Migrating Open Science Grid to RPMs*

Alain Roy[1] on behalf of the Open Science Grid

[1] Open Science Grid Software Coordinator, University of Wisconsin–Madison, USA

Email: roy@cs.wisc.edu

Abstract. We recently completed a significant transition in the Open Science Grid (OSG) in which we moved our software distribution mechanism from the useful but niche system called Pacman to a community-standard native package system, RPM. In this paper we explore some of the lessons learned during this transition as well as our earlier work, lessons that we believe are valuable not only for software distribution and packaging, but also for software engineering in a distributed computing environment where reliability is critical. We discuss the benefits found in moving to a community standard, including the abilities to reuse existing packaging, to donate existing packaging back to the community, and to leverage existing skills in the community. We describe our approach to testing in which we test our software against multiple versions of the OS, including pre-releases of the OS, in order to find surprises before our users do. Finally, we discuss our large-scale evaluation testing and community testing, which are essential for both quality and community acceptance.

1. Introduction

In this paper, we describe how, and more importantly *why*, we changed how we distribute the Open Science Grid (OSG) [1, 2] software stack from being distributed with a useful but niche system called Pacman [3] to RPM [4], the widely used native package system that originated with the Red Hat Linux distribution, but is also used on other Linux variants. (We use the term *native package system* to mean the packaging system that is most natural to use on a given platform, generally the same one used by the operating system installation mechanism.)

The OSG is a multi-disciplinary partnership to federate local, regional, community and national cyberinfrastructures to help share computing and storage resources of research and academic communities at all scales. It provides common services and support for more than 100 resource providers and scientific institutions using a distributed fabric of high-throughput computational services. The OSG does not own computing resources but provides software and services to users and resource providers alike to enable the effective use and sharing of their resources based on the principles of Distributed High Throughput Computer (DHTC). In this paper, we will focus on the OSG Software Stack (also known as the VDT, or Virtual Data Toolkit), which provides the software to enable the implementation and use of DHTC services.

The OSG Software Stack provides both computing and storage middleware as well as client access to these resources. This software is intended for installation by each site's system administrator for use by domain scientists, who will use the OSG to run scientific workflows and access data. The stack includes software to:

* Published in Computing and High Energy Physics (CHEP) 2012.

- accept jobs at a site (i.e., Globus GRAM [5]),
- manage storage resources (Hadoop File System [6], XRootD [7], Bestman [8]),
- manage membership in virtual organizations (VOMS [9]),
- manage authorization at a site (GUMS [10], edg-mkgridmap),
- manage user jobs (Condor [11, 12]), and
- manage pilot jobs (GlideinWMS [13]).

There are many other software packages included as well—this list shows the breadth of included software.

When we first began distributing software as part of projects that preceded OSG (GriPhyN and iVDGL [14]), we used Pacman, a packaging system that is used to download, install, and manage software distributions. Pacman has a number of very desirable features that made it attractive to us, including the ability to install software as either a privileged user (root) or a regular user; the ability to install software multiple times in multiple locations; and support for multiple operating systems. This combination of features allowed us to support a broad mix of users. For example, previous versions of the software stack supported up to seven operating systems (including Scientific Linux, Mac OS X, and AIX) on four different architectures (x86, x86-64, IA64, and PowerPC). [15] Also, because software installations occur in a single non-system directory, users could backup an installation before doing an upgrade. If the upgrade was a problem for any reason, they could easily move back to the previous version. Users would regularly install software once on a shared filesystem to be shared between computers in a cluster. This combination of features was particularly powerful.

In late 2011, we moved away from Pacman and switched entirely to RPM in order to better meet the needs of our community (see more below). Earlier we had made forays into the world of RPMs, but they were unsuccessful. Those efforts attempted to leverage our existing software build mechanisms, and therefore we only produced so-called *binary RPMs*. RPMs come in two flavors: source and binary. The community standard is to use source RPMs, which contain both the source code as well as all the instructions needed to both build and install the software. Our early binary RPMs only allowed users to install the software, but they could not easily modify or rebuild the software. Therefore this time we chose to create source RPMs, which allow our RPMs to be easily reused and modified. (See more details on this in Section 2.) Providing both source and binary RPMs was seen as a significant improvement by our users.

While as a result of moving to RPMs, we no longer have some of the desirable features of Pacman (easy rollback, installation on shared filesystems, natural support for wide array of operating systems), we believe that this move was the right choice for both our users and us.

2. Why move to RPM?

If we lost important technical advantages in using Pacman, why did we switch? The key observation is that technical superiority is not always the primary reason to choose a solution. Instead, finding a solution that works well within a community is more important. Understanding the constraints, operations, and use cases that our community has helped us decide to transition.

The portion of the Linux community that uses RPM (especially Red Hat and clones) has clear guidelines [16] on the right way to package software. The guidelines include rule such as the need for anyone to be able to build the software from source, the proper locations for files, the right way to version software, and much more.

Pacman's primary disadvantage is that it is nearly completely unknown outside of our user community. When an institution hires a new system administrator, there is a good chance that they know about RPM; if they do not, it is easy to find many books and articles to teach them what they need to know. [17, 18] However, there is nearly no chance that a new hire will know anything about Pacman, and there are few community resources to help them.

Perhaps more importantly, it is very hard to share labor with a wider community, which is essential to our ability to sustain software distributions. For example, if someone creates a new source RPM, it

is usually straightforward to share it with many communities: it can usually be reused with little or no work. In addition, it is easy for people to learn how to create RPMs due to extensive existing documentation. However, with Pacman it is not so straightforward. Due to the tight coupling of packages within a Pacman distribution, it is fairly challenging for one person to create a Pacman package and have it be easily usable by other groups. Therefore, by providing RPMs that match the community standards, we gain two significant advantages. First, it is easier for people to donate software packages to us. Second, it is easier for us to donate software packages to other software distributions (such as Fedora, Red Hat, or third-party software repositories such as EPEL [19]). When our software packages become part of these distributions, not only can we continue to maintain them but other people in the larger Linux community may also help to maintain the software, thus sharing the labor with us. Note that our earlier efforts with binary RPMs could not benefit from this community effort at all: not only would they simply not have been accepted as part of other software distributions, but the missing source code and build instructions prevented anyone else from easily helping, even if they were willing.

There are a few potential downsides to this approach. First, we support a smaller set of operating systems than we did with Pacman. Today, we only support Red Hat Enterprise Linux and clones (Scientific Linux and CentOS), with some minimal support for Debian Linux. However, in the last several years the majority of our users have converged on exactly our supported systems. While it would be nice to support a wider variety of systems (and we may do so in the future), the lack is not inhibiting most OSG users. If we expand our support to support a wider variety of operating systems, it will take substantial effort. This is because properly fitting into a community requires understanding that community deeply. Although packaging systems like RPM and deb (used on Debian Linux) appear superficially similar, there are many differences in the tool chain that is used, the definition of a "good" package, proper location of files, and many more technical details.

The next downside is the difficulty in using RPMs to install software on a shared filesystem for user in a cluster. While this is certainly true, most sites are already using cluster management software such as Puppet [20], Chef [21] and CFEngine [22, 23] to manage their sites, including installing RPMs. Thus, extending their sites to install our software stack via RPM is not a serious difficulty. There are a few sites that have found it to be a hardship, and for them we are considering alternative approaches to install a subset of our software (the subset commonly installed on the cluster, called the worker node software) without the use of RPMs. This would not be a completely new implementation but would leverage the existence of our RPMs.

It is useful to think about this change, not in terms of RPM and Pacman, but in terms of in terms of leveraging and sharing with existing communities. When beginning our work on transitioning to RPMs, we defined our Principle of Community Packaging: [24]

> When possible, we should use packages from existing and/or broader communities;
> Otherwise we should make our own packages but contribute them back.
> Therefore, we should package software only when one of the following is true:
>
> 1. The software is not already packaged; or
> 2. The software is packaged but needs significant changes to be acceptable to our users.

The implied corollary of this principle is that we can do this only when we adopt the same packaging mechanism (in our case RPM) that is used in the wider community.

3. How we build and distribute software

3.1. Building the software

We have chosen to use standard community mechanisms to build our RPMs. Because they are well described elsewhere, we only describe them briefly here. We use a system called Koji [25] to manage our builds, which is a standalone build system that allows people to submit source RPMs to be built. It

enforces compliance to some rules, such as not re-using a version number on a package. Internally, Koji uses other standard mechanisms to build the software. In particular, it creates a chroot [26] environment, installs a minimal operating system only the declared pre-requisite software, and converts the source RPM to a binary RPM with the rpmbuild software. The chroot with minimal OS is important because it ensures that the RPM declares all needed pre-requisite software: if it is not declared, it will not be installed and the build will fail.

We have implemented a small layer on top of Koji to help enforce our own internal policies. For example, we require software builds to be reproducible, so therefore all builds that might be released to production must be pulled from our Subversion source code repository instead of from a developer's personal workspace. The source code repository contains the unpatched software source code, any patches we with to apply to the software, and the RPM's "spec" file, which is used to create the source RPM.

3.2. Distributing the software

Users install the OSG software stack with *yum* [27], the standard mechanism for installing RPMs on Red Hat-like operating systems. The Koji software also allows us to manage yum software repositories. We maintain a set of repositories per supported operating system version:

1. *osg-development*: The "wild west" of our work, where OSG Software Team members can place software freely without worrying about the impact on users.
2. *osg-testing*: When the software has had basic internal verification, it can be moved from osg-development to the osg-testing repository. This is the staging ground for future software releases. Brave early-adopter users may get software from here.
3. *osg-release*: The current released version of software. Software must have gone through osg-testing first. This is the standard software repository that most users will use. This repository contains all the software we have released. Although yum by definition installs the most recent version, users can choose to downgrade to older versions in case of problems.
4. *osg-contrib*: This is an auxiliary repository for software that the OSG does not fully support but may be useful to some users. ('Contrib' is short for 'contributed', since much of the software may be contributed by third-parties.) This allows OSG members to quickly share useful software without going through the usual testing and release processes. It is particularly useful when the software is meant for a small subset of OSG.

Although our Koji service hosts these yum software repositories, it is not intended to be a highly available resource. Therefore we mirror the repositories to a central repository hosted by the OSG Grid Operations Center (GOC), which provides a scalable and highly available service. In addition, users can make their own mirrors if they want to ensure reliability in the face of network outages or want to make snapshots of the repository at certain times. Currently three sites provide public mirrors for the yum repository, which improves the scalability and reliability for all of OSG. Yum, unlike Pacman, transparently supports the use of mirrors.

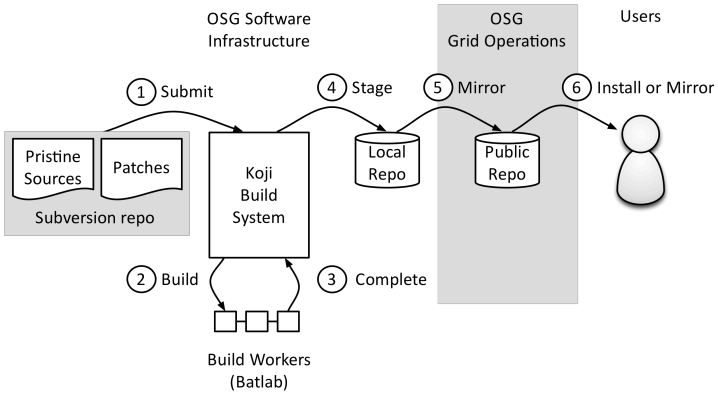

Figure 1: An illustration of the build and distribution process as described in the text. 1. Developers submit builds to Koji based on unmodified source code and (possibly) patches to the software. 2/3. Koji builds the binary RPMs from the source RPM and saves information about the build. 4. Koji puts the RPM into a local yum repository, which is then (5) mirrored to the OSG's central yum repository. 6. Users can either install the RPMs directly from that repository or mirror them to their own local mirrors.

We ensure that every time we modify the osg-release repository we increment the OSG Software version number and write release notes. [28] Users can feel assured that no changes are made without this documentation; thus, we provide clarity for our users. The "version number" is a bit odd, because it covers a set of RPMs, and because users may change the exact set of RPMs they have. For instance they might have installed, say, OSG 3.1.2, but installed some RPMs from OSG 3.1.0, 3.1.1, and 3.1.2. Therefore the version number refers to the set of RPMs that were provided at the time of the release, not necessarily the set of RPMs that someone has installed. That is, the version number is primarily a label that we use to describe the set of RPMs, but users may choose to mix software from different releases (at their own peril).

In unusual circumstances, users may wish to get exactly the set of RPMs that were in a previous release. This is hard to do without extensive archaeological work even though all RPMs are available in the osg-release repository because a user has to know exactly which RPMs should be installed. To simplify this, we have a set of versioned repositories that have exactly the set of RPMs that were provided as part of each release. These are not widely used, but are available as a failsafe mechanism when needed.

When users install the OSG Software, they cannot rely on just our software repositories. Our software also requires software available from the standard operating system distribution as well as a commonly used third-party software distribution called EPEL.

4. How we test software

We have two kinds of software testing that we use to ensure that our releases are of high quality: internal automated tests and wide-area testing in a testbed.

4.1. Internal daily testing

We do daily, automated tests of the full software stack. These are not unit tests of the sort that developers create when writing software, but end-to-end functional tests of the entire software stack. For instance, one test may submit a job from Condor to Globus GRAM, authorize the running of the job with GUMS, and then run the job with PBS. Given the newness of our conversion to RPMs, we are still expanding the set of tests but have reasonable coverage to catch basic problems. Currently the tests are run within a single machine, not across multiple machines. This simplifies the deployment of tests at the expense of being unable to test more complicated scenarios. However, it is still able to catch a significant number of problems.

We test various combinations of our software against the operating system. In particular, we test both the pre-release of our software (from osg-testing, above) and the current release. The purpose of

the pre-release is clear—it allows us to see problems before we release. The reason we test the current release is to ensure that there are no surprises and that we have not accidentally broken our software repositories. We also test against a variety of operating systems (Red Hat Enterprise Linux 5 & 6, Scientific Linux 5 & 6, and CentOS 5 & 6) to ensure that there are no subtle differences between platforms. Although Scientific Linux and CentOS are supposed to be compatible rebuilds of Red Hat Enterprise Linux, we test to ensure there are no problems.[†] We also plan to start testing soon against pre-releases of the operating system. This is particularly important to us because we want to discover problems caused by changes to the operating system before our users do.

Daily emails are sent to all members of the OSG Software Team to notify them of problems. In addition, the results can be viewed on our website.

Operating System	OSG Release
Red Hat Enterprise Linux 5 & 6	Current Pre-release
CentOS 5 & 6	Current Pre-release
Scientific Linux 5 & 6	Current Pre-release
Scientific Linux 5 & 6 pre-release (planned)	Current Pre-release

Table 1: The software combinations we test against

4.2. Wide-area testing

When we have completed initial verification of internal testing, we proceed to end-to-end testing in a wide-area testbed. We have created a testbed for this purpose called the Integration Test Bed (ITB). The ITB is composed of four sites, each maintained independently. The ITB site administrators install software from the osg-testing repository and run a variety of real jobs in realistic scenarios to ensure the software works. While we could in theory replace them by automated scripts, they provide valuable insight we do not get from automated testing. They complain (sometimes loudly) when there are inconveniences in the software installation or the software has bugs we missed in internal testing. Because they are also administrators for production sites, they provide valuable insight and help us avoid problems that would aggravate many users before we ship the final release.

In addition to basic verification of the software, we also run actual scientific workflows through the testbed, to ensure that end-user needs will be met. While we do not require all scientific workflows to be run for all releases, we do run them for major upgrades.

While there is a core of four regular sites in the ITB, we will occasionally gather more sites for our significant updates. For example, the initial release of RPMs involved significantly more sites who provided testing and feedback.

Only after the testing has completed do we release the software stack be to production (the osg-release software repository). The time from building the software to the time it is deployed on users sites can vary considerably depending on the complexity of the software. In some cases, such as urgent security releases, it can happen in a week or less, while in other cases considerable testing may be done and it may take months.

5. Future Work and Conclusions

The RPM-based OSG Software Stack has been in production release on RHEL 5 (and clones) since December 2011. We added support for RHEL 6 (and clones) in April 2012. In the future we will add software and new operating support to support our users.

[†] To date, we have not found any incompatibilities.

There is one particularly important lesson to draw from this paper:

Lesson: When there is a strong community standard, it is often wise to follow the standard even if you have an alternate solution that may be technically better. While not always true (there is often a need for disruptive technologies!), in the case of software packaging we were able to better leverage the existing community packages as well as the skills of people by adopting the community standard. This allowed us to focus our time and energy on other aspects of our work (testing, user support, documentation, etc.) that are more central to our mission.

6. Acknowledgements

Thanks to the entire OSG Software Team, who worked so hard on this transition. They are: Tim Cartwright, Scot Kronenfeld, Terrence Martin, Matyas Selmeci, Neha Sharma, Igor Sfiligoi, Doug Strain, Suchandra Thapa, and Xin Zhao. Significant contributions were also made by Dave Dykstra, Tanya Levshina, and Marco Mambelli. Brian Bockelman and Derek Weitzel were instrumental in both kicking off the transition and guiding us through the technical processes. The Build and Test Facility at the University of Wisconsin–Madison's Center for High-Throughput Computing provided the installation an support for our Koji service. The OSG Operations Team, particularly Soichi Hayashi, set up our central yum repository and dealt with regular requests when we changed our mind in how it was set up. Chander Sehgal and Dan Fraser and the OSG Executive Team provided management consulting. Fermi National Laboratory provided the FermiCloud service that allowed the OSG Software Team members to use virtual machines to develop and test. And, of course, we could not have done any of this work without the vibrant Linux open source community that provided so many excellent tools.

The Open Science Grid is funded by the National Science Foundation and the Office of Science, Department of Energy.

References

[1] M. Altunay, B. Bockelman, and R. Pordes, "Open Science Grid", available from http://osg-docdb.opensciencegrid.org/cgi-bin/ShowDocument?docid=800

[2] Open Science Grid web site: http://www.opensciencegrid.org/

[3] Pacman web site: http://atlas.bu.edu/~youssef/pacman/

[4] http://en.wikipedia.org/wiki/RPM_Package_Manager

[5] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke, "A Resource Management Architecture for Metacomputing Systems". In *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, pg. 62-82, 1998.

[6] HDFS Web Site: http://hadoop.apache.org/hdfs/

[7] XRootD web site: http://xrootd.slac.stanford.edu/

[8] Bestman web site: https://sdm.lbl.gov/bestman/

[9] R. Alfieri, R. Cecchini, V. Ciaschini, and F. Spataro, "From gridmap-file to VOMS: managing authorization in a Grid environment", Future Generation Computer Systems, pg. 549-558, 2005.

[10] GUMS web site: https://www.racf.bnl.gov/Facility/GUMS/1.3/index.html

[11] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience", *Concurrency and Computation: Practice and Experience*, Vol. 17, No. 2-4, pages 323-356, February-April, 2005.

[12] M. Litzkow, M. Livny, and M. Mutka, "Condor–A Hunter of Idle Workstations", *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104-111, June, 1988.

[13] I. Sfiligoi, D. Bradley, B. Holzman, P. Mhashilkar, S. Padhi, and F. Wurthwein, "The Pilot Way to Grid Resources Using glideinWMS," Proceedings of the 2009 WRI World Congress on Computer Science and Information Engineering – 2 (2009): 428-432. doi:10.1109/CSIE.2009.950.

[14] P. Avery, K. Baker, R. Baker, J. Huth, R. Moore, "An International Virtual-Data Grid

Laboratory for Data Intensive Science", 2001.
[15] VDT 1.10.1 system requirements web page: http://vdt.cs.wisc.edu/releases/1.10.1/requirements.html
[16] Fedora Packaging Guidelines: http://fedoraproject.org/wiki/Packaging:Guidelines
[17] Fedora RPM Guide: http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/index.html
[18] Maximum RPM: http://www.rpm.org/max-rpm/
[19] Extra Packages for Enterprise Linux (EPEL): http://fedoraproject.org/wiki/EPEL
[20] Puppet web site: http://puppetlabs.com/
[21] Chef web site: http://www.opscode.com/chef/
[22] M. Burgess, "CFEngine: a site configuration engine", *USENIX Computing systems*, Vol8, No. 3 1995.
[23] CFEngine web site: http://cfengine.com/
[24] A. Roy, T. Cartwright, et. al. "Community Packaging: A Proposal", available from https://twiki.grid.iu.edu/bin/view/SoftwareTeam/CommunityPackagingProposal
[25] Koji web site: http://fedoraproject.org/wiki/Koji
[26] Description of chroot from Wikipedia: http://en.wikipedia.org/wiki/Chroot
[27] Yum Package Manager web site: http://yum.baseurl.org/
[28] OSG Software 3 Release Notes: https://twiki.grid.iu.edu/bin/view/Documentation/Release3/