EGEE-III

DEVELOPERS' GUIDE

MSA3.6

Document identifier:	EGEE-III-MSA3.6-979788-v-4.pdf
Date:	**24/03/09**
Activity:	SA3
Document status:	**APPROVED**
Author:	**Andreas Unterkircher & Elisabetta Molinari**
Document link:	https://edms.cern.ch/document/979788/1

Abstract:

This document gives a middleware developer all the necessary information needed in order to interact with the Integration, Testing and Release Process.

Table of Contents

gLite Developer's Guide

Applicable to gLite as of version 3.1

February 2, 2008

1.Introduction

1.1 Purpose

This guide is for anyone developing or modifying EGEE Middleware. Aim of the document is to provide developers with a quick reference guide to quickly become familiar with development tools, software integration process and release management. It does not contain material about coding standards and naming schemas. The interested reader is advised to consult the EGEE II Developer Guide [R2] where these topics have been treated.

1.2 Document Organization

This guide should be considered as a live document. The guide is written in a wiki to allow faster and easier changes, contributions and document management. Twiki2Pdf Add−On will be used to provide a pdf version as required for official documents by EGEE Document Management Procedure [R1]. The document contains two appendices with technical information about the usage of environment variables, daemons and services in gLite as well as a statements on the license used by gLite and IPV6 compliance.

The URL for this document is https://twiki.cern.ch/twiki/bin/view/EGEE/DevelopersGuide.

1.3 Application area

This document applies to the all software process, including development, integration and testing activities, seen from the point of view of developers.

In several sections ETICS [R7] command line commands are used. At the time of writing this document the new ETICS client was not yet available. Thus ETICS commands given in this document are subject to change.

1.4 References

R1	EGEE Document Management Procedure http://project−egee−iii−na1−qa.web.cern.ch/project−egee−iii−na1−qa/EGEE−III/Procedures/DocManagmtProcedure/DocMngmt.htm
R2	EGEE II Developer Guide https://edms.cern.ch/document/790125/0.4
R3	EDG Developers Guide https://edms.cern.ch/document/358824
R4	Central CVS Service − HOW TO http://cvs.web.cern.ch/cvs/howto.php
R5	CERN Account https://cernaccount.web.cern.ch/cernaccount/Default.aspx
R6	Wiki home of ETICS: The Grid Quality Process https://twiki.cern.ch/twiki/bin/view/ETICS/WebHome
R7	ETICS Web Page http://etics.web.cern.ch/etics/
R8	ETICS User Guide https://edms.cern.ch/file/795312//ETICS−User_Manual−latest.pdf
R9	Recommendations (and a few rules) for logging in JRA1 Code https://twiki.cern.ch/twiki/pub/EGEE/EGEEgLite/logging.html
R10	A Proposal To Standardize Configuration, Logging and Error Handling https://edms.cern.ch/document/486630
R11	Software License https://twiki.cern.ch/twiki/bin/view/EGEE/EGEEgLiteSoftwareLicense

R12	How to test IPV6 compliance https://edms.cern.ch/document/930868/1
R13	MSA3.4.1 Definition and documentation of the revised software life−cycle process https://edms.cern.ch/document/973115
R14	Guidelines on how to fill a patch https://twiki.cern.ch/twiki/bin/view/EGEE/HowToFillAPatch
R15	gLite Middleware Savannah group https://savannah.cern.ch/projects/jra1mdw/
R16	Service certification checklist https://twiki.cern.ch/twiki/bin/view/EGEE/SA3testing
R17	Savannah Command Line Interface https://twiki.cern.ch/twiki/bin/view/EGEE/SavannahCommandLineInterface
R18	Documentation for accessing CERN CVS http://product−support.web.cern.ch/product−support/UI/Docs/CVS/howto.html
R19	IPV6 compliant code http://www.euchinagrid.org/docs/EUChinaGRID−Del2.2v3−1.pdf
R20	Pre Production Service http://egee−pre−production−service.web.cern.ch/egee−pre−production−service/

1.5 Feedback

For feedback or questions on this guide please contact Andreas Unterkircher at CERN. His contact details can be found in the CERN directory http://consult.cern.ch/xwho.

2. Software Configuration and Version Control

2.1 Overview

CVS (Concurrent Versions System (CVS), also known as Concurrent Versioning System) and ETICS (eInfrastructure for Testing, Integration and Configuration of Software) are used to manage software complexity and to improve the software quality. CVS is used to allow several developers to collaborate on a software module development in a distributed (in space and time) environment. CVS also keeps track of all work and all changes in the source code. ETICS is used to

- build software modules managing complex dependencies and software integration with configuration concept
- build software modules on different platforms
- manage software configuration versioning
- manage software release process

2.2 Repository

The gLite CVS repository is hosted by the CERN Central CVS Service [R4]. It can be accessed via the generic `CVSROOT`:

```
glite.cvs.cern.ch:/cvs/glite
```

For up to date information on how to access the CERN CVS have a look at the CERN Central CVS Service [R4] webpage.

2.3 Read Only Access gLite CVS Repository

The CVS repository can be accessed, in read only mode, two ways:

- Direct access with http://glite.cvs.cern.ch:8180/
- with anonymous connection

```
export CVSROOT=:pserver:anonymous@glite.cvs.cern.ch:/cvs/glite
```

The full documentation for accessing CERN CVS can be found on the CERN CVS page [R18].

2.4 Authentication: Read/Write Access

In order to have CVS write access, it is necessary to have a CERN AFS account. Instructions to request an account are available on the CERN account webpage [R5].

2.4.1 Read/Write Access using Kerberos V

```
export CVSROOT=':gserver:glite.cvs.cern.ch:/cvs/glite'
```

2.4.2 Read/Write Access using SSH

```
export CVSROOT=:ext:<afs-user-name>@glite.cvs.cern.ch:/cvs/glite
export CVS_RSH=ssh
```

To set up SSH, follow the instructions for Configuring SSH access from Linux/Unix on the CERN CVS page.

2.5 Components, Subsystems and CVS Directories

A component is a CVS directory that contains the source code for a given functionality and a subsystem is a logical group of components. The name of CVS directory for a component should be in the form of *org.glite.subsystem.component*. Dots should not be used in the subsystem or component name. Each component should create a package where the package name is in the format *glite−subsystem−component*. Anyone with write access to CVS should be able to create a components directory in CVS. After creating this directory you will need to obtain write access by emailing a request to middleware−integration. The CVS directory for the component should contain the following directories.

```
src: contains the source code
doc: contains the documentation
etc: configuration files
tests: contains test material
build: not need in CVS but should be use for building the code.
```

2.6 CVS tags

When a component is ready to be released, the CVS componet directory should be tagged. The format of the tag should be *glite−subsystem−component_R_x_y_z_w* where x, y, z and w are the major, minor, revision and age numbers, respectively, for the release.

3. The build system

ETICS [R7] is the build system used by EGEE. The main goal of ETICS is to handle every aspect of the software development workflow, i.e. build, release and testing, in such a way that every operation has to be completely independent from the platforms, the languages and the tools involved in the process. The content of an ETICS site is structured as a set of hierarchies of entities, with one hierarchy for each project registered into the service; an entity can be a subsystem of the project or a component of a given subsystem. Every entity groups different configurations; a configuration contains all the information and instructions required for building, packaging and testing the software on one or more platforms. In the following it is referred to the set of information and instructions of a configuration as metadata. All gLite components and subsystems should be in the project gLite Middleware project. Interaction with ETICS can be done via the web interface [R7] or command line tools. For more in−depth details and tutorials on ETICS please see the Etics documentation [R6].

Note that it is difficult to build the middleware without the use of ETICS.

3.1 ETICS and user permissions

Any user must register to ETICS in order to have the correct set of permissions granted to its role. The registration can be performed at the following page:
https://etics.cern.ch:8443/eticsAdmin/public/registration/requestRegistration.jsp

Since the authentication mechanism of ETICS is based upon X509 public key infrastructure (PKI) before sending the request for registration the user must be sure that his credentials, the private key and the related certificate, have been correctly installed into the web browser. ETICS grants access even to unregistered users, providing a special "guest" account in this case; the system allows those users to perform just read−only operations.

ETICS provides a fine−grained permission management; rights can be set up for the entire project or for each subsystem or component and also for a given configuration of subsystem or component. For further details about the permission management refer to: Understanding Security in ETICS

3.2 Installation, configuration and set up of the ETICS command line client

The steps required for the installation of the ETICS client are:

Download the Etics workspace setup script

```
wget
"http://eticssoft.web.cern.ch/eticssoft/repository/etics-client-setup.py
-O etics-client-setup
```

Run the Etics workspace setup script

```
python etics-client-setup
```

Set the ETICS_HOME env variable

```
export ETICS_HOME=`pwd`/etics
```

Add the Etics commands to your PATH

```
export PATH=$PATH:$ETICS_HOME/bin
```

All the configuration parameters for the ETICS client are defined into a configuration file whose default location is $HOME/.etics.conf. Most of the parameters listed into the configuration file can be also redefined in the command line. The basic properties for the account setting are:

the path of the user certificate

```
x509_user_cert
```

the path of the user private key

```
x509_user_key
```

they are required in order to establish the secure connection and to authenticate against the ETICS server and are contained into the *user* section. The required parameter for the connection is *server*, which is the fully qualified domain name of the service to contact and is contained into the *system* section, the main service running is located at *etics.cern.ch*. The ETICS client is usually able to detect the correct platform of the system where it is running on, whever this is not possible the user can force the platform definition with the *platform* property listed into the generic *properties* section. Every ETICS command can work only inside a well formatted workspace, i.e. a directory of the local filesystem which contains the project metadata files. The commands that must be invoked for setting up the workspace are:

for the initialization

```
etics-workspace-setup
```

for downloading the project metadata

```
etics-get-project org.glite
```

3.3 Creation of an ETICS entity

The creation of a subsystem is carried out by the project administrator; the creation of a component is carried out by the project administrator or by the module administrator of the subsystem that component belongs to. The suitable command for this operation is *etics−module*:

add a subsystem to the project

```
etics-module add --parent org.glite --subsystem subsystem
```

add a component to a subsystem

```
etics-module add --parent subsystem --component component
```

By default the system creates a configuration for both the subsystem and the component whose name is *org.glite.subsystem.HEAD* or *org.glite.subsystem.component.HEAD* respectively.

The naming convention for subsystem and components should follow the same rules defined for the CVS module name, discussed in section 2

the subsystem name should be

```
org.glite.subsystem_name
```

the component name should be

```
org.glite.subsystem_name.component_name
```

There's no relation between the name of the entity and the name of the CVS module whom the entity refers to.

Every operation on a subsystem or component configuration can be performed using the command *etics−configuration*:

add a configuration to a component

```
etics-configuration add -c component_config -i input_file component
```

modify metadata of a configuration

```
etics-configuration modify -c component_config -i input_file
component
```

clone a configuration

```
etics-configuration clone -c target-configuration
source-configuration component
```

list the existing configuration for a component.

```
etics-list-configuration component
```

Most of the commands described above requires that a set of metadata must be defined into a file. This file, called in the following *"ini−file"*, is divided into sections and must contain

- the main definition for a configuration:

```
[Configuration-glite-mysubsystem-mycomponent_R_1_0_0_0]
moduleName = org.glite.mysubsystem.mycomponent
displayName = Brief description of mycomponent
description = Full description of mycomponent
projectName = org.glite
age = 0
tag = glite-mysubsystem-mycomponent_R_1_0_0_0
version = 1.0.0
path = ${projectName}/${moduleName}/${version}/noarch/
```

- the section that defines the CVS commands for the *default* platform:

```
[Platform-default:VcsCommand]
displayName = Brief description of the CVS commands of mycomponent
description = Full description of the CVS commands of mycomponent
tag = cvs -d ${vcsroot} tag -R ${tag} ${moduleName}
checkout = cvs -d ${vcsroot} co -r ${tag} -P ${moduleName}
```

- the section that defines the build commands for the *default* platform:

```
[Platform-default:BuildCommand]
postpublish = None
packaging = None
displayName = Brief description of the build commands of mycomponent
description = Full description of the build commands of mycomponent
doc = None
prepublish = None
publish = None
compile = None
init = None
install = None
clean = None
test = None
configure = None
```

for further details concerning the syntax of an ini−file refer to the ETICS user guide [R8].

3.4 Dependencies for an ETICS entity and the lock operation

One powerful feature of ETICS is the capability of handling the software dependencies, such as libraries and tools, required by a given subsystem or component with a minimum amount of effort from integrators and developers. Any dependency or resource must be registered in the system; in this way any entity can be accessed by a project, any possible combination is allowed, a subsystem can be a dependency for a project or a subsystem contained into it, a component of a project can be a dependency for another component of a different project.

The most important example concerns the project *externals* which collects all the resources published by external projects. It is strongly recommended to use only the external resource contained into that project in order to exploit all the features of the system during the release step.

According to the way an ETICS entity uses a given dependency the system distinguishes two different criterions:

- *static* or *dynamic* dependency: the former defines a fixed release version for the resource inside an entity configuration, the latter allows the entity to inherit the release version from the parent entities; they're mutually exclusive.
- *build−time* or *run−time* dependency: the former specifies that the resource is required only during the build or testing step, the latter instructs the system in order to have a dependency when the artifact produced is used, typically putting such requirement into the package; the two definitions can be combined together.

When an entity defines a dynamic dependency it delegates the system to select the correct version of the resource, in general the version decided for the entire project. It is strongly recommended to make use of this feature, especially for the external dependencies, if no particular requirements must be considered.

Dynamic dependencies are not able to guarantee that a given configuration for an entity can be reproduced at any time. It is necessary to freeze, or lock, the configuration of that entity together with all the configurations of the sub−entities and dependencies. The *lock* operation is performed with the command *etics−configuration lock*, as in the following example:

```
etics-configuration lock --parent-config glite_branch_3_1_0     -c component_configuration compon
```

3.5 Checking out and building a component

The steps required for downloading and building the code of a component are:

Check out a specific configuration

```
etics-checkout --config component_configuration_name
--project-config project_configuration_name component_name
```

Locally build the code

```
etics-build component_name
```

Integrators and developers can schedule a remote build for the configuration of the managed component with the following command

```
etics-submit build --config component_configuration_name --platforms
slc4_ia32_gcc346,slc4_x86_64_gcc346,sl5_ia32_gcc412,sl5_x86_64_gcc412
component_name
```

Also a release build against a locked component can be scheduled

```
etics-submit build --config locked_component_configuration_name
--platforms slc4_ia32_gcc346,slc4_x86_64_gcc346 --register
component_name
```

The system can publish the artifacts, such as tarballs or packages (rpm, deb), either into a temporary area, called *"volatile"* repository, or into the permanent repository. The volatile repository is suitable only for development purposes, the permanent repository must be used for publishing the official artifacts of a release and it can be populated only if the configuration is locked. The results and every reference to artifacts produced by the remote build are contained into the ETICS reports site [R7].

All the operations above can be applied to an entire subsystem replacing the component name and the configuration with subsystem name and the corresponding configuration.

3.6 ETICS and the software release management

For each subsystem and component registered in ETICS it is possible to distinguish two different types of configuration:

- the ETICS configuration for a branch
- the ETICS configuration for a release

The former is suitable for development purposes, it contains references to CVS branches or to the HEAD main branch and cannot be locked as any definition can be changed according to the development requirements. The naming convention for a branch configuration should be as in the following:

for a subsystem

```
glite-subsystem_branch_major_minor_revision
```

for a component

```
glite-subsystem-component_branch_major_minor_revision
```

The latter represents a frozen release thus it must be guaranteed it cannot be changed; it contains only references to CVS tag and must be locked in order to freeze the entire dependency structure and the set of commands, properties and variables. The naming convention for a frozen release configuration should be the same as the CVS tag, thus it should follow the notation:

for a subsystem

```
glite-subsystem_R_major_minor_revision_age
```

for a component

```
glite-subsystem-component_R_major_minor_revision_age
```

If the configuration refers to an internal release, suitable for tests only purpose for example, any notation is allowed but it is strongly recommended to put in the tag the name of the component or the subsystem together with a string summarizing the goal of the release. The naming convention for a branch configuration states that the configuration name should contain the prefix

for a subsystem

```
glite-subsystem_branch
```

for a component

```
glite-subsystem-component_branch
```

followed by a string that identifies the branch. The only exception allowed is the HEAD main branch for which the ETICS provides the default configuration name *org.glite.subsystem.HEAD* or *org.glite.subsystem.component.HEAD*.

A release configuration can be obtained from a branch configuration with the ETICS clone operation of the web application or taking the ini−files of the branch configuration as templates for the etics−tag command of the command line client.

4. Release Workflow

4.1 Understanding the release

Using ETICS as a build system, the release is approximated in ETICS by using a project configuration. Subsystem configurations are used to select the version of the components which should be in the release. Currently this involves managing subsystem configurations which creates and extra layer of abstraction however it would be advantages if the components could be managed individually. The environment is generally specified per platform and usually refers to external components. These are set by specifying the .DEFAULT property for each platform in the project configuration.

In this section we talk about "packages" meaning the software package format used on the current platform, e.g. for SL5 with "package" we mean "rpm".

4.2 Project configurations

The release team maintains three project configurations:

- **glite_branch_3_1_0**
- **glite_branch_3_1_0_cert**
- **glite_branch_3_1_0_dev**

The configuration **glite_branch_3_1_0** contains the latest configurations released to production. As we allow different versions of the same subsystem/component to be deployed at the same time on different node types **glite_branch_3_1_0** can only be an approximation to the situation in production. If two versions of one subsystem/component are released simultaneously the newer one is being added to **glite_branch_3_1_0**. Building against **glite_branch_3_1_0** is conceptually close to installing an up to date build machine with production packages and then building your sources there. After a release to production **glite_branch_3_1_0** is being cloned to **glite_branch_3_1_0_backup_yyyy_mm_dd**. The cloned configuration is being locked. Then

glite_branch_3_1_0 is updated with the latest configurations released to production.

Furthermore the release team maintains a configuration **glite_branch_3_1_0_cert** that contains the configurations of recently certified patches (patches in status "certified" and patches that are in PPS) that are not yet released to production.

The configuration **glite_branch_3_1_0_dev** is maintained to hold the latest tags of interest to developers, i.e. the configurations currently 'in work' and which are likely to subsequently be locked and used for a release candidate build. Any build related bugs can be considered fixed in this build. This build will not be used to create releasable software but provides a build of the latest tags against new versions of dependencies, enabling anticipation of problems. This build will not be used to create releasable software but provides an automatic build of the latest tags against new versions of dependencies, enabling anticipation of problems. Note that the tags contained in **glite_branch_3_1_0_dev** will be used to create releasable software, when locked and built against **glite_branch_3_1_0**. Developers can always build any configuration against **glite_branch_3_1_0** in the usual way.

When a release is required, the 'in work' configuration in **glite_branch_3_1_0_dev** should be locked and built against **glite_branch_3_1_0**. The next configuration to be 'in work' should be communicated to the integration team so it can be added to **glite_branch_3_1_0_dev**. When the packages are certified and released, the appropriate configuration will be transferred by the integration team to **glite_branch_3_1_0**.

4.3 Integration scenarios

Both ETICS and CVS are arranged on a subsystem basis. To work best, a subsystem should

- Group together components with synchronised evolution, where updates are frequently correlated.
- Contain no other unrelated components, as this imposes an artificial connection and prevents independent evolution

Unfortunately, the gLite subsystems as currently defined often break both these rules, as for historical reasons they are arranged along administrative lines. Nevertheless the scenario where updates of a component only affects the other components in the same subsystem can be regarded as the default scenario for gLite integration. Experience shows that most updates fall into this category. However there are exceptions to this. For example, during the development work on a component A it turns out that (updated) dependencies not present in **glite_branch_3_1_0** are needed. At the same time development work on another component B (in a different subsystem than component A) can introduce updated dependencies incompatible with component A. Currently such conflicts cannot be resolved in an automated way. Ultimately the bi−weekly EMT phone conference involving representatives from JRA1, SA3 and ETICS is the place where such conflicts are discussed and resolved.

Developer builds

This is the mechanism by which a developer or subsystem administrator can produce releasable components. A release candidate build is performed in a controlled environment (i.e. in a remote build), its products cannot be changed and its build log is archived with the artifacts. Deployment testing should be done at this stage so that only installable packages reach certification.

The standard use case

The following diagram illustrates the use case of a single component being affected. It can be extended to the case of several components within the same subsystem.

Action	Command

Check out latest configuration with glite_branch_3_1_0	**etics-checkout --project-config glite_branch_3_1_0 -c <config> <component>**
Commit changes	**cvs commit**
Tag changes	**cvs tag**
Create new local configuration using new CVS tag	**etics-configuration prepare -c <config> <component>** **vi Configuration.ini** **etics-configuration add -noautocommit -i Configuration.ini -c <new config> <component>**
Build the new configuration locally	**etics-checkout --local -c <new config> <component>** **etics-build -c <new config> <component>**
Test provisional build	
Create release tag	**cvs tag**
Point ETICS to release tag	**vi Configuration.ini** **etics-configuration modify --noautocommit -i Configuration.ini -c <new config> <component>**
Commit ETICS configuration	**etics-commit**
Lock configuration	**etics-configuration lock --parent-config glite_branch_3_1_0 -c <new config> <component>**
Registered 'Release Candidate' build	**etics-submit build --project org.glite --project-config glite_branch_3_1_0 --register --platforms < platform(s) name> --versioneddeps -c <new config> <component>**
Do a deployment test in ETICS	
Create and lock a subsystem configuration containing the new component(s)	to be done by the subsystem integrator

A deployment test will be provided in ETICS (at the time of writing this document such a test was not yet available in ETICS). For a given set of packages and gLite node types it first installs the production version of the node type and then updates it with the given packages. The test has to be done for every node type affected by the packages.

Note that even for the case of updating a single component a new subsystem configuration is needed as this currently is the only way to express in **glite_branch_3_1_0** the fact that a certain component is in production.

Other use cases

Developers may face the situation where they work on a component that depends on other components that are not yet part of **glite_branch_3_1_0**. A reason might be that these components are already certified but not yet released to production. To cope with this situation the release team maintains the configuration **glite_branch_3_1_0_cert** which contains the configurations of recently certified patches (patches in status "certified" and patches that are in PPS) that are not yet released to production. Instead of building against **glite_branch_3_1_0** a developer can use **glite_branch_3_1_0_cert**.

A developer can also request to get a clone of **glite_branch_3_1_0** named after a patch in Savannah, e.g. **glite_branch_3_1_0_patch1234**. The patch may stay in status "With Provider" during the whole development phase and is ultimately used to release the new components. Working with a clone of

glite_branch_3_1_0 is useful when working on several, interdependent components. However when the development work approaches "Release Candidate" status potential conflicts with other components in production have to be solved. The forum to do this is the bi−weekly EMT phone conference resp. the EMT mailing list. When the patch is finally released to production the corresponding project configuration will be deleted.

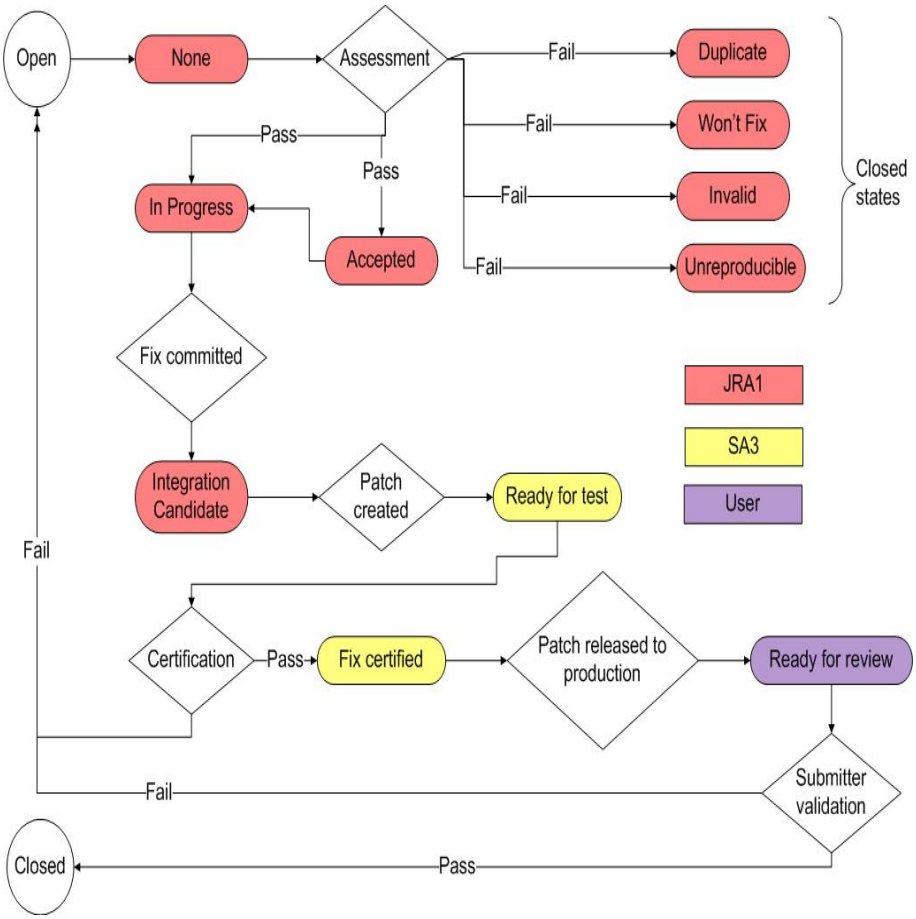
4.3 How to handle bugs

Bugs are handled via Savannah [R15] using the bug tracker.

When submitting a new bug to Savannah the "Category" field determines the list of persons getting notified about the bug. This list includes the coordinators of the Category as well as other developers associated with it. Ultimately the coordinator should ensure that the bug is being assigned to a developer. Alternatively the bug can be already assigned to a developer by the person submitting the bug. Taking up the bug the developer should decide if the bug can be accepted or if it has to be closed immediately. Justification for immediate closure are: Duplicate, Won't Fix, Invalid and Unreproducible.

If the bug is being accepted it should be put in status "Accepted". When development work for the bug starts the bug should go in status "In Progress". The bug may go directly into "In Progress", bypassing "Accepted", if the developer starts to work on it immediately.

After a fix for the bug has been committed the bug should go to "Integration Candidate" awaiting attachment to a patch. When a corresponding patch (with the bug attached) finally has been created the bug goes to "Ready for test". Now it's the certifier's task to assess if the bug has indeed been fixed. If the bug does not pass certification it is being set back to "None". In this case the bug may be detached from the patch or the patch might be rejected and cloned. If the bug passes certification it is being set to "Fix certified". Finally the patch is being released to production and the patch then gets set to "Ready for review". Now it's the submitter's task to validate that the bug is fixed in production. If it turns out that the bug is still present the submitter sets it back to status "None", otherwise to status "Closed". The time window for the submitter to check if the bug has been fixed is one month. After this period of time, if no complaints have been received, the bug will be set to "Closed" automatically.



Bug states diagram

4.4 How to handle patches

Patches are handled via Savannah [R15] using the patch tracker.

Patches are our way to apply changes to the gLite release in production. With changes we mean bug fixes as well as new services, new supported platforms and new features. When submitting a patch the different fields in the Savannah patch tracker have to be filled according to the "How to fill a patch" guidelines [R14].

Initially the patch is in state "With Provider" meaning that the developers are editing the fields and adding bugs. When this work is finished the patch should be put into "Ready for Integration". At this stage the integration team will apply a set of acceptance criteria. If they are not fulfilled the patch is being set back to "With Provider" waiting for the developers to fix the outstanding issues. The acceptance criteria are as follows:

- All fields have been filled according to the "How to fill a patch" guidelines [R14].
- An ETICS deployment test on all affected production node types has been run with the packages listed in the patch.
- The requirements on documentation as stated in section 3.5.1 "Before certification" in the document Definition and documentation of the revised software life−cycle process [R13] are fulfilled. Note that in the most frequent case of an update of an already existing node type only the updates to the documentation (if there are any updates) must be given in the patch. However for new node types documentation has to be provided accordingly.

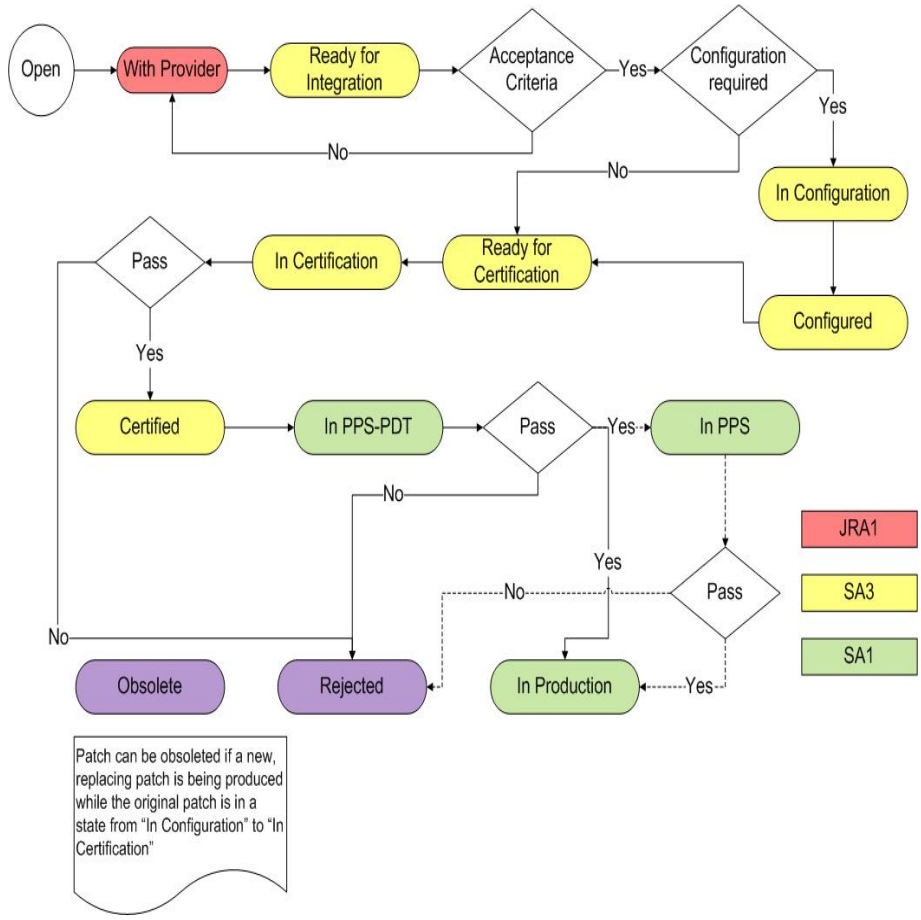
If the acceptance criteria are fulfilled and no configuration changes are necessary the patch is set to "Ready for Certification" meaning that the integration team produced the repository for this patch. If the patch needs

configuration changes it is first set to "In configuration". At this point the configuration coordinator has to provide the required configuration changes (typically updates to yaim). If they are in place the patch gets to "Configured" and subsequently to "Ready for Certification". Certification may start now, indicated by the state "In Certification". The patch might be rejected due to the following criteria:

- Some of the packages in the patch have to be changed as they break existing functionality. The functionality is being checked with the tests explained in SA3Testing [R16].
- Some important attached bugs are not fixed by the patch.

Note that if bugs are detached from the patch this does not necessarily lead to a rejection of the patch. If a patch gets rejected a clone is being produced, i.e. a new patch with the same content (including attached bugs and comments). This frees the developer from the burden of re−filling all the fields. Only new information has to be added (new packages etc.). The cloned patch starts again from status "With Provider". Patch cloning can be done easily with the SavannahCommandLineInterface [R17]. The clone should be produced by the original submitter of the patch as the clone has as "submitter" the user who executed the cloning tool.

Patches may depend on other patches for certification. If this is necessary patch dependencies have to be added in Savannah. We note however that patch dependencies should be avoided. Dependencies between patches make the certification process more difficult and may ultimately delay the release to production.



Patch states diagram

Abbreviations:

- PPS: Pre Production Service [R20]
- PPS−PDT: Pre Production Service Pre Deployment Test

Appendix A. Environment

This chapter describes the characteristics of the environment in which gLite daemons and services will run. This comprises the environmental variables the services have to use, guidelines for running and configuring daemons and services, and resource usage. As much as possible the standard locations for configuration files, start−up scripts etc. should be used as well as the standard methods to start/stop and configure services. Adherence to for example the unix/linux standards described in http://www.pathname.com/fhs/ and http://www.linuxbase.org/, simplifies the deployment of individual components and subsystems outside the gLite scope. Much of the contents of this chapter is taken from the EDG Developers Guide [R3].

A.1 Enviromental variables

It must be possible to install and configure the gLite software suite as a non−privileged user and the whole tree should be relocatable. This means that as much as pos-si-ble software should be installed in the gLite root, which can be specified by setting the en-vironment va-riable `$GLITE_LOCATION`. The software should rely for its installation and configuration on this va-riable.The gLite software services shall source/parse a top level gLite configuration file and shall not rely on this being previously done (i.e. never assume that variables defined in this file are already present in the environment). The location of this file is platform dependent. For example on RedHat systems it would be `/etc/sysconfig/glite`, on Debian systems `/etc/defaults/glite`. This plat-form independent configuration for gLite services should be stored in `/etc/glite.conf` , which can be overridden by `$HOME/.glite.conf`.

This system dependent top level configuration file will have a key−value based syntax, containing all the needed basic prefixes with the following format:

```
GLITE_LOCATION=/opt/glite
GLITE_LOCATION_VAR=/var/opt/glite
GLITE_TMP=/tmp
GLITE_HOST_KEY=/etc/grid-security/hostkey.pem
GLITE_HOST_CERT=/etc/grid-security/hostcert.pem
```

The values should only be literal values. The typical values are given abo-ve with the exception of `$GLITE_LOCATION_VAR` which has a default of `$GLITE_LOCATION/var`.

For `$GLITE*` values not defined in the file, the defaults may be assumed by the daemon.

The variable `$GLITE_LOCATION` will point to `/opt/glite` by default

The variable `$GLITE_LOCATION_VAR` will point to the root of a tree for machine−specific files. Since the `$GLITE_LOCATION` tree should be mountable read−only by clients, the var directory for "variable" data should not be located within this tree, but in the top level var directory. The use of this va-riable will allow the gLite software to have a central complete installation which is shared by many ma-chines.

The variable `$GLITE_TMP` will point to the root directory of a world−writable area for temporary files. By default, this will have the value `/tmp`. The `$GLITE_TMP` area is subject to periodic clea-ning by cron scripts and at reboots. Consequently, daemons should not store state information in this area; such information should instead be placed in `$GLITE_LOCATION/`.

Packages are free to define their own environmental variables of the form

```
$GLITE_<package>_<anything>
```

The value of package should be negotiated with those managing the release. In all cases, these variables must be derived from the standard $GLITE_LOCATION, `$GLITE_LOCATION_VAR`, and `GLITE_TMP` variables.

The $GLITE_LOCATION_VAR area should be organized as to contain the following directories:

- etc : machine−specific configuration files

Note that the directory $GLITE_LOCATION_VAR/etc is for machine specific configuration files only. Configuration files that are machine independent should go in the directory $GLITE_LOCATION/etc.

More information on what should be contained in these directories can be found in the sections below.

A.2 Daemons and Services

Most of the grid services are intended to run continuously while the machine is up to service user re-quests. As such, a permanently running daemon controlled via an init.d script is appropriate rather than using inetd services.

Daemons should not run as root. If this is unavoidable, then the daemon should be written with a minimal kernel which runs as root. This daemon kernel can then either interact with another non−root daemon or spawn non−root processes to handle the bulk of the processing.

All initialization of a daemon should be done with the init.d script which controls the daemon. The init.d script should minimally support the start, stop, status, and restart methods. The init.d scripts are by default located in /etc/init.d, but it must be possible during the build to configure this location to a directory writable by non−privileged user, e.g. $GLITE_LOCATION/etc/init.d. It is advised to have a separate configuration package for each component and for each platform to maintain relocatability and flexibility.

Log files should be written to the standard area /var/log/glite and be written in such a way that the logs can be rotated. Each log entry must comprise the time stamp and the process identifier generating it. All authentication and authorization logging should by default be diverted to a separate secure log and such a way that it will never enter the normal logs. Such a secure logging target could be a Trusted Timestamping Service (TTS) or a remote secure syslog service.

Below is an example of a log entry generated by the process 8876 on 12 July 2004 at 3:44:59 p.m.:

```
2004/07/12 15:44:59 [8876] ERROR>

rotator.java:14 rotate() &#8211; [APPLICATION_FAULT] RotationService – No rotation parameter
```

This format allows for easy concatenation of logfiles and sorting by time stamp. Please note that the length of the time stamp is fixed.

It is foreseen that services will use common gLite logging facilities, which are based on industry−standard logging tools, such as log4j, log4perl, log4Cpp and log4c. The logging tools will be customized as described in https://edms.cern.ch/document/486630 [R10].

A.2.1 Daemon/Service credentials

The daemon should use the X509 credentials that are given by the environment variables $GLITE_HOST_CERT and $GLITE_HOST_KEY . If these are not set the services should respect the default SSL settings, e.g. by using X509_get_default_cert_dir_env() and X509_get_default_cert_file_env(). These locations can be set by $SSL_CERT_DIR and $SSL_CERT_FILE, respectively.

A service should check the user credentials and manage its local resources accordingly. However, if the service interacts with another service even a remote instance of the same service, then the user's (delegated)

proxy must be used. This is the only way to allow the remote site to verify the credentials itself and then to correctly log transactions and quotas.

When a daemon runs as a service user, host/service credentials should be made available for that user. The current practice is to make a copy of /etc/grid−security/host*.pem in the same directory (/etc/grid−security/tomcat*.pem) or into a subdirectory (/etc/grid−security/dpmmgr/dpmmgr*.pem) and change the ownership of the credentials to the service user.

A.2.2 Daemon/Service Configuration

The `init.d` scripts must contain the comments necessary to support the chkconfig command un-der RH Linux.

A full, annotated template configuration file must be provided with the package. This should have a .template suffix so that upgrades to the package do not wipe out a working configuration file. Changes in the package which require changes in the configuration file MUST be clearly stated in the package release notes and documentation.

Configuration components must be provided for each service or `init.d` sysV script that needs to be reconfigured.

A.2.3 Linux Packages profile.d Scripts

The package profile.d scripts must be installed in the standard gLite tree in `etc/profile.d`. These will be executed by a pair of master gLite profile.d scripts. The package profile.d script may assume that the `GLITE_LOCATION*` variables have been defined.

The master gLite profile.d scripts will ONLY execute those scripts in the `$GLITE_LOCATION_VAR/etc/profile.d` area. The profiles in `$GLITE_LOCATION/etc/profile.d` should always take the values in the variables set by the master profile. System administrators who wish to have a separate machine−specific area must make symbolic links to the desired scripts in `$GLITE_LOCATION/etc/profile.d`.

There must be no dependencies between the scripts in the `$GLITE_LOCATION/etc/profile.d` area, as the order of execution of files in this area is not guaranteed.

Developers must provide both sh and csh versions of profile.d scripts The sh version must be called *.sh and must execute correctly with the sh, bash, and zsh shells.The csh−shell version must be called *.csh and must execute correctly with csh and tcsh shells. This requirement is extremely important otherwise the grid software will place restrictions on what shells the end−user can or cannot use.

On installation, the scripts should have the executable bits set.

A.2.4 Cron Jobs and Scheduled Tasks

Cron jobs are appropriate for administrative tasks which must be carried out periodically. The scripts should be installed in the `etc/cron.d` area of the installation tree given by `$GLITE_LOCATION`.

To increase the portability of these scripts, use the standard Bourne shell features. In addition, make sure your script sets the appropriate environment it needs to work properly (`PATH`, `LD_LIBRARY_PATH`, etc.) and doesn't expect it to be set by the caller.

Within, the script should be a brief description of what the script does and a recommendation on how frequently it should be run. These scripts should not require additional customization beyond the standard

package configuration.

The execution of the cron jobs should be logged in `/var/log/glite`.

A.3 Resource Usage

A.3.1 Temporary Space

Developers should use the `$GLITE_TMP` variable to locate temporary disk space local to the ma-chine. This location is guaranteed to be world−writable. If the location is not set, `$TMPDIR` should be used. The use of the standard routines and methods to deal with temporary disk space is encouraged, e.g. tmpfile() and `mkstemp()` for C/C++ and for java the tmpfile method from `java.lang`

In the temporary area, create a subdirectory if your package will generate a large number of files, or there is a reasonable chance of collision with someone else's files. The name of the subdirectory should be descriptive of the package and reduce the chance of collision.

The package is responsible for cleaning up all files, sockets, etc. created by the package during its exe-cu-tion. Preferably this is done when the file is no longer needed but at the very least when the daemon shuts down. Individual files or directories may be left if an error has occurred and the developer feels that the temporary file may help with debugging or tracing the problem.

On worker nodes the jobs must use `$TMPDIR` (when set) when looking for temporary space.

A.3.2 Ports

Ports used by a service must be clearly documented. Additionally, the service port(s) must be easily and completely configurable as well as the port ranges for temporary ports. Services should never rely on fixed port numbers. Sites should be able to organize the network as they wish, internal or external connectivity, NAT, firewall etc.

A.3.3 Network Addresses

It should be possible to set the service to advertise some other host alias as the location than the real IP address. For example if the service is running in a box with an internal IP address it should register itself into the information system with the NAT server's address. This is also needed for systems with multiple interfaces.

A.3.4 Logging Formats

A set of logging rules can be found here [R9].

Appendix B. License and IPV6 compliance

B.1 License

The EGEE License is a Berkeley Software Distribution (BSD) style Open Source License. The text of which is at [R11] and in a file called LICENSE. gLite TWIKI website (https://twiki.cern.ch/twiki/bin/view/EGEE/EGEEgLite) contains refs to the Apache license: http://www.apache.org/licenses/LICENSE−2.0

A reference to the license should be included in every source file by including the license text [R11] . Currently a lot of source files either don't have any reference or have it wrong, attention should be paid in the future in order to include the right license reference in the new source files and correct the old ones.

B.2 IPV6 Compliant Code

To produce IPv6 compliant code the programmers should first use standard APIs that follow RFC 3493, 3542, 4038 recommendations. There are libraries offering compatible interfaces for many languages including C, C++, Java, Python and Perl. Programmers should stay at high levels of abstraction as far as possible. For example use DNS names instead of numeric address and take IPv6 programming guide into consideration. The main problem in porting applications to use IPv6 is to deal with different operating systems and their possible configurations and to be able to offer the best connectivity, i.e. to answer both to IPv4 and IPv6 requests if possible. The gLite deployment platform is mainly Scientific Linux 4 or 5. Scientific Linux 4 and 5 are dual stack nodes where the deployment platform will be configured to enable IPv6 or IPv4. Depending on the operating system features available (separate stack, dual stack, IPv6BindOnly) and language used by the application there are two main options for a server application:

1. wait for request on both IPv6 and IPv4 interface with two sockets
2. wait for request only on IPv6 interface and use IPv4 mapped IPv6 address to redirect IPv4 requests to the IPv6 part of the stack

The first point to consider is the targeted operating system and what features are available on it: dual−stack, separated stack, IPv4 mapped IPv6 address, ability for IPv6 socket to listen only IPv6 request (IPV6_ONLY / bindv6only). The second point to consider is the specific socket options that are required by the application. Not all of the socket options that are available in the IPv4 API may be available in the IPv6 API. In that case IPv4 mapped IPv6 addresses might not be suitable. Some languages, like Java, may hide some complexity and take account of the different operating possibilities. Java prefers IPv4 mapped IPv6 address if possible. Unfortunately the mapped addresses need special security consideration that may need to be handled by setting up an appropriate filter on the router or firewall. Furthermore the application itself may have to take the usage of IPv4 mapped IPv6 address into account. In the Guidelines for IP version independence in GGF specification the authors recommend using IPv4 mapped IPv6 address in part because the code is easier to develop. According to the specific needs of gLite and the deployment platform targeted the programmers and deployment staff have to consider this problem to define a strategy. A unique method may not handle all cases in gLite but a guideline on these topics will guide programmers in their choice. It is believed that having a coherent and seamless approach is desirable and will help, particularly during deployment when several different applications may be running on a single node.

A document with guidelines on how to produce ipv6 compliant code for the GRID can be found here: http://www.euchinagrid.org/docs/EUChinaGRID−Del2.2v3−1.pdf

A document explaining how to test IPV6 compliance for your application can be found here: https://edms.cern.ch/document/930868/1 [R12].

−− AndreasUnterkircher − 02 Feb 2009

−− AndreasUnterkircher − 19 Dec 2008

−− AndreasUnterkircher − 29 Oct 2008

−− ElisabettMolinari − 29 Sep 2008

This topic: EGEE > DevelopersGuide

History: r30 − 02 Feb 2009 − 13:50:07 − AndreasUnterkircher