# Experiences with moving to open source standards for building and packaging

Dennis van Dok, Mischa Sallé and Oscar Koeroo

CHEP 2013 Amsterdam

## 1 What everybody wants

*What everybody wants is to use software, free and unencumbered.*

The reality of software is that it gets in our way more often than not. For the first part of this talk I want to spend a few minutes to share some insights in the mechanisms of software in our field, because it's relevant to the work described in the title of this talk.

**Software midwifery**

The graphic in Figure 1 shows a familiar scene. In the top left we find the *user*, whose motivations we won't question for the sake of brevity. But he or she clearly expresses a desire, and that desire concerns a certain piece of software.

Centrally in the picture is the *software developer.* This person creates what the user desires, and one would conclude that a deal between the two can be made and that's that. The conclusion would be premature, for other than the old lady who bought Windows 95 because a TV commercial told her so and then put it in the window because she didn't actually own a computer, most users wish to *run* the software and that requires hardware.

On the left we see what this meant in the old days, probably predating all but a few here present. A single machine, serving its users one at a time, or in time sharing fashion. If user and developer were not in fact one and the same, the developer was usually found in the same building.

In more modern days with its myriad computer systems the situation is more complicated. The user has to talk to the system administrator to get the software installed. And system administrators have better things to do than figuring out how to install a certain piece of software. It either installs straight away or not at all. The dreaded answer 'no' is heard all too often.

Mind you that the situation does not improve if the user and the administrator are one and the same. Users are just as unlikely to go through the mental agony to install difficult software.

If we look at the bottom half of this graphic, we can see that that the developer is hardly to blame for the situation. He delivers version after version of the good stuff. The only problem is, the good stuff is raw material. Good enough for a log cabin, but it needs to be processed before it's suitable for home furniture.

Depicted downstream is a happy little mill that works the logs into something that requires little or no assembly at home. Is the miller is a person of flesh and


Figure 1: How do we get software to the user?

blood, or an automaton? It requires a bit of both, as it turns out. The craft of this fourth role, that of the packager, is not often found among the ranks of people involved in the software conundrum. And that is a rotten shame.

# 2 History

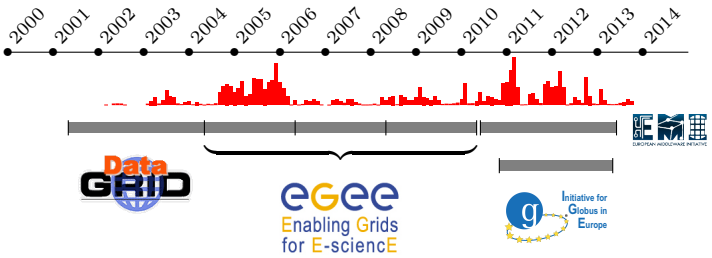
In 2001 the DataGrid project kicked off a series of EU funded projects to develop a *European Computer Grid;* this involved everything from infrastructure to middleware. We contributed several components to the middleware, notably LCMAPS and gLExec; to this day we maintain a dozen or so packages.

The red bar graph represents the number of commits in version control for our middleware. It shows that the heaviest development happened during EGEE-I. For EGEE-II, consolidation of the software stack and supporting the production grid were the main focus.

**Where we come from**

- Middleware contributions in a series of EU funded grid projects: DataGrid, EGEE (I, II and III), EMI and IGE.

- current sustained 'maintenance mode' through SURF e-infrastructure.




# 3 ETICS

As the grid grew, the EGEE project struggled to manage to support the software stack for production use *and* development. The coordination effort between contributing teams, and the demand for reliable software called for a system that gave the integrators full control over the builds. Thus ETICS was born, and it would grow up to become a EU funded project on its own.

The ETICS system worked most of the time, but it was a hassle work with. Every small change meant going through a series of manual steps in a quirky web interface, and a build could take hours to be scheduled.

Essentially we had to track all our up- and downstream dependencies, and our software needed modifications specifically to fit the project. We even needed to make special accommodations to produce a variant build of our software for OSG.

Although the software was still, in name, open source, in practice it might as well have been closed source at this point. It was hard if not impossible to build outside of the ETICS environment. It was hard or impossible to build for other platforms than those targeted by the project, essentially only Scientific Linux.

Still, the system worked. There were no alternatives, so we had to use it. To be fair, in the EMI years some improvements were made. But it was too little, too late. The ETICS project came to a halt pretty much at the same time as EMI and IGE (the initiative for Globus in Europe). The portal was shut down and the product teams had to find other means of managing their own codes.

**ETICS' shortcomings**

- It was not easy to build outside of ETICS
- Building in ETICS was too slow for debugging/test cycles
- The system required specific `m4` macros to build
- There was little or no focus on portability
- The output consisted of binary products and no rebuildable source material.

- The ETICS project stopped in 2013 and the portal went off-line.

**We had to do things ourselves.**

# 4 What we did

At the end of the EGEE era in 2010 it was clear that there would be no indefinitely prolonged funding for middleware. We started to think about the sustainability of our software.

The basic idea is this: if we were going to have to support this software, we should be in charge of how it is built and released. We need to be able to reproduce bugs, and trace software behavior to actual lines in the source code.

The list of code improvements on this slide is not in any particular order.

- The automake/autoconf macros that were specific for the ETICS system had to be generalized to work outside of that environment.
- To improve portability, we stuck to using the POSIX standard where we could.

The first fruits of our efforts were the improved ease with which software could be configured and built on a laptop. This drastically reduced the time for a debug/fix/build/test cycle.

Looking around, we saw that the major distributions had already addressed the issue of package management and had developed guidelines on how software should behave—both at run-time and during the build phase—and that these were all very good ideas. We decided to actively pursue sticking to these guidelines for our own software. In comparison, many of the other middleware components in EMI were not in the same shape and the project's focus was on establishing an easier way to build RPM packages for them.

We had to work a hybrid mode for a while, where each improvement to make the build less dependent on ETICS had to be augmented with tricks to make it still work with ETICS. This became easier when ETICS started to move to using *mock* for building RPMs.

- We designed and documented a software process for building, tagging and packaging.
- We implemented packaging for Fedora/Red Hat and Debian/Ubuntu following their respective guidelines.
- We cleaned up dependencies, so that the complete inclusion of external packages such as gsoap was no longer required.
- Improved documentation, such as man pages.
- We documented our software process
- We actively tested the portability of some components to many different platforms, which uncovered more bugs.
- Cleanup of insecure constructs and implementation of secure coding practices, following an independent security assessment.

We changed from using CVS at CERN to our own subversion, after importing all of the history.

We set up our own Koji system (modeled after Fedora) to automate the builds.

In the later phase of the EMI project, the ETICS builds for our components were actually based on the source RPMs from our own build system.

# 5 What we achieved

Our software is configured with the GNU Autotools: Autoconf and Automake. This is a reasonable choice for C programs. The common pattern for building such software is basically this:

1. download a tar ball
2. unpack the tar ball
3. run configure; make; make install

Our software was (initial) not able to do this very simple thing. Nowadays we stick to running

```
make distcheck
```

for all our packages. This makes sure configure and make can be run outside the source directory, build and install cleanly to a stage directory, and uninstall cleanly. Once this is accomplished, the packaging is relatively little work.

**What we (eventually) managed**

Most (autotools based) open source software can do this out of the box:

```
./configure
make
make install
```

We actually stuck to the mantra:

```
make distcheck
```

which builds outside the source tree and tests if an install with `DESTDIR` works.

# 6 How we roll

The following steps can, in principle, be done by anyone. Many open source packages are maintained by people with no direct involvement in the software development. But for our software we do all our own packaging.

The phases here start from the tarball that is produced in the previous section.

The packaging is done differently for various distributions; the commonality is that they all use recipes which describe the way software should be unpacked,

configured, build and installed; but the formulation of these recipes is radically different between Fedora, which uses rpm packaging from a single SPEC file, and Debian, which has its own packaging system based on an entire directory of files.

We keep both package descriptions in version control as well, separate from the sources. We treat ourselves as 'upstream' when we put the packaging hat on.

We make use of version control for packaging by using a commit trigger on the `tags/` tree to launch the corresponding Koji builds.

**Fedora packaging**

- Tagging in SVN (of the `SPEC` file) triggers a Koji build
- Koji does mock builds of the source and binary RPMs for all targeted platforms, i.e. the latest Fedora releases and EPEL5 and EPEL6.
- The builds are signed with a tool called *sigul.*
- The *mash* utility generates the repositories that can be installed through yum

*koji, sigul and mash are also used by the Fedora project.*

**Debian packaging**

For Debian, the procedure is slightly different. There is no equivalent of Koji for Debian ☹.

- a Debian source package is created for currently supported Ubuntu versions, Debian unstable, stable and oldstable,
- each source package is build with cowpoke/cowbuilder/pbuilder (equivalent to mock).
- the resulting packages are signed with the packager's GPG key
- The package is uploaded to a software repository from where it can be installed with apt-get.

**Catching common errors**

For Fedora, use `rpmlint`; for Debian, `lintian` to see if packages do not contain silly mistakes (lintian helped find several common spelling errors.)

The automated build logs revealed more warnings due to using different compiler settings.

This is not a substitute for real testing, of course.

# 7 Who we support

**Communi{ty,cation}**

- Mailing lists We've set up a few mailing lists:
  - grid-mw-security-support@nikhef.nl for support questions and
  - grid-mw-security-announce@nikhef.nl for announcements of new versions. This list has an open subscription policy.

  No general discuss mailing list (yet)... no sizable community either (besides wLCG/EGI there is OSG).

**Sources, binaries and bugs**

- Version control in SVN
  https://ndpfsvn.nikhef.nl/viewvc/mwsec/
  https://ndpfsvn.nikhef.nl/ro/mwsec/
- Download sources
  http://software.nikhef.nl/security
- RPM/Deb distribution
  http://software.nikhef.nl/dist
- Bug tracking **NEW:** https://bugzilla.nikhef.nl/, moving away from CERN Savannah.

# 8 What we learned

Going through all of the trouble of setting up the right build environment, cleaning up the source code and packaging, familiarising ourselves with the guidelines, and defining policies and procedures was quite an investment. The question that rises is whether this investment is paying off.

It's not easy to measure. But each step along the way made sense on its own. Either it reduced the time to build, or improved the quality of the software This went on as a step by step process.

Doing everything the 'open source way', meaning that we play along with common practice and be open and nice about what we do, and how we do it. There are really few barriers for inclusion in mainstream distributions and some of our software made it into Debian (with some help) and we're still aiming for inclusion in Fedora.

None of this really works towards attracting a large crowd of users or developers, but the benefits are already there, even without external help.

We were also able to help others within the project because of our early experiences.

The guidelines are in fact a distillation of years of knowledge of advanced packagers. They are not so much red tape as actually useful and meaningful in their own right. We have already experienced ourselves what the benefits are that come out of following good practice.

*The sustainability of our software has been improved greatly since this move.*

**What we got in return**

Some of the benefits our work rendered:

1. playing fair with package management avoids conflicts
2. installation of software becomes trivial
3. reproducing bugs becomes easier
4. pin-pointing bugs to source lines is easier
5. cycle time to deliver updates becomes shorter
6. uncovered some lurking bugs
7. improved portability
8. easier integration with third parties
9. using common technology makes it easier to pass the support to future staff members.

**Last slide**

This slide is intentionally left blank.

How to tell if a FLOSS project is doomed to FAIL

Our overall fail score: 55 points.

**References**


- Guide to setting up the Koji build system
  http://fedoraproject.org/wiki/Koji/ServerHowTo
- Nikhef Security Access Control sofware procedures
  https://wiki.nikhef.nl/grid/SAC_software_procedures
- Fedora packaging guidelines
  https://fedoraproject.org/wiki/Packaging:Guidelines
- Debian Policy
  http://www.debian.org/doc/debian-policy/
- Debian upstream guide
  https://wiki.debian.org/UpstreamGuide