LHC COMPUTING GRID

LCG 2 MIDDLEWARE DEVELOPERS GUIDE

Document identifier:	
EDMS id:	**n/a**
Version:	
Date:	**September 3, 2004**
Section:	**misc**
Document status:	**Final**
Author(s):	CERN GRID Deployment Group
File:	**LCG-Midleware-dev-guide**

Abstract: *This document is a guide for any developers developing or modifying software for LCG.*

CONTENTS

1. OBJECTIVES OF THIS DOCUMENT

This document is a guide for anyone developing or modifying code for LCG. This guide is directly derived from the European Datagrid Developer's Guide. The LCG version differs to suit the requirements of LCG and is more concise. The main objective of this guide is to define the procedures used by LCG for software development to ensure that the software produced meets quality required for a production system. This guide focuses on the basics in order for it to be easily followed, flexible and applicable to other projects. This guide should also be an example to anyone producing software for LCG demonstrates what is expected in matters of quality.

The guide covers the following areas;

- CVS structure and source code management.
- Required documentation.
- Software configuration.
- Software packaging.
- Software testing.
- Autobuild
- Submission for addition to LCG.

LCG developers have all had an opportunity to provide input for this guide and as such is it hoped that everyone will follow the guidelines. If there is anything in this guide that needs improving then it should submitted for updating. However, there must be a general consensus among the developers for it to be included in the guide.

2. PROCEDURE

The software development procedure can be broken down into a few simple steps.

- Create a new module in CVS.
- Write the code and documentation.
- Thoroughly test the code.
- Tag the CVS tree for the module.
- Ensure that Autobuild has successfully created the package.
- Thoroughly test the packages.
- Submit the autobuilt package to LCG.
- Fix any bugs found by the integration and certification process.

2.1. QUALITY CONTROL AND FEEDBACK

The initial quality control is with the developer. The developer should ensure that the software has the required working functionality. This is usually done with unit tests. Others test should be considered as well. The methods and results of these additional tests should be documented.

The second quality control is on the Autobuild machine. Packages will only be published in the Autobuild repository if the Autobuild is successful. The feedback to the developer is via email and the Autobuild results page.

The Final quality control is done by the LCG integration and certification process. If any problems are found, these should be fed back to to developer by using the Savannah bug tracking tool.

3. CVS AND SOURCE CODE MANAGEMENT

All source code is held in the CERN central CVS repository http://cvs.web.cern.ch/cvs/, in the project lcgware.

- CVSweb
  `http://isscvs.cern.ch:8180/cgi-bin/cvsweb.cgi/?cvsroot=lcgware`
- Read only access
  export CVSROOT=:pserver:anonymous@isscvs.cern.ch:/local/reps/lcgware
- Read/Write access
  . export CVSROOT=:ext:isscvs.cern.ch:/local/reps/lcgware
  . export CVS_RSH=ssh
  . Developpers have to have an afs account at cern.
  . Contact the CVS librarian (email) for account opening.

For more informations see CERN CVS documentation pages http://product-support.web.cern.ch/product-support/UI/Docs/CVS/.

3.1. MODULES, META-MODULES AND CVS DIRECTORIES

A module is a CVS directory that contains the source code for a given functionality. The name of the module should be in the format of *project-name-type*. Each module should create a package with the same name of the module eg. the module lcg-example-program should create the package lcg-example-program. As one module should create one package, the granularity of packaging should be considered when creating a module, see section on packaging.

A meta-module is a CVS directory that contains modules. If a number of similar modules exist then they should be grouped into a meta-module. eg. The meta-module lcg-example could contain the modules lcg-example-client lcg-example-server.

3.2. CVS STRUCTURE

The CVS directory content for modules and meta-modules should be as follows.

- The top-level CVS directory should contain only modules and meta-modules.
- A meta-module should contain only modules, a README file and a Makefile.
- A module should also contain such files plus the directories, src, etc, doc, build and test.

The files and directories mentioned above should be used in the following way.

- The README file, see the documentation section for details.
- Build files are required for Autobuild.
- The src directory contains all the source code for the module.
- The etc directory contains all the static configuration files, templates etc. for the module.
- The doc directory should contain all the documentation for the module. This should include any example configuration files.
- The test directory should contain all testing material including unit tests and descriptions of one-off tests that have been conducted. See the testing section for more details.
- The scratch areas used for compiling, testing and packaging should not be checked into CVS.

Table 1: Software Tree Layout

$LCG_LOCATION/bin	end user executables
$LCG_LOCATION/etc	machine independent configuration files (use *.conf extension)
$LCG_LOCATION/include	header files
$LCG_LOCATION/lib	sharable and static libraries (excl. java)
$LCG_LOCATION/libexec	support executables invoked by bin or sbin commands
$LCG_LOCATION/sbin	non-end user, system, or configuration executables
$LCG_LOCATION/sbin/test	test commands for packages
$LCG_LOCATION/share	shared architecture-independent files
$LCG_LOCATION/share/java	java libraries (*.jar, *.war, etc.)
$LCG_LOCATION/share/doc/name-version	LICENSE, README, and documentation
$LCG_LOCATION/share/man	man pages
$LCG_LOCATION/share/man/man1	User programs
$LCG_LOCATION/share/man/man2	Unix system calls
$LCG_LOCATION/share/man/man3	Programming libraries
$LCG_LOCATION/share/man/man4	Device node files
$LCG_LOCATION/share/man/man5	System configuration file formats
$LCG_LOCATION/share/man/man6	Games
$LCG_LOCATION/share/man/man7	Macro file formats
$LCG_LOCATION/share/man/man8	Daemons
$LCG_LOCATION/share/man/man9	Local system calls
$LCG_LOCATION/share/man/mann	Miscellaneous
$LCG_LOCATION/var	area for machine specific files
$LCG_LOCATION/var/etc	machine specific configuration files (use *.conf extension)
$LCG_LOCATION/var/etc/profile.d	machine specific profile scripts
$LCG_LOCATION/var/log	log files

3.3. VERSION NUMBERING

Version numbers, CVS tags should be in the format *version_revision_patch* e.g. lcg1_0_2. The version number should change only when there are non-backward compatible changes or when there are major changes in the features. The revision number change is for additional backward compatible changes. The patch number is for bugfixes. When a tag is made the Autobuild machine will automatically build the code and do some automatic testing.

3.4. INSTALL LOCATIONS

The table below show where installed files should be located. If new directories need to be created they should not be created at the top level. To avoid conflicts with other packages and to uniquely identify the documentation for a package, the documentation should be placed in the directory $LCG_LOCATION/share/doc/name with “name” being the package name. Below $LCG_LOCATION/share/man, the usual Unix conventions should be followed. $LCG_LOCATION is typically /opt/lcg, although this can not be assumed. Care must be taken to make the package entirely relocatable.

4. DOCUMENTATION

Every package should contain a README file. Other types of documentation will depend on what is in the package. The documentation should be in at least html and man page format.

4.1. THE README FILE

This file should be in plain text and should contain the following information. A link to to where the information can be found is adequate.

- The function of the module.
- A short history of recent modifications.
- The main download/information page for the software.
- The license of the software.
- Build and runtime dependencies.
- How to build and install.
- A configuration guide.
- How to get more informations, especially about usage.
- Known bugs and workarounds.
- Planned evolution (future of the software).
- Contact: who made the software and how to contact them.

4.2. OVERVIEW DOCUMENTATION

All large packages should contain an overview of what the package does and how it works.

4.3. API DOCUMENTATION

All APIs should have API documentation. This can easily be provided with Doxygen for C/C++ code, Javadoc for Java code and POD for Perl.

4.4. END USER DOCUMENTATION

Every program that is executed by either a user system administrator requires documentation, especially a man page. This is anything that goes in bin or sbin.

4.5. CONFIGURATION DOCUMENTATION

If a package requires configuration then documentation on how to do the configuration is required. All LCFG objects require a man page explaining the usage and the configuration parameters used.

4.6. TEST DOCUMENTATION

All testing that has been conducted on the module should be documented. This does not need to be distributed but should reside in the modules test directory in CVS. See testing section for more details

5. CONFIGURATION

To make the configuration easy for both manual installs, LCFGng and extensible to other configuration and installation tools, the following guidelines should be applied.

The software should be be configured by one configuration script and one configuration file. There is no reason why the configuration scripts can not create multiple configuration files and call helper scripts.

An example configuration file should be provided in the documentation.

For manual installation all that is required is to copy the example configuration file to the correct location. Edit it with the required values and run the configuration script.

The LCFG object should use a template and the values in the machine profile to create the configuration file and then execute the configuration script.

6. TESTING

Everything to do with testing of a module should be contained in the test directory for that module.

Every module should have a unit testing suite that can be run from the compiled code. This test suite could be automatically run by the autobuild machine after it has compiled the code. The coverage of the unit tests and limitations should be described in a text file within the test directory.

Any one-off test such as stress tests and performance tests should be documented with results, in such a way that someone else could repeat the test if so desired.

Another testing suite should test the module once installed. This could be packaged but is not necessary. Also instructions of how someone can test the package once installed should also be contained in a text file within the test directory.

7. SOFTWARE PACKAGING

This section describes how to ensure packages are consistent with the installation tools and do not conflict with other packages.

7.1. GRANULARITY OF PACKAGING

A package is a single file that contains the software product or some subset of it for distribution. If the guide in the section on CVS management has been followed then the granularity of packaging should already have been decided. The following should be considered when deciding the granularity of packaging.

- The packages must be large enough that no cyclic dependencies exist between them.
- The packages should be split to separate code likely to be deployed on separate machines, e.g. client and server code.
- There must be no conflicts between the packages, e.g. a file in the same location in two packages.
- Packages should be small to allow individual components to be independently upgraded.
- Public interfaces should be in separate packages and versioned independently.
- Code intended only for developers should be split into a separate package.

7.2. PACKAGE DEPENDENCIES

If the packaging method contains dependency management, this should be used. The build and runtime dependencies should already be listed in either the README file and or in the documentation.

7.3. SPECIFIC PACKAGE FORMATS

Names of packages should correspond to the CVS module name. Also the version of x.y.z should correspond exactly to a CVS module tag of lcgx_y_z. If packages are released for different compiler versions, an extension should be added to the package name in order to retain this information in the name. The current acknowledged extension is the string _gcc3_2_2 for gcc3.2.2. To help giving the appropriate name to a package, the Autobuild system defines the variable LCG_RPM_RELEASE_VERSION to either the empty string or _gcc3_2_2. Developers are encouraged to rely on this variable in order to set the appropriate name.

RPM Packages

For more information about RPMs see: http://www.rpm.org

The RPM spec file should be located in the etc CVS directory for the module. The following points should be considered when building RPMS.

- RPM building should be called using rpmbuild commad not rpm -b.
- All of the fields in the spec file should be completely and correctly provided.
- RPM packages allow a release number in addition to the version. As the RPM spec file is in CVS, each change in the package will involve at least a patch level revision. Consequently, the release number will always be 1.

- Any RPM packages built from the LCG fork of the EDG code will require the release number to be lcg1. This will ensure that there are no conflicts with version numbers and make the source repository easy to locate.
- The RPM format allows a hierarchical categorization of packages. For LCG software set group name in spec file to grid/lcg.
- RPM is continually evolving and adds convenient macros. Unfortunately using them makes the package incompatible with earlier versions of RPM. Therefore, avoid fancy features of the latest and greatest RPM including macros such as %makeinstall (distribution dependent), and features of rpmlib (version dependent).
- Configuration varies greatly from platform to platform and should be clearly separated from the package itself. Consequently, RPM installation scripts like %post, %pre, for configuration should be avoided. However, it is ok to replace path names resulting from the build procedure.
- The Autobuild machine builds and installs the module using the Makefile. The only line required for the build section in the RPM spec file should be as follows.

```
\%build
make install install-doc prefix=\%{buildroot}\%{prefix}
```

- To enhance the portability of the package, do not disable RPM's internal "provide" mechanism. This means for example that the following should *not* be in the spec file:

```
AutoReqProv:    no
```

- Correctly specifying the dependencies simplifies installation. The distinction between build and run-time dependencies should be made. The "BuildRequires" tag specifies build dependencies and the "Requires" tag specifies run-time dependencies. Every required package not automatically found should be manually added.

```
Requires:    lcg-example-client
```

- All RPMs should be buildable by a non-root user. The BuildRoot feature should be used and following lines should be in the specfile.

```
%files
%defattr(-,root,root)
```

8. AUTOMATIC BUILD SYSTEM

The Autobuild system is used to automatically build code that is in the LCG CVS repository. The autobuild system will build the latest tag of the format lcgx_y_z for each module. If the build is successful, and any tests on the packages have passed, the generated packages will be published in the autobuild repository. If the package fails to build or does not pass the tests, the package will not be published and an email will be sent to the package owner.

The result are shown on the Autobuild result Pages:
http://lxshare0297.cern.ch/LCG/autobuild/rh7.3-lcg/.

8.1. AUTOTOOLS

Modules managed by Autotools must be able to create the configure script from a CVS checkout with the following command: ./autogen.sh. The new name is more compliant with usages in the open source community.

The autogen.sh file resides at the root of the module and contains the necessary first steps for Autotools to run, for example:

```
#!/bin/sh
set -x
aclocal -I config
autoheader
libtoolize --automake
autoconf
```

The generated configure script is then run to generate the Makefile.

8.2. MAKEFILE

The top-level Makefile must contain the required targets and perform the required tasks.

all Default target, build the package (create executable and libraries).

install Install the software (default location must be $EDG_LOCATION).

dist Create a source tar.gz archive using the naming convention described in Section **??**.

binary-dist *(optional)* Create a binary tar.gz archive using the naming convention described in Section **??**.

rpm Generate RPM packages. Built RPMs must be found under a RedHat directory tree analogous to the one in /usr/src/redhat. The _topdir should either be left undefined (managed by the autobuild program), set to the build directory or set under rpm in this directory. Autobuild will look in the following places for generated rpms:

- If rpm directory[1] exists, then rpm/RPMS/¡arch¿/ and rpm/SRPMS/.
- Else, if RPMS directory exists, then RPMS/¡arch¿/ and SRPMS/.
- Otherwise, the actual rpm `_topdir` location will be used as base path.

apidoc Generate developer documentation (using Doxygen for C/C++ or JavaDoc for Java). This documentation should be found in doc/apidoc directory.

userdoc Generate user documentation. It should be found in doc/userdoc directory.

[1] All paths are relative to the module's root

check Some automatic checking method will be implemented, but several levels of checking may be needed that will not be addressed by a single make target. This item has yet to be developed further.

All targets which succeed must return zero to the caller, all targets which fail must return a non-zero value.

Invoking any of the required targets must never result in an interactive prompt. The build procedure is an automatic process which has no ability to respond to the prompt and hangs as a result.

8.3. ANT

Some Java modules use ant as a building tool. Ant targets should be the same as those described for Makefile, except that the "all" target is named "compile" with this tool.

All targets which succeed must return zero to the caller, all targets which fail must return a non-zero value.

Invoking any of the required targets must never result in an interactive prompt. The build procedure is an automatic process which has no ability to respond to the prompt and hangs as a result.

8.4. BUILD ON DEMAND

The Build on demand system allows a build to be performed immediately on request.

To trigger the BOD system, a special file, called bod_dummy that should reside at the top-level of the module has to be tagged[2] by:

- A regular release tag (like `lcg1_0_2`). In this case, the built RPM will immediately be published.
- A special "build" tag of the form `build[0-9]*`, that is the string `build` followed by any arbitrary number. The produced RPMs will not be published in that case.

It's very important to tag the bod_dummy file only with an already existing tag, so for a new tag, it's better to first tag every file except bod_dummy and then tag that file to trigger BOD. The reason is that there is a race between effective tagging of every files, and checkout of this tag for BOD operation (checkout of course fails if the file isn't yet tagged).

The build results and logs are published on a dedicated web page, follow the links from the autobuild web page (under "Build On Demand page").

A Build on Demand request on a module following another one on the same module will only be processed if more than ten minutes elapsed between the two orders.

As BOD orders cannot be done in parallel, they are queued, so the time between the request and the processing can be arbitrarily long.

[2] Any tag operation will do (add, delete, etc). To keep building a version several time, a succession of cvs tag lcg1_0_0 and cvs tag -d lcg1_0_0 would do.

9. LCG SUBMISSION

The LCG package submission page is used to submit packages to LCG for consideration for inclusion in a release. This system is for submission of all new LCG software and external rpms. It is useful for tracking the evolution of the LCG source code and middleware. All packages should be submitted to LCG via this page. This includes LCG packages, EDG packages and external packages.

The submissions page can be found at the following URL.
https://savannah.cern.ch/patch/?func=additem&group=lcgoperation
Note: You have to create an account and log in on the system before using it.
The submission page comprises of a web form with the following fields.

- Name (package/component/archive):
- Revision Tag:
- Names of binary packages:
- Names of source packages:
- Priority:
- Assigned to:
- Originator Name:
- Originator Email:
- Component Version:
- Place to find packages:
- Dependencies:
- Summary:
- Original Submission:

When this form is submitted, it is automatically checked to ensure that all the required fields have been filled. Each new package is assigned a status to show where it is in the test phase. The management system will automatically inform the people concerned. Any problems found with the integration of this new package will be entered into the relevant section in Savannah.