Supporting the Development Process of the DataGrid Workload Management System Software with GNU autotools, CVS and RPM[*]

A. Gianelle, R. Peluso[†] , M. Sgaravatto, INFN Sezione di Padova, Via Marzolo 8, I-35131 Padova, Italy
F. Giacomini, E. Ronchieri[‡] , INFN CNAF, Viale Berti Pichat 6/2 , I-40127 Bologna, Italy
G. Avellino, B. Cantalupo, S. Beco, A. Maraschini, F. Pacini, DATAMAT S.p.A., Via Laurentina 760, I-00143 Roma, Italy
A. Guarise, R. Piro, A. Werbrouck, INFN Sezione di Torino, Via P. Giuria 1, I-10125 Torino, Italy
D. Kouril, A. Krenek, Z. Kabela, L. Matyska, M. Mulac, J. Pospisil, M. Ruda., Z. Salvet, J. Sitera, M. Vocu, CESNET z.s.p.o., Zikova 4, 160 00 Praha 6, Czech Republic
M. Mezzadri, F. Prelz[§] , INFN Sezione di Milano, Via Celoria, 16, I-20133 Milano, Italy
S. Monforte, M. Pappalardo, INFN Sezione di Catania, Via S. Sofia 64, I-95123 Catania, Italy
D. Colling, Imperial College London, Blackett Laboratory, Prince Consort Road, London, SW7 2BW, UK

Abstract

Supporting the development of the Workload Management System in the context of the European DataGrid was a challenging task as the team was characterized by a high geographic and administrative dispersion, with developers distributed in various institutions and countries. Further, software dependencies were complex as it was required to integrate and interface a significant number of external software packages. In this paper, we discuss how a combination of Concurrent Version System, *GNU autotools* and other tools and practices was organized to allow the development, build, test and distribution of the software. With the proposed solution, we managed to combine ease-of-use for distributed developers while preserving the central coordination needed by the project-wide steering.

INTRODUCTION

The distributed development model of the European DataGrid (EDG) project had to deal with several issues. A large number of persons spread all over Europe had to write software packages that are inter-dependent, therefore called for frequent integration. In particular, the Workload Management System (WMS) [1] development was rather complicated because the team was both geographically and administratively dispersed (four institutions with developers at nine different locations in three countries). Moreover, the software dependencies were numerous and complex. Therefore, the Workload Management Architecture was divided into components under the responsibility of local development teams.

The fundamental requirement for concurrent development directed us to the use of Concurrent Version System (CVS) [2], which is the most common solution in the open software community. CVS allows developers to separately modify a file and to keep track of the changes made by the others. Moreover, it allows for several software versions in one repository.

[*] This research was partially funded by the IST Program of the European Union under grant IST-2000-25182 (DataGrid project)

[†] rosario.peluso@pd.infn.it

[‡] elisabetta.ronchieri@cnaf.infn.it

[§] francesco.prelz@mi.infn.it

The EDG WMS package contains daemons, libraries, test programs, documentation, etc. It is divided into components that encapsulate independent functionalities part of the package. The whole package is organised in a single directory tree. Each component is identified by a subdirectory and the main directory name is "workload". Further levels are present inside each component sub-directory. These inner levels do not have a common structure, as components are quite different: some of them are daemons, while some others are libraries. The lack of a common structure makes even more difficult to have a simple and common build strategy for each of them. Clients of the daemons, for example, are claimed to be compiled even by different components than the current one. This is the main source of inter-dependency between components themselves.

In the remaining part of the paper, we show the EDG WMS dependency categories, we discuss how to configure, release and distribute the EDG WMS using *GNU autotools* and other tools and practices.

THE EDG WMS DEPENDENCIES

In this section we show a simplified version of internal dependencies present in the WMS components (see Figure 1) for the sake of simplicity.

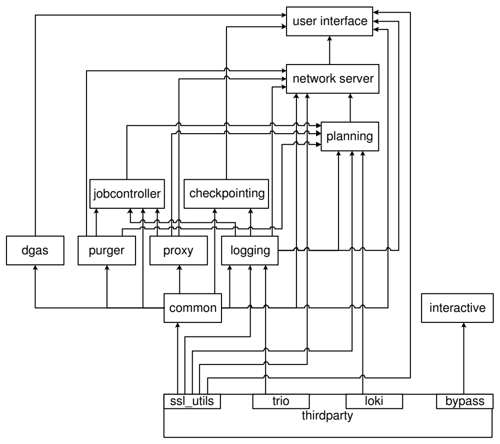

Figure 1: The WMS dependency graph.

Each box represents one of the WMS components. Sometimes part of the internal structure of a component is shown with a small box. The direction of the arrow means that the destination object depends on the source one.

The components contained in the WMS have a complex dependency structure. The dependencies can be divided into four categories:
1. **Non-EDG packages:** packages that are developed outside the EDG project, such as MySQL, Boost libraries, Condor libraries and executables.
2. **Non-WMS EDG packages:** packages that are developed by other EDG work-packages. For instance, in this category we can find the Data Management and Information Service libraries.
3. **Modified non-EDG packages:** packages developed outside the EDG project that needed to be modified by WMS work-package. Bypass, trio and the Globus FTP server belong to this group. They needed to be customised to meet our requirements and they are managed and distributed by the WMS work-package.
4. **WMS components:** software components developed entirely by the WMS work-package. The cross-dependencies among these components make it rather difficult to cleanly structure them as separate packages. However, the internal dependencies were identified in order to provide several WMS RPMs and not just a single monolith.

These constraints specifically affect the semantics of the package configuration options, which must have the ability to scan and resolve the dependencies needed by either a single WMS component or the entire system.

GNU autotools AND THE EDG WMS CODE

One of the requirement for the EDG WMS software was to build and run on the architecture RedHat Linux 7.3. Since we wanted to easy the port to possibly new platforms, we decided to adopt a generalized approach in all software packages and components by using *GNU autotools* [3], which refer to GNU Autoconf [3], Automake [3], and Libtool [3] packages.

GNU autotools provides developers and maintainers a set of prepackaged and flexibly modifiable tests for various conditions that may differ across systems. *GNU autotools* also simplify the build and distribution of source code programs, as the building is organised in a simple, standardised two step process (`configuration` and `build`), which does not require the instantiation of any special tool in order to compile the code. The `configuration` step generates `Makefile`'s and perhaps other files, which are then used during the build step. The `build` step uses the standard Unix `make` program, which reads a set of rules in a `Makefile` and use them to build the program.

GNU autotools also allow to cleanly address a specific requirement of the WMS, namely the configuration of individual components inside the same package, and the handling of numerous and diverse external or third-party packages and libraries. In addition to this, they can be easily used to obtain the configuration even of sub-packages. Finally, the handling of external or third part software and libraries is quite easy.

In the EDG project it was decided to release and distribute the code using the RPM Package Manager [4], as it provides extensive and accessible package management services. We directed RPM in the build process by creating a `spec` file. In order to accomplish this task we needed to follow simple DataGrid rules [5] and added some internal procedure to quickly check the WMS RPMs.

Configuration, Release and Distribution of the EDG WMS Structure

Each dependency affects the configuration of the whole package. The main goal of our configuration system is to allow the developers of a single component to compile as little as possible instead of compiling the whole package. The build of the entire package can take up to 90 minutes on a commodity PC, e.g., give a footnote such as "Pentium IV, 1.6 GHz, 512 MB RAM". Furthermor, not all external dependencies are required by every component. Capturing the complex component interaction pattern shown in Figure 1, has proven to be difficult and, in some specific cases, impossible with the available tools.

We have found necessary to develop some sort of *autodep* tool that allows to express and manage both internal and external dependencies in a fashion similar to the "common" autotools.

This work has been started but not finished due to lack of time, and it has become a future work. In this situation, the "fast and furious" way has been to hardcode enabling options and conditionals directly into the `configure.in` file. Each conditional was tied to a submodule (or part of it) and its (de)activation has been obtained with a large number of `''if''` statements hardcoded in the `configure.in` .

The effect of enabling some components or parts of them is the possibility to allow the building of specific programs/libraries or to include entire directory trees necessary for that part. The compilation of each component is enabled by default: it can be explicitly disabled by using the appropriate configure option (`--enable-`*`submodulename`*`=no` or `--disable-`*`submodulename`*) or by enabling another module that does not depend on it. Each enabled submodule will also trigger the check for the presence of any other external package related to it.

As previously mentioned, EDG WMS depends on various packages that are provided either by the DataGrid project or externally. In order to correctly detect the presence and the position of such packages, it is necessary to create a large number of specific tests and set a number of variables. These tests may be put directly inside the `configure.in` file, but this would make this file difficult to read. For this reason, we decided to create a specific M4 [3] file for every package that does not already provide one. These M4 files define just one macro called for instance AC_*PACKAGE*. In general they take three arguments: the version of the package (when applicable), the action to perform if the correct package is found, and the action if it is not. The purpose of such macros is to check whether (and where) the include files and libraries are available (if the package is a library), or to try to understand the path in case of executables (for example for Perl). Sometimes they perform both operations (for example for Swig). The macros of the first type define two (or more, in some special cases) Makefile variables, usually called *PACKAGE*_LIBS and *PACKAGE*_CFLAGS. The first one will contain the path and the name(s) of the library(ies), while the second will contain the path for the include files (if present). Some of these macros will also define C macros containing other information useful to the compiler. The second kind of macros, instead, defines Makefile variables usually called RUN*PACKAGE*, which will contain the full path for the required executable.

For each new release, the procedure adopted was the following. First of all, the set of bugs and new features to be addressed (as extracted by the project bug-reporting system) is defined, and communicated via e-mail or on the IRC channel (where most of the communication within the WMS work-package occurred). When all the related development issues are resolved, an e-mail is sent to the work-package WP1 mailing list, communicating the start time of the test session. Before this time, developers have to commit all pending changes with respect to the upcoming release. When the announced time arrives, a CVS branch called test_*<version>* is created. Software is entirely rebuilt starting from that branch and various tests are performed, including the execution of the work-package WP1 specific regression test suite. If errors are found, these are fixed and committed to the branch. When the test results are satisfactory, the release is tagged on the branch and all the applied fixes are merged to the main trunk.

Example of enabling/disabling a component

We show an example of how we allow the enabling of a component and how its dependencies are handled inside the configuration. We have chosen a "simple" case where few dependencies need to be handled: the proxyrenewal - a standalone daemon that retrieves renewed user proxies from an external service -.

- Put some M4 macros in the configure.in **-** In the configure.in file, we add a conditional variable ''opt_enable_renewal'' and put it equal to the value ''yes''. Together with it, we need to add the corresponding Autoconf M4 macro in order to have the correct configuration switch in the configure script:

```
AC_ARG_ENABLE(renewal,
    [ --enable-renewal build proxy renewal
    [default=yes]],
    enable_renewal=''$enableval'',
    enable_renewal=no)
```

Note that if the option is not supplied to the configure script, the default value of the enable_*submoduleoption* is ''no''.

In the configure.in file, after the declaration of all the configure options and variables, an articulated ''if'' statement is used to disable the submodules not explicitely enabled in the configure invocation:

```
if test ''x$enable_opt1'' = ''xyes'' \
    -o ''x$enable_renewal'' = ''xyes'' \
    -o ''x$enable_optN'' = ''xyes'' ; then
    opt_enable_opt1=$enable_opt1
    opt_enable_renewal=$enable_renewal
    opt_enable_optN=$enable_optN
fi
```

The net effect of this test is that if one or more options are enabled, only the corresponding ''opt_enable_*'' variable is set to the value ''yes''. Once all these variables (e.g ''opt_enable_*opt1*'' and ''opt_enable_*optN*'') have the correct value, several tests are done in order to check what external dependency has to be tested for. The values of the ''opt_enable_*'' variables are then copied to another variable (e.g. ''have_renewal=$opt_enable_renewal''). These have_* variables are then tested together with the conditions on external dependencies in order to understand which of the (selected) submodules have to be built. This set of tests will result in having all the have_* variables set to ''yes'' or ''no''. They are then used to enable the appropriate Automake [3] conditionals, using the construct AM_CONDITIONAL(AMC_BUILD_RENEWAL,.). In this example the have_renewal variable may also enable other conditionals, such as AMC_BUILD_COMMON, AMC_BUILD_THIRDPARTY and others on which proxyrenewal depends. The system is made in a way such that by enabling another component that depends on proxyrenewal, all these conditionals are automatically enabled too, together with any other one specific to that component.

Inside the configure.in file, M4 macro called AC_MYPROXY is used. That is, if one of the components that depend on the MyProxy external package is enabled, the test for its presence is carried on. In this specific example, we were interested in this test if the opt_enable_renewal variable was set. In order to be sure that the value of the have_* variables is something like ''yes'' or ''no'' their values are set by default to ''no'' before actually performing all these tests.

```
if test ''x$opt_enable_w_renewal'' = ''xyes'' \
    AC_MYPROXY([],have_myproxy=yes,have_myproxy=no)
fi
```

- Put some conditions in the Makefile.am **-** These conditionals are used in the Makefile.am [3] files by the automake tool to understand which programs, libraries and subdirectories have to be included in the build process. In the main Makefile.am (the one in the workload directory) we will find something like:

```
if AMC_BUILD_RENEWAL
WL_RENEWAL = proxyrenewal
endif
...
SUBDIRS = config m4 $(WL_SUBDIR1) ...  \
   $(WL_RENEWAL) ...  $(WL_SUBDIRn)
...
```

The `WL_*` variables represent the subdirs to be included in the build process.

In the `Makefile.am` relative to the `proxyrenewal` subdir we will find something like folow:

```
if AMC_BUILD_RENEWAL
bin_PROGRAMS = edg-wl-renew
...
endif
```

That is, we will compile the `edg-wl-renew` command line.

Example of automatically delivering WMS RPMs

In this section we show an example of how we automatically deliver WMS RPMs. We describe how we check `spec` files before build WMS RPMs.

- Put some M4 macros in the `configure.in` **-** We added the M4 macro `AC_EDG_RPMS`, which sets the directory where WMS RPMs must be built. Its default is `'pwd'`. We also added another M4 macro `AC_RPM`, which defines the variables `RPM_LIBS`, `RPM_CFLAGS`, and `RPM_BIN_PATH`. These variables are used in the `Makefile` to build a simple piece of code that reads spec files and returns the ones that go in the RPMs.

- Put rpm target in the `Makefile.am` **-** The main `Makefile.am` (the one in the `workload` directory) includes the `rpm` target. As a consequence, it is enough to run the command `make rpm` to build the WMS RPMs. In our package we have organized the code in four `spec` files. The main one covers the WMS services and APIs: two of them apply to a couple of external packages that needed modifications, the last one just includes the testsuite description. In addition, we have added another target called `make rpm-check DESTDIR=<install location>`, in order to avoid the full time-consuming command `make rpm` in case of errors in the package (e.g. when a new header file was added in the code but not included in the `Makefile.am`). This target builds the program `checkfiles` which reads the spec files and extracts the list of files that goes in the RPMs, which is saved in a file called `rpmfiles.tmp`. Then it runs the commands `make apidoc`, and `make install DESTDIR=<install location>`. The following piece of code produces two files: `installedfiles_notinthespecfile.txt` and `specfiles_notinstalled.txt`. Once obtained such files the file `specfiles_notinstalled.txt` can be edited to check wheather the spec files contain some old files. By viewing the file `installedfiles_notinthespecfile.txt` it is possible to verify if the spec files need to be upgraded.

CONCLUSION

WMS is developed by a group of persons working for different institutions in different European countries. The support of the development activity in such situation require both an appropriate choice of tools and organizational guidelines. In our case, the role of the *packager* was introduced, with the task of organizing the code tree structure, providing templates for the packaging of new components, overseeing on the application of project-wide rules [5] and on the uniformity of build procedures. While this role entailed was called to solve many build and installation issues for any component, it also allowed all developers to converge towards common formats for the `Makefile.am` and M4 files, and a `configure.in` organized for all needed tasks. Finally, it allowed developers to concentrate just on code development without worrying about its overall organization.

We have summarised our experiences, the limits found and the extensions added to standard code management and packaging tools adopted by the EDG project, in order to accomodate the needs of the Workload Management. A number of problem related to the integration of a very complex set of internal and external dependencies were succesfully identified and solved. The main tools used are: the CVS repository, the *GNU autotools* to manage the building of the package, and RPM for package management. The main missing functionality discovered was related to the description of dependencies at the *GNU autotools* level, for which a dedicated "autodep" tool was felt to be useful.

ACKNOWLEDGEMENTS

The authors wish to thank the WMS team for their help and support in developing a better organization of the WMS package. We also thank the EU and our national funding agencies for their support of this work.

REFERENCES

[1] G. Avellino *et al*, "The EU DataGrid Workload Management System: towards the second major release", 2003 Conference for Computing in High-Energy and Nuclear Physics (CHEP03), La Jolla, California, 24-28 Mar 2003.

[2] K. Fogel *et al*, "Open Source Development with CVS", Coriolis Group, October 2001.

[3] G.V. Vaughan *et al*, "GNU Autoconf, Automake, and Libtool", New Riders, October 2000

[4] D. Barners, "RPM HOWTO. RPM at Idle", Free Software Foundation, November 1999.

[5] Quality Assurance Group, European DataGrid Developers' Guide (2003).