a given build
fsck	Check the consistency of a repository
Administrative Commands (Preceded by *admin*)	
init	Initialize a /ostree directory
deploy	Check out a revision from the local ostree repository and set up the boot loader
update-kernel	Update the current kernel and the boot loader

Some of the limitations include, first, the time required to build software is variable. Some projects can be built in minutes after a change in the source code. However, projects like WebKit can take several hours, making the continuous integration slower. Second, the developers of a project using OSTree should take care of the security updates in any part of the operating system, not only in the software they are developing. Even though security updates can be done via merging new commits in a tree, they could demand more human resources. Third, the user data is shared and can be migrated from one older format to a newer one after an upgrade. Although this is a rare issue, it can cause problems to rollback to a previous version. Fourth, a similar problem can cause the triggers performed after a checkout. However, they can be disabled if needed.

V. RELATED WORK

`dpkg` and `rpm`, used by Debian and Red Hat based distributions respectively, are used for building and deploying the most popular Linux-based systems (as Debian, Ubuntu, Fedora, OpenSuSE, etc.). These have no support for multiple *roots*, which makes difficult to implement either atomic upgrades or multiple operating systems installed in parallel [6].

Conary [7] addresses the lack of rollbacks in systems like `rpm` and `dpkg`. Unlike traditional package systems, Conary uses a distributed version control approach to keep track of files in distribution. Packages are stored in a distributed repository, from where those are picked as changsets. OSTree set asides the concept of package and ships versions of a whole operating system tree, it removes the need for numbered versions of installed packages.

Razor is a package manager system that aims to replace `rpm` and `yum`, by proving a fast implementation of package management and dependency solving altogether. Thus, the time window during an update will be reduced, minimizing system corruptions when the system is in an inconsistent state.

As OSTree, NixOS [3] supports the idea of multiple and independent systems that are bootable. In NixOS, the entire system is based on checksums of package inputs (build dependencies), in contrast with OSTree that calculates the checksums based on the object content, including extended attributes. As a consequence, in OSTree any identical data is transparently and automatically shared on disk (as it happens in any `git` repository). Additionally, OSTree is not tied to any particular build system, it is possible to put any data inside an OSTree repository, no matter how it was built. So for example, while one could make a build system that follows the approach of NixOS, it also works to have a build system that just rebuilds individual components (packages) as they change, without forcing a rebuild of their dependencies.

A different approach is to use the package model on filesystems that provide snapshots, such as BTRFS or ZFS, where it is possible to rollback to a previous state of the system. However, the snapshots are applied to the whole filesystem, not only to the packages installed or updated. If the user notices a regression in one or two packages, but has made other unrelated changes in the same filesystem (for instance, in `/etc`), those changes will be lost when booting with the previous snapshot. This can be exacerbated if the upgrade process of a package span multiple partitions.

VI. CONCLUSIONS AND FUTURE WORK

This paper describes two problems that the package model used in Linux-based systems have for developers of projects like GNOME, with a high number of modules interrelated and different kind of developers geographically distributed that need to prepare a release of a system as a whole. In a package model, it is not possible to have multiple versions of the same program, which makes it cumbersome to test development versions of a whole system without the risk of breaking the system. In addition, the detection of regressions can be delayed making the quality assurance process slower. OSTree combines features from a version control and package systems to offer multiple versions of a program and multiple parallel installations. This makes it possible for contributors to try builds from a project like GNOME minutes or hours after a change has been committed in the source code repository.

REFERENCES

[1] M. Michlmayr, F. Hunt, and D. Probert, "Release Management in Free Software Projects: Practices and Problems," in *Open Source Development, Adoption and Innovation*, ser. 234, J. Feller, B. Fitzgerald, W. Scacchi, and A. Silitti, Eds., vol. 234, no. August 1999. Springer, 2007, pp. 295–300.

[2] I. Jackson and C. Schwarz, "Debian policy manual," http://www.debian.org/doc/debian-policy, 1996, accessed: 02/01/2013.

[3] E. Dolstra, A. Löh, and N. Pierron, "NixOS: A purely functional Linux distribution," *Journal of Functional Programming*, vol. 20, no. 5-6, pp. 577–615, Oct. 2010.

[4] E. Flanagan, "The Yocto Project," in *The Architecture of Open Source Applications, Volume II*, A. Brown and G. Wilson, Eds. Lulu.com, May 2012, ch. 23, pp. 347–358.

[5] S. Chacon, "Git internals," in *Pro Git*. Apress, Aug. 2009, ch. 9, pp. 223–250.

[6] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in FOSS distributions: details and challenges," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades - HotSWUp '08*. New York, New York, USA: ACM Press, 2008, p. 1.

[7] M. K. Johnson, E. W. Troan, and M. S. Wilson, "Repository-based System Management Using Conary," in *Ottawa Linux Symposium, Vol. 2*, Ottawa, Ontario, Canada, 2004, pp. 557–571.