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

Elisabetta Ronchieri
(on behalf of the EDG work package 1)

CHEP 2004, 27. Sept. - 1. Oct., Interlaken

Acronyms

- European DataGrid (EDG)
- Workload Management System (WMS)
- European DataGrid Workload Management System (EDG WMS)
- Concurrent Version System (CVS)
- RPM Package Manager (RPM)

Content

- Problem Description
- The EDG WMS Code structure
- The EDG WMS Dependencies
- Brief - GNU autotools – autoconf, automake, libtool
- The EDG WMS Code configuration, releasing and distribution
- Deployment Procedure
- A bit of sociology (D'HO)
- The EDG WMS Status
- Future Work
- Conclusion

Problem Description 1/2

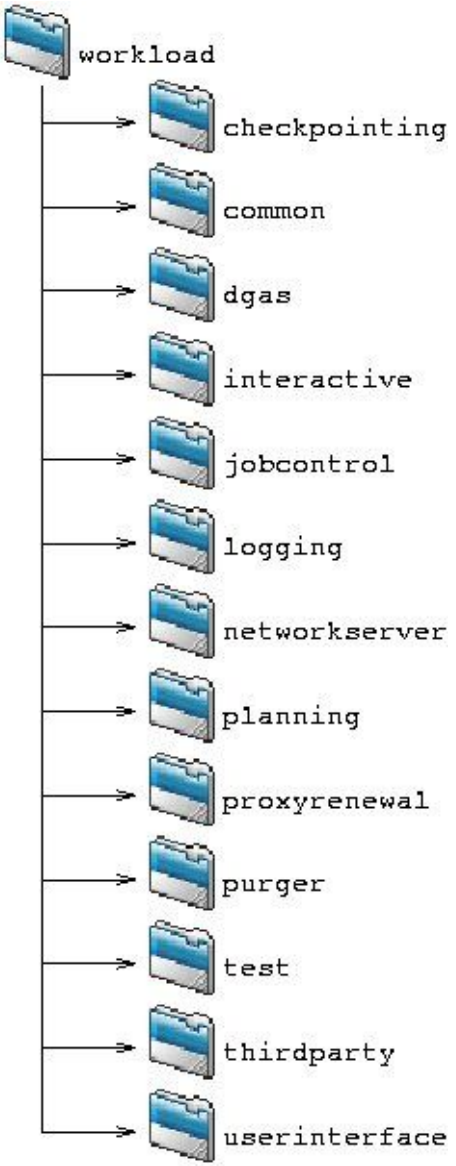
- The team was characterized by a high geographic and administrative dispersion (as shown by the ↘)
- The EDG WMS Software was divided into several components under the responsibility of local development teams

Problem Description 2/2

- Concurrent development and maintenance of various versions of the same code file were required
- Software dependencies were complex
  - it was required to integrate and interface several external and internal dependencies
- Software organization was not well defined
  - part of the code was already written
- Software had to build and run on the architecture Redhat Linux 7.3 and other platforms
  - currently it runs on Redhat Linux 7.3

The EDG WMS Code Structure

- The EDG WMS Code contains daemons, libraries, etc
- The EDG WMS package is organized in a single directory tree shown on the right
  - the main directory is called workload
- The package is divided in smaller components
  - Further levels are present inside each component sub-directory
  - They do not have a common structure
    - It was difficult to have a simple and common build strategy for each of them (NOTE!)

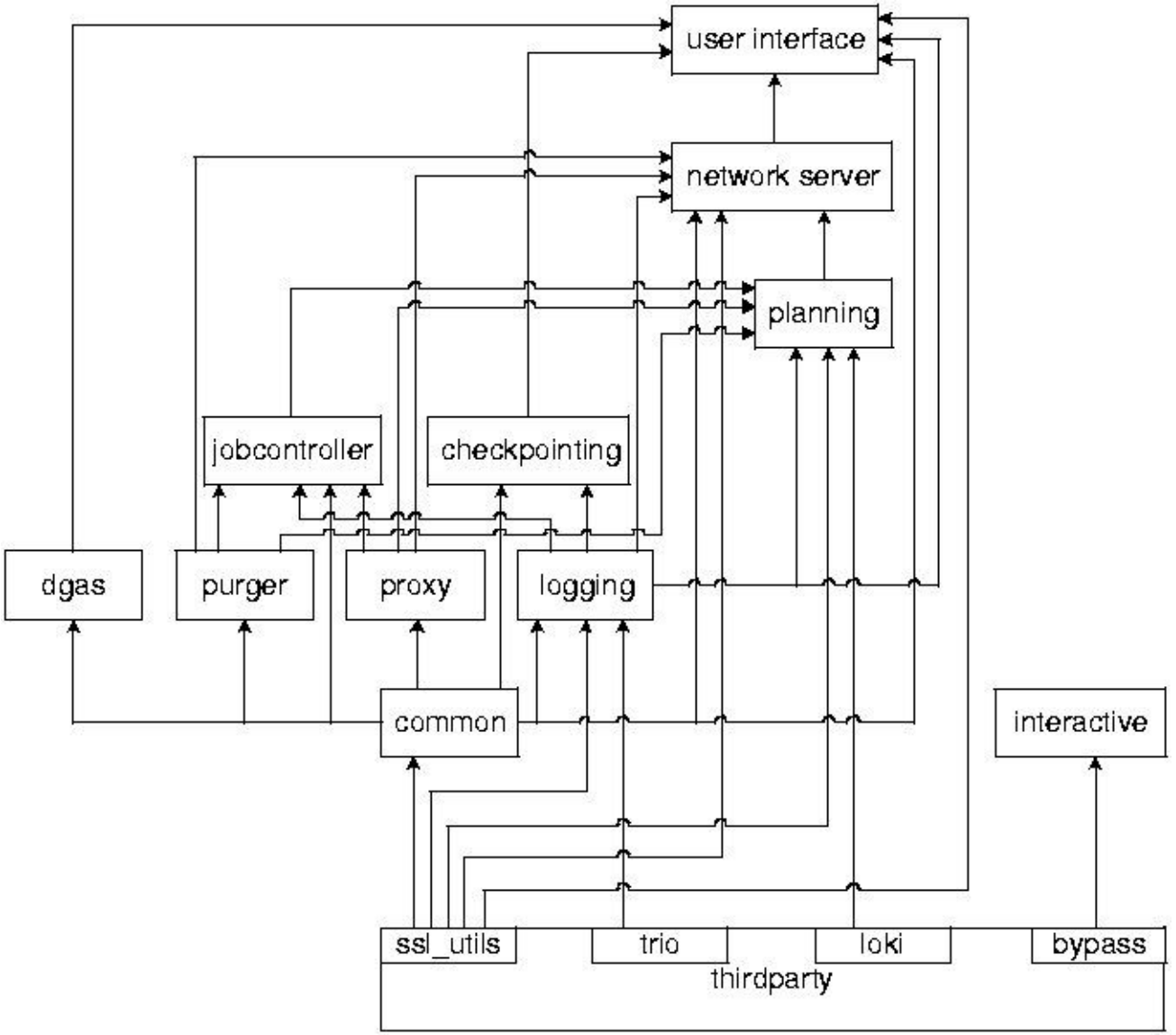


The EDG WMS Dependencies

- The dependencies are divided in four categories:
  1. Non-EDG packages – developed outside the EDG project
  2. Non-WMS EDG packages – developed by other EDG work packages
  3. Modified non-EDG packages – developed outside the EDG project that need to be modified by the WMS work package
  4. WMS components – developed entirely by WMS work package
- They have been scan and resolved by the configuration options (NOTE!)

The EDG WMS Internal Dependencies

- The EDG WMS internal dependencies are shown on the right
  - Each box represents one component
  - Part of the internal structure of a component is shown with a thin box
  - The direction of the arrow means that ”the destination object depends on the source one”



GNU autotools: automake, autoconf, libtool

- provide developers a set of prepackaged and flexibly modifiable tests
- simplify the build and distribution of code
- organize the building in two steps
  1. configuration - generates Makefile's and perhaps other files, used by the build step
  2. build - uses the Unix make program, which reads a set of rules in a Makefile and use them to build the program
- enable/disable the build of a subset of the components (EDG WMS requirement)
- easily configure sub-packages

GNU autotools: automake, autoconf, libtool

- autoconf is a system of M4 macros and Bourne shell script
- configure.in is a template of M4 macro invocations and shell code fragments, used by autoconf to produce configure script
- autoconf performs small tests designed to check each feature
- automake generates any number of Makefile file
- Makefile.am is a template of Makefile file read by automake to produce Makefile.in file
- These Makefile.in files are used by the configure script at configure time - configure script performs some substitutions on the templates to produce
- libtool is used to automate the building, linking, and installation of shared and/or static libraries
- M4 macros are used to compile the configuration files

The EDG WMS Code configuration, releasing and distribution

- Developers of a single component wanted to compile as less module as possible
- Not all external dependencies are required by every component
- Enabling options and conditions were hard-coded into the configure.in file
- The component (de)activation has been obtained with several if statements hard-coded in the configure.in file
- Each component is enabled by default
- Each enabled component will trigger the check for the presence of any other external package related to it

The EDG WMS Code configuration, releasing and distribution

- Specific M4 file for every external package has been created to detect the presence and the position of such package, testing easy functions, and set some variables
- M4 files define some macros and Makefile variables
  - e.g. if the name of the external dependency is PACKAGE
    - AC_PACKAGE
      - takes three arguments: version (when applicable), action to perform when the right dependency is found, action when it is not
    - PACKAGE_LIBS
      - contains the path and the name of the library(ies)
    - PACKAGE_CFLAGS
      - contains the path for the include files (if present)
    - RUNPACKAGE
      - contains the full path for the required executable

Example: enabling/disabling the proxyrenewal component

- configure.in

```
opt_enable_renewal=yes
.....
dnl
dnl proxyrenewal option
dnl
AC_ARG_ENABLE(renewal,
[--enable-renewal build proxy
renewal [default=yes]],
enable_renewal=``$enableval``,
enable_renewal=no)
.....
```

```
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
.....
have_renweal=$opt_enable_renewal
.....
```

Example: enabling/disabling the proxyrenewal component

- configure.in

```
.....
if test ``x$have_myproxy``=``xno``;
   then
have_opt1=no
have_renewal=no
fi
.....
AM_CONDITIONAL(AMC_BUILD_RENEWAL,
test x$have_opt2=xyes\
-o x$have_renewal=xyes)
.....
```

```
.....
if test ``x$opt_enable_opt2``=``xyes`` \
-o test ``x$opt_enable_renewal``=``xyes``;
then
AC_MYPROXY(4.1.1, have_myproxy=yes,
have_myproxy=no)
fi
.....
```

Example: enabling/disabling the proxyrenewal component

- Makefile.am (in the main directory workload)

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

- Makefile.am (in the directory proxyrenewal)

```
if AMC_BUILD_RENEWAL
sbin_PROGRAMS=edg-wl-renewd
bin_PROGRAMS=edg-wl-renew
noinst_LTLIBRARIES=libedg_wl_renewal.la
endif
....
```

Example: delivering automatically EDG WMS RPMs

- configure.in

```
....
AC_EDG_RPMS
.....
AC_RPMS
.....
```

Makefile.am (in the main directory workload)

```
....
rpm: $(RPMS_SPECS)
  .....
rpm-check:
  .....
```

- There are four spec files:
  - one covers the EDG WMS services and API
  - two of them apply to a couple of external packages
  - the last one includes the testsuite description

Release Deployment Procedure 1/2

- Specific Internal Procedure Steps
  - the set of bugs and new features were defined and communicated via e-mail or on the IRC channel
  - when the development issues were resolved, an e-mail was sent to the work package mailing list, communicating the start time of the test session
    - before this time, developers have to commit all pending changes

Release Deployment Procedure 2/2

- Specific Internal Procedure Steps
  - when the start time was arrived, a CVS branch called *test_<version>* was created
  - software was rebuilt starting from the new CVS branch, and some tests were performed
    - including the execution of the work package specific regression test suite
  - when tests were satisfactory, the release was tagged on the branch and all the applied fixes are merged to the main trunck

A bit of sociology

- The role of the packager was introduced to help WMS working group
  - organizing the code tree structure
  - providing templates for the packaging of new components
  - overseeing on the uniformity of build procedures
  - providing all developers common formats for the Makefile.am files, M4 files, and configure.in file
  - allowing developers to concentrate just on code development

The EDG WMS Code Status

- Current version
  - makes use of the shown solution
- Prototype of new version – uses GNU autotools – has not committed yet
  - improves the description of the internal dependencies
    - re-writing the configure.in file
  - still uses M4 files for checking the external dependencies
    - removing obsolete checks
  - For the developers, configure output is more readable than it was before (IMPORTANT!)

Future Work

- autodep tool
  - is in the development phase
  - should be a new feature of GNU autotools
  - should simplify the configure.in file, making it simpler for reading, once the tool is terminated
  - should take into account complex interaction pattern
  - is able to express internal and external dependencies in a fashion similar to that of common autotools

Conclusion

- Summarized our experiences
- Described the encountered limits
- Added extensions to manage and package code accommodating the needs of the EDG WMS
- Successfully handled complex set of internal and external dependencies

Conclusion

- Requirement: code organization should be well-defined before code writing phase
  - Otherwise, it will be very difficult:
    - making developers accept architectural/structural changes on the fly
    - handling/managing dependencies between components
- Observation: a clear, well-designed initial model will allow to immediately identify major problems

Pointers

- The EDG WMS home page
  - http://infnforge.cnaf.infn.it/cgi-bin/cvsweb.cgi/workload/?cvsroot=workload
- CVS repository
  - http://infnforge.cnaf.infn.it/cgi-bin/cvsweb.cgi/workload/?cvsroot=workload
    - access: ssh, (anonymous) pserver
- Thanks to the WMS team and the European Datagrid project