Table of Contents

Table of Contents

Integration Procedure

Overview

Packages are built using the ETICS build system, for more details on how to use the ETICS build system see the ETICS user guide. The aim of integration is to take the packages from the ETICS repository and maintain yum repositories for certification, pre-production and production. One aspect of integration is to control what versions of packages are used and to ensure that everything is installable. Various integration scripts have been created to help automate many of the tasks involved. The integration dashboard shows the status integration. The node lxn1190 has been set up to act as a integration machine and has glite-release checkout out in the directory /home/glbuild.

Building

The ETICS build system is seen as a "Build Factory" which creates packages and places them in the ETICS repository. Integration is done at the package level and ensures than any problems with building or the build system will not affect integration. The ETICS repository should be seen as the interface between building and integration. All packages must be taken from the ETICS repository which can be found in AFS.

```
  /afs/cern.ch/project/etics/repository/org.glite           for gLite software
  /afs/cern.ch/project/etics/repository/externals         for external software
```

The rpms in there are stored in subdirectories of the format:

```
  /ETICS_MODULE_NAME/VERSION/PLATFORM/
```

e.g. in org.glite.data.gfal/1.9.4/slc4_ia32_gcc_346/GFAL-client-1.9.4-1.i386.rpm

For 3.1, the build is defined using the configuration *glite_branch_3_1_0* in the project org.glite. When a developer updates a component and creates a new subsystem, this will need to be communicated to the maintainer of the glite configuration. Usually this is done via the patch submission in Savannah. The builds occur automatically four times a day and currently takes about eight hours. The builds are done in parallel on all the supported platforms.

In addition, RPMs that are not yet in the ETICS system are stored in

```
  /afs/cern.ch/project/gd/www/integration/repository/org.glite
```

in the same structure as they would have within the ETICS repository for the certification and release process.

YUM Repositories

Note on using protect directive

YUM repositories which haven been 'protected' are **not fully protected**! Let me show you this on an example package.

A metapackage m_A has a dependency a_DEP. This is provided by a package B in a protected repository but also provided by package C in another repository (it doesn't matter if it is protected as well or not). YUM might then **not** take package B to satisfy a_DEP. The directive protect only prevents updates from other repos on already installed packages.

gLite Repositories

The YUM repositories for certification and PPS can be found in AFS, /afs/cern.ch/project/gd/www/glite/, which corresponds to http://grid-deployment.web.cern.ch/grid-deployment/glite/.

In 3.1, symbolic links per node should be made to the directory *generic* so that unique repositories can be used in the case of conflicts between node types. All the node types repositories are symbolic links to *generic* except *glite-WN*, dcache node types, which contain a symbolic link to *generic-dcache*, and *glite-HYDRA_mysql*.

In 3.2, all the packages are under *glite-GENERIC* and each node type contains also a repository with only repository metadata.

REPO	AFS	WEB
3.1 CERT	`/afs/cern.ch/project/gd/www/glite/cert/3.1/patches`	http://grid-deployment.web.cern.ch/grid-dep
3.2 CERT	`/afs/cern.ch/project/gd/www/glite/cert/3.2/patches`	http://grid-deployment.web.cern.ch/grid-dep
PPS (prepare)	`/afs/cern.ch/project/gd/www/glite/prepare/pps/`	http://grid-deployment.web.cern.ch/grid-dep
PPS	`/afs/cern.ch/project/gd/www/glite/pps/`	http://grid-deployment.web.cern.ch/grid-dep
BETA (prepare)	`/afs/cern.ch/project/gd/www/glite/prepare/beta/`	http://grid-deployment.web.cern.ch/grid-dep
PROD (prepare)	`/afs/cern.ch/project/gd/www/glite/prepare/prod/`	http://grid-deployment.web.cern.ch/grid-dep
PROD	`/afs/cern.ch/project/egee/gLite/`	http://linuxsoft.cern.ch/EGEE/gLite
Integration	`/afs/cern.ch/project/gd/www/integration/repository`	None
ETICS	`/afs/cern.ch/project/etics/repository`	http://eticssoft.web.cern.ch/eticssoft/reposit

Integration Repository

http://grid-deployment.web.cern.ch/grid-deployment/glite/integration/repository/externals/

On AFS: `/afs/cern.ch/project/gd/www/glite/integration/repository/` This directory contains packages which are not built in ETICS nor available from ETICS external repository. An example here are the dCache rpms.

VDT Packages

The gLite middleware uses some VDT packages which -in case of an update- may have to be downloaded into the integration repository or into etics (requesting external package form). You can find the VDT packages here: http://vdt.cs.wisc.edu/documentation.html.

These repositories include the following packages :

- uberftp-client
- vdt_globus_essentials
- vdt_globus_rm_essentials
- vdt_globus_jobmanager_common

Alternatively the vdt packages can be found in ETICS here http://eticssoft.web.cern.ch/eticssoft/repository/vdt/.

Certification

http://grid-deployment.web.cern.ch/grid-deployment/glite/cert/

On afs: `/afs/cern.ch/project/gd/www/glite/cert/`

The repository structure is as follows for each patch:

```
./$release/patches/$patchnumber>/$os/$arch/RPMS.$patchnumber
./$release/patches/$patchnumber>/$os/$arch/RPMS.externals
```

Note: In order to install the patch, the production repository for the affected metapackage has to be enabled too.

Pre-production Service (PPS)

http://grid-deployment.web.cern.ch/grid-deployment/glite/pps/

On afs: `/afs/cern.ch/project/gd/www/glite/pps/`

The repository structure is as follows for 3.1 (where repo is generic, glite-WN, generic-dcache or glite-HYDRA_mysql; and arch is i386 or x86_64):

```
./$release/repo/sl4/$arch/RPMS.release
./$release/repo/sl4/$arch/RPMS.externals
./$release/repo/sl4/$arch/RPMS.updates
./$release/repo/sl4/$arch/repodata
```

The repository structure is as follows for 3.2:

```
./$release/glite-GENERIC/sl5/x86_64/RPMS.release
./$release/glite-GENERIC/sl5/x86_64/RPMS.externals
./$release/glite-GENERIC/sl5/x86_64/RPMS.updates
./$release/glite-NODE_TYPE/sl5/x86_64/RPMS.release/repodata
./$release/glite-NODE_TYPE/sl5/x86_64/RPMS.externals/repodata
./$release/glite-NODE_TYPE/sl5/x86_64/RPMS.updates/repodata
```

Production

http://grid-deployment.web.cern.ch/grid-deployment/glite/prepare/prod

On afs: On afs: `/afs/cern.ch/project/gd/www/glite/prepare/prod/`

This location is where we work, but the actual production repository is in `/afs/cern.ch/project/egee/gLite` or `http://linuxsoft.cern.ch/EGEE/gLite`.

ATTENTION WITH CHANGES : Changes into `/afs/cern.ch/project/egee/gLite` are effective every 30min past the full hour (1.30,2.30, 3.30, etc..)!

The structure of the repository is as follows for 3.1:

```
./$release/generic/$os/$arch/RPMS.release       for originaly released gLite rpms
./$release/generic/$os/$arch/RPMS.updates      for any rpm released after the initial release
./$release/generic/$os/$arch/RPMS.externals    for external rpms released originally
```

The repository structure is as follows for 3.2:

```
./$release/glite-GENERIC/sl5/x86_64/RPMS.release
./$release/glite-GENERIC/sl5/x86_64/RPMS.externals
./$release/glite-GENERIC/sl5/x86_64/RPMS.updates
./$release/glite-NODE_TYPE/sl5/x86_64/RPMS.release/repodata
./$release/glite-NODE_TYPE/sl5/x86_64/RPMS.externals/repodata
./$release/glite-NODE_TYPE/sl5/x86_64/RPMS.updates/repodata
```

Package Life Cycle

A set of packages are requested to be added to the release. A patch represents this set of packages. A patch is usually identified by a number, however, for an initial release the patch can be identified by a node type name.

A patch must first be added to the certification repository so that it can be tested.

```
/afs/cern.ch/project/gd/www/glite/cert/3.X/patches/os/arch/RPMS.patchnumber
```

Once a patch has been certified, it can be considered for pre-production or production. In the pps and production repository, the packages required should be put into one of the following directories.

New Packages
: RPMS.release

Updated Packages
: RPMS.updates

External Packages
: RPMS.externals

In 3.2, packages are signed once they are moved into the intermediate `certified` repo, which is just a location under `http://grid-deployment.web.cern.ch/grid-deployment/glite/certified/3.X/patches/os/arch/RPMS.patch` where the relevant packages are signed and stored. This happens just before moving to pps. The sequence would be cert -> sign rpms -> certified -> pps -> prod.

In 3.1, we don't sign packages but we also use the `certified` intermediate repository.

Information Management

As integration is done at the package level, the name and version is required to identify a package. To be able to locate the package in the ETICS repository, the corresponding component name that is used in the ETICS system is also required. The format used to store this information in a file is

```
|package-name|package-version|etics-name|etics-version|*
```

The star at the end is only required to identify the first level dependencies which are used to create a meta-package. All the entries without the star are lower level dependencies. Putting only the first level dependencies in the meta-package is required for testing, however, for production all the lower level dependencies will also need to be include in the meta-package. This is required as we need to check the dependencies during testing but for the release we need to include all packages so that the version of the meta-package identifies what is installed.

Lists of packages for each node type are created for the release. These lists can be found in the CVS directory *glilte-release/prod-lists* or *glilte-release/pps-lists* directory. The list should checked back into CVS after changes have been made and a tag should be made when the list is released.

Repository Management

The repositories are created when creating a release to production or pps. The script to be used in *make-release* and it uses the lists to build the repositories. It writes by default in the *prepare area* and later the repositories can be moved to the official pps and production locations.

The Application Area

For the 3.1 releases we are introducing an extra distribution route for a subset of the clients. We will maintain a section of the Application Area repository in order to give the experiments early access to updates and to enable them to use these clients in building their own distributions.

The client list is held in the file `glite-AA` and this should be maintained just like any other rpm list. The Application Area repository should be populated with any client updates which **enter** certification, and the list defining what has been released should be updated when a release is made. The full story is described here.

Patch Handling

To make it easier to manage the transition of patches between the different repositories, *the move-patch* command can be used to move the patches between the different state and hence repository locations.

In order to better understand the different patch and bug states, please check these two links:

- How to handle bugs
- How to handle patches

Tracking

All build and integration problems must be tracked in Savannah. The field *Category* should be set to *Integration*.

Problems found in a release should be described in the Integration Diagnosis page.

All communication of integration issues should be done through the integration email list:
`gd-release-team@cern.ch`

HowTos

Handling patches within product teams

Product teams should create a patch whenever they want to release a new version of their products to production. Follow the Product Team Integration guide to know the details of the process.

Handling patches with scripts for CERT, PPS, Production

All scripts are located here: here . They are configured to use a common config file **config.sh** containing all relevant locations/control flags, etc. This can be found in `glite-release/config.sh`.

Each script contains a help/documentation.

Checking out the release script(s)

```
user=yourname
klog.krb $user;
kinit $user;
export CVSROOT=:gserver:$user@glite.cvs.cern.ch:/cvs/glite;
export CVS_RSH=ssh;
cvs co glite-release;
yum install createrepo;
yum install repoview;
```

Checking out the savannah client

more Information here : SavannahCommandLineInterface

```
cvs co org.glite.integration.savannah;
org.glite.integration.savannah/cli/install_deps.sh
```

Releasing to Certification

Follow the 'Move to certification' check list that includes the steps to be followed when moving a patch into `Readyfor certification`.

Releasing to PPS

Follow the preproduction release check list that includes the steps to be followed when doing a preproduction release.

Releasing to Staged rollout

Follow the Staged rollout release check list that includes the steps to be followed when doing a staged rollout release.

Releasing to PROD

Follow the production release check list that includes the steps to be followed when doing a production release.

Signing RPMs

RPMS are signed in gLite 3.2. Check the certification signing procedure to know all the details about the signing procedure.

In order to sign rpms, you need to create the following files in the machine where you want to sign the rpms:

```
/root/.gnupg -> This contains the key and the necessary files to sign with the GPG key.
/root/.rpmmacros -> This file should contain:

%_signature gpg
%_gpg_name Gd Integration <gd-release-team@cern.ch>
```

RPMs are signed only in 3.2 and this is done automatically within the glite-release scripts. The integrator has to pay attention when moving patches into `certified` since the password of the key will be asked.

How and when to use the Integration repository

The integration repository is located under: `/afs/cern.ch/project/gd/www/integration/repository`.

And it contains two directories called:

```
externals
org.glite
```

WHEN to use this repository: whenever we need a package that is not built in ETICS, and it s not available in DAG, jpackage or the OS repositories either.

HOW to use this repository:

- If the package comes from an external project like maui, torque, dcache or vdt, use `externals`.
- If the package is built by us or anybody else in the experiments not using ETICS, then use `org.glite`.

Just check first if there s a subdirectory under `externals` or `org.glite` with the structure `package_name/x.y.z/platform`, otherwise create it and copy the package there. Please, put a comment in the Changelog file and always keep the Integration team informed. It s important to know what we put there.

Why is this repository important? It s important in particular for 3.2 releases, becase in gLite 3.2 external packages are not signed and only the packages coming from `/cern.ch/project/gd/www/integration/repository/externals` are regarded as externals and won t be signed. The packages under `org.glite` are not considered external packages and they will be signed.

Rolling back the latest release of PROD

- Restore CVS prod lists
    - For each affected metapackage of update NN
        - cvs update -j HEAD -j glite-release_glite-XXXX_R_x_y_z-1_0 glite-release/prod-lists/slc4_[a32|x86_64]_gcc346/glite-XXXX
        - cvs ci -m "Restoring prod list for glite-release_glite-XXXX_R_x_y_z-1_0" glite-release/prod-lists/slc4_[a32|x86_64]_gcc346/glite-XXXX
        - cvs tag glite-release_glite-XXXX_R_x_y_z_0 glite-release/prod-lists/slc4_[a32|x86_64]_gcc346/glite-XXXX
- Prepare a backup of the release pages
    - To mount the glite_repository in the machine:

```
modprobe cifs
mount -t cifs -o username=CERN\\glbuild //glite.web.cern.ch/glite /mnt/glite_reposit
```

- ♦ Copy the directories that you want to backup
- Prepare the release pages for the next update
  - ♦ For each affected metapackage of update NN
    - ◊ cd /mnt/glite-release/package/R3.X/(sl5_)(x86_64)/deployment/glite-XXXX/
    - ◊ cp latest-old/glite-XXXX latest/glite-XXXX
- Make the release as usual (check with SA1 SA3 the value of the release field in the patches)

The rollback script generates the list of CVS tags to be applied into the *lists/pps-lists* or *lists/prod-lists* to rollback to a previous pps or production update. It also deletes from the repo the packages introduced in the current version.

```
rollback -s [pps|prod] -p [slc4_ia32_gcc346|slc4_x86_64_gcc346|sl5_ia32_gcc412|sl5_x86_64_gcc412]
```

When the script is executed, find in the `TMP_DIR` the list of rpms to keep, rpms to remove, set of concerned patches and the script to apply the CVS tags into the lists.

Create a new metapackage in PPS or PROD

⚠ ***Note:*** For this, the Savannah patch must have all the packages which belong to this new metapackage 'to be added' in Metapackage changes field.

Example (introduction of a new metapackage) :

Savannah Metapackage changes field

```
A|glite-SCAS|glite-security-scas|0.2.2-5
A|glite-SCAS|glite-security-saml2-xacml2-c-lib|0.0.14-2
A|glite-SCAS|glite-yaim-scas|1.0.0-8
A|glite-SCAS|glite-yaim-core|4.0.5-7
```

Execute Command (example for PPS, same is for PROD)

```
#move to PPS
./scripts/move-patch -s pps -m PPS-glite-SCAS 2511
```

Result
The Result will be a new file in pps-lists/<DEFAULT_PLATFORM>/PPS-glite-SCAS with this this content:

```
#> cat PPS-glite-SCAS
glite-security-saml2-xacml2-c-lib|0.0.14-2|org.glite.security.saml2-xacml2-c-lib|0.0.14
glite-security-scas|0.2.2-5|org.glite.security.scas|0.2.2
glite-yaim-core|4.0.5-7|org.glite.yaim.core|4.0.5
glite-yaim-scas|1.0.0-8|org.glite.yaim.scas|1.0.0
```

The same procedure has to be done for production lists.

Example (introduction of a new metapackage with changes to other lists) :

The mechanism allows not only to introduce a new metapackage with a patch, but also to do changes at the same time to other metapackage lists. This is archived by adding metapackage changes for those lists.

The Example here shows the introduction of glite-SCAS to PPS and at the same time adding new packages to glite-WN.

Savannah Metapackage changes field

```
A|glite-WN|glite-security-lcmaps-plugins-scas-client|0.2.2-5
A|glite-WN|glite-security-saml2-xacml2-c-lib|0.0.14-2
```

```
A|glite-SCAS|glite-security-scas|0.2.2-5
A|glite-SCAS|glite-security-saml2-xacml2-c-lib|0.0.14-2
A|glite-SCAS|glite-yaim-scas|1.0.0-8
A|glite-SCAS|glite-yaim-core|4.0.5-7
```

Other

Creating a Metapackage

```
 Usage : create-metapackage -l list -u update_nr -p platform -v version [-m metapackage name] [-r
scripts/create-metapackage -l prod-lists/slc4_ia32_gcc346/glite-BDII -u 24 -p slc4_ia32_gcc346 -v
...
Wrote: /root/glite-release/create-metapackage.tmp/SRPMS/my-BDII-metapackage-3.1.12-0.src.rpm
Wrote: /root/glite-release/create-metapackage.tmp/RPMS/i386/my-BDII-metapackage-3.1.12-0.i386.rpm
..
```

Creating a version marker (glite-WN x86_64 case)

```
scripts/create-metapackage -l prod-lists/slc4_x86_64_gcc346/glite-WN -u 24 -p slc4_x86_64_gcc346
...
Wrote: /root/glite-release/create-metapackage.tmp/SRPMS/glite-WN-version-3.1.12-0.src.rpm
Wrote: /root/glite-release/create-metapackage.tmp/RPMS/x86_64/glite-WN-version-3.1.12-0.x86_64.rp
..
```

Creating a rpm group description

```
 Usage: create-rpm-group-description -p platform -v version -l list -o outfile
scripts/create-rpm-group-description -p slc4_x86_64_gcc346 -v 3.1.0-0 -l prod-lists/slc4_x86_64_g
```

Generating a group repository

```
 scripts/create-rpm-group-description -p slc4_x86_64_gcc346 -v 3.1.0-0 -l prod-lists/slc4_x86_64_
# Generate the repository, rpms must be there already (e.g. RPMS.release)
scripts/generate-yum-repository -d /tmp/del2 -g /tmp/comps-del.xml
```

Creating a Metapackage for the first time with 32bit and 64bit dependencies

In order to create new metapackages with 32bit and 64bit dependencies, the first time you move the corresponding patch into `Ready for Certification`, you need to do the following manual steps:

- Create the dependency list in the glite-release module in CVS under `glite-release/prod-lists/architecture/glite-metapackage`. The dependency list should contain the `#META-add-package-platform` entries for the 32bit rpms. See the next section `Packages for Groupinstall must be available from both platforms` for more details. Commit the changes.
- In the Savannah patch, specify in the `RPM name` field only the 64bit rpms. Don't forget to specify the Metapackage changes for each of these rpms.
- Create the patch directory `/afs/cern.ch/project/gd/www/glite/cert/3.x/patches/patch_number/slx/x86_64/RPMS.patch_numbe`
- Copy the 32bit rpms in the patch directory you've just created.
- Move the patch into `Ready for Certification` as usual. Remember to install the metapackage with `yum groupinstall` to get 32bit and 64bit versions.

Moving a patch into certification with 32bit and 64bit dependencies

In order to create a patch repository with 32bit and 64bit dependencies, you need to do the following manual steps:

- Create the patch directory `/afs/cern.ch/project/gd/www/glite/cert/3.x/patches/patch_number/slx/x86_64/RPMS.patch_numb` or `/afs/cern.ch/project/gd/www/glite/cert/3.x/patches/patch_number/slx/x86_64/RPMS.externals` if the package is an external one.
- Copy the 32bit rpms in the patch directory you've just created.
- If the package is a new dependency, add in the corresponding prod-list the `#META-add-package-platform|package_name|sl5_ia32_gcc412` to be able to properly build the groupinstall taking the new dependencies.
- Move the patch into `Ready for Certification` as usual. Remember to run the following commands in certification to get 32bit and 64bit versions:
    - `yum groupinstall cert-glite-UI/WN` if there are new depencies.
    - `yum groupupdate` if there are only new versions.

Backup copy of the gLite web pages

Select the path `\\glite.web.cern.ch\glite` and right click on the folder or file you would like to retrieve. You can then select `properties` and in the properties window the tab `previous version`.

You can still make the request through Helpdesk giving the name of the file or folder to recover and the date to get it back from.

Testing

To perform automatic tests after moving a patch to certification, or after creating a new preview or production release, a set of scripts have been developed. These scripts allow to perform the needed tests inside a chroot environment using mock, and can be the base for more automatic or manual tests.

Prepare a new machine for testing

Before performing the tests, we need to prepare the test machine with the following steps.

Get a copy of glite-release from CVS

First, we need a copy of glite-release where we can find the scripts needed under glite-release/scripts/chroot_test_scripts.

Prepare the environment

The prepare_env script will install and configure mock and will create specific config files for the different platforms. Run it without parameters.

Prepare the chroot tar files

Afterwards, we need to generate the clean chroots for each platform that will be the base for the test, that can be done executing prepare_tar for each needed platform:

```
./prepare_tar sl-4-i386.cfg
./prepare_tar sl-4-x86_64.cfg
./prepare_tar sl-5-x86_64.cfg
```

Create the test

A test consist in a directory containing all the needed information to perform the automatic test. This directory structure can be generated automatically using the script create_test.

The options accepted are:

```
Usage: create_test -r release -p platform -s state [-a] -n [patch_number|update_number]

 options:
  -r   release
  -p   platform
  -s   [cert|preview|prod]
  -n   patch_number if -s cert, update_number if -s [preview|prod]
  -a   don't reinstall the Savannah DB
```

Create a test for a patch

```
./create_test -r 3.2 -p slc5_x86_64_gcc412 -s cert -n 3225
```

Will create a test directory named 'patch_3225'.

Create a test for a preview

```
./create_test -r 3.1 -p slc4_ia32_gcc346 -s preview -n 05
```

Will create a test directory named '3.1.0_Bundle_05-i386'.

Create a test for a release

```
./create_test -r 3.1 -p slc4_x86_64_gcc346 -s prod -n 55
```

Will create a test directory named '3.1.0_Update_55-x86_64'.

Test directory structure for verification and manual modification

In this section the test directory structure is explained to allow manual verification before running the test or to introduce modifications in the default tests created with create_test.

Directory estructure

Inside a test directory there is a directory for each metapackage to test. Inside the metapackage directory there are two files and another directory as follows.

In this example, we have the metapackage 'glite-CREAM' the fake metapackage 'glite-XXXXX' to show a test for multiple metapackages. The final directory structure is:

```
3.1.0_Update_56-i386/
3.1.0_Update_56-i386/glite-CREAM
3.1.0_Update_56-i386/glite-CREAM/rpms.list
3.1.0_Update_56-i386/glite-CREAM/repos
3.1.0_Update_56-i386/glite-CREAM/repos/dag.repo
3.1.0_Update_56-i386/glite-CREAM/repos/glite-CREAM.repo
3.1.0_Update_56-i386/glite-CREAM/repos/jpackage.repo
3.1.0_Update_56-i386/glite-CREAM/repos/lcg-CA.repo
3.1.0_Update_56-i386/glite-CREAM/install_cmd
3.1.0_Update_56-i386/glite-XXXXX
3.1.0_Update_56-i386/glite-XXXXX/rpms.list
3.1.0_Update_56-i386/glite-XXXXX/repos
3.1.0_Update_56-i386/glite-XXXXX/repos/dag.repo
3.1.0_Update_56-i386/glite-XXXXX/repos/glite-XXXX.repo
3.1.0_Update_56-i386/glite-XXXXX/install_cmd
```

Install command

Inside each metapackage directory, there is a file named 'install_cmd' containing the installation command needed for this metapackage, for example:

```
/usr/bin/yum -y install glite-CREAM
```

Rpm lists

There is another file named 'rpms.list' containing a list of rpm names that should have been installed after the test, for example:

```
glite-ce-blahp-1.12.4-0.slc4.i386.rpm
glite-ce-ce-plugin-1.11.1-13.noarch.rpm
glite-ce-cream-1.11.1-13.noarch.rpm
glite-ce-job-plugin-1.11.1-13.noarch.rpm
glite-ce-monitor-1.11.1-13.noarch.rpm
glite-yaim-cream-ce-4.0.9-2.noarch.rpm
```

Repo files

Finally, there is a directory named 'repos' containig the special repo files needed to install the meta package, in this example:

```
repos/dag.repo
repos/glite-CREAM.repo
repos/jpackage.repo
repos/lcg-CA.repo
```

Run a test

Running a test

After creating and/or modifying a test we have to run it. For that, there is a script named 'run_test'. The options accepted are:

```
Usage: run_test -r release -a arch -t test_directory -d

 options:
  -r   release
  -a   arch
  -t   test_directory
  -d   don't delete the chroot
```

To continue with the previous example we could run:

```
./run_test -r 3.1 -a i386 -t 3.1.0_Update_56-i386
```

It will give some output and if the test is successful it will print: "The installation test was successful!"

Playing with the chroot if the test fails

If the installation test was not successful (or if we provided the -d option) we can go to the chroot and see what is the state after the test.

The chroots are in /var/lib/mock/, if the test fails, we can go there, enter in the chroot directory, and we will see a root directory containing the whole chroot. The name of the chroot depends on the platform, the default names are sl-4-i386, sl-4-x86_64, sl-5-x86_64.

If we provide the options -d, we could end up with more than one chroot (one for each metapackage to test), therefore, the metapackage name is append to the chroot name.

If we want to enter in the chroot as the script does we can use the mock commands. First, we need to log as mockuser:

```
su mockuser
```

And then, we can run a mock shell with:

```
mock shell -r sl-4-i386.cfg
```

changing the config file depending on the architecture as before.

Note that if you want to do this after using the -d option, first you have to rename the chroot of the metapackage you want to it's original name. For example:

```
mv /var/lib/mock/sl-4-i386_glite-CREAM /var/lib/mock/sl-4-i386
```

If the installation fails, it's needed to clean the chroot before running a new test. See the next section to learn how to do it.

Cleaning the chroot

WARNING!!

To clean a chroot NEVER use an rm command. Can be some mounted filesystems and that could remove important files (for example in AFS).

Instead, use the 'clean_chroot' script (always as root, not as mockuser):

```
./clean_chroot sl-4-i386
```

If you have used the -d option, you should move the chroot to the original name before using 'clean_chroot'. It should be implemented an option in 'clean_chroot' to do this automatically.

Known Issues

Packages for Groupinstall must be available from both platforms

It is currently only possible to have 32 **AND** 64 Bit versions of a certain package on a group installation. This means, that a groupinstall will always contain the 64 Bit packages, but not a 32Bit version of which 64 Bit version is not stated in the list.

3 Examples for a metapackage list :

```
..
#META-add-package-platform|classads|slc4_ia32_gcc346
classads|0.9.8-2|classads|0.9.8
..
```

OK . This will copy both versions of classads in the repo and state the package in the `comps.xml`

```
..
classads|0.9.8-2|classads|0.9.8
..
```

OK . This will copy the 64Bit versionof classads in the repo and state the package in the `comps.xml`

```
..
classads|0.9.8-2|classads|0.9.8
..
```

WRONG! . This will copy the 32 Bit version of classads but it will **NOT** be stated in the `comps.xml`

Links

Node Tracker : Glite31NodeTracker
Integration Dashboard : integration dashboard
AA RSS Feed : http://grid-deployment.web.cern.ch/grid-deployment/glite/aa-updates.rss
gLite RSS Feed : http://glite.web.cern.ch/glite/news.rss
CA update : https://twiki.cern.ch/twiki/bin/view/LCG/SecurityCaUpdate#Procedure_for_CA_update

---

This topic: EGEE > IntegrationProcedure
Topic revision: r73 - 2009-11-11 - MariaALANDESPRADILLO