Ideas to improve the development of gLite

Francesco Giacomini

JRA1 All Hands Meeting
Nicosia, 6-7 May 2009

Information Society and Media

Summary

- Integration
- Source RPMs
- Use of subsystems
- /usr vs /opt/glite
- CVS replacement
- Major and critical bugs

Integration

- **Integration at different stages**
  - Here interested only at **build** time
- **Purpose: build cleanly all gLite on a number of platforms**
- **How:**
  - Stable gLite baseline
  - Clean configurations
  - Portable code
  - Portable build instructions
- **Nightly builds**
  - JRA1 twiki → Build Reports
  - Everybody should look at them and fix any failure
- **Need one person responsible for the integration**

Stable gLite baseline

- **(quasi-) periodic gLite releases**
  - e.g. about every six months
  - gLite N vs gLite X.Y
    - On as many platforms as possible
  - A gLite release is characterized by well-defined versions of all the components
    - Backwards compatibility of the interface
    - Extreme stability for external dependencies
      - *But platform-dependent*
    - Care for internal core components
      - *Need to list them*
    - More flexibility with high-level components
- **External dependencies come normally from the OS distribution**
  - Exceptionally they can be distributed directly as part of gLite

Interface

- **What constitutes an interface?**
  - In practice, not only logically
  - Everything made visible by a component
- **API, ABI, WSDL**
- **Properties in ETICS configurations**
- **Libraries used by static libraries**
  - If not masked by libtool, pkg-config, M4, or similar
  - What about Java?
- **…**

Backwards compatibility

- **A change is backwards compatible if it requires no more than a rebuild of a dependent component**
  - Acceptable
- **A change is backwards incompatible if it requires changes in a dependent component**
  - In code, build instructions, configuration, ...
  - By default not acceptable
  - Needs to be negotiated before being included in the integration build

Clean configurations

- **Ideally:**
  - A component configuration:
    - Specifies only dynamic dependencies in the Default platform for direct dependencies
      - Direct means that something of that component is used (in the code, in the build, in the configuration)
    - Does not override properties
    - Does not specify .DEFAULT properties
  - A subsystem configuration is empty, apart from the list of component configurations
    - See later
  - A project configuration specifies .DEFAULT properties per platform
- **Exceptions are managed case by case in the EMT**

Use of subsystems

- **Set of components with significant relationship at development/build time**
  - Two components go in the same subsystem **if and only if** they usually need to be developed/built together
  - Consider that ETICS doesn't make it simple to (remote) build two different subsystems at the same time
- **A single component should be buildable without the need to build the whole subsystem**
  - Avoid to keep information, e.g. properties, at subsystem level
- **If you wish to integrate two configurations of the same subsystem into the same project config (e.g. for two major releases of the same service), consider having two subsystems**
  - e.g. LB 1 and LB 2 available in gLite 3.2

One component version, two gLite releases

- **What if the same version of a component/subsystem is compatible with two gLite releases?**
  - e.g. LB 2.3.4 can be included both in gLite 5 and gLite 6?
- **Does it make sense?**
- **Two identical (deep) clones of the LB 2.3.4 configuration, with glite5/glite6 in the age**
  - glite_lb_R_2_3_4_glite5, glite_lb_R_2_3_4_glite6
- **Consequences:**
  - The age should not be kept in CVS (the “code” would not be identical)
  - The age should be passed as arguments to the build commands
  - Better not to use the value of the age in the CVS tag

Source RPMs
(thanks to Elisabetta Molinari)

- **Goal: having buildable source RPMs as an artefact of an ETICS build**
- **`etics-build --createsource` ... creates source tarballs and source RPMs in addition to binary ones**
- **It works if build instructions and configurations are well written, e.g.**
  - Dependencies are correctly defined
    - B on X-devel, R on X
  - Do not assume a certain file structure outside the component being built, e.g.
    - Use –`with-X=` in `configure` invocations
    - Use `$(*.location)` ETICS properties

Source RPMs: an example

- **glite-wms-manager-3.2.1-14.src.rpm**

```
# rpmbuild -bi glite-wms-manager.spec
error: Failed build dependencies:
glite-wms-helper >= 3.2.1 is needed by glite-wms-manager-3.2.1-14
glite-wms-utils-jobid >= 3.1.5 is needed by glite-wms-manager-3.2.1-14
```

- **After installing those dependencies:**

```
# rpmbuild -bi glite-wms-manager.spec'
configure: error: unknow version of boost
error: Bad exit status from /var/tmp/rpm-tmp.77448 (%build)
RPM build errors:
Bad exit status from /var/tmp/rpm-tmp.77448 (%build)
```

  - Missing B dependency on boost-devel

Multi-package components

- **New ETICS feature**
  - Still in design and early prototype phase
  - https://twiki.cern.ch/twiki/bin/view/ETICS/SA1Multipackaging
- **Give the possibility to produce multiple packages out of the same build**
  - Each package is represented by a subconfiguration of an ETICS component, specifying a different install target
  - Dependent components can specify runtime deps on a package
  - Typical usage: runtime (shared libs) and build time (header files and static libs) packages
- **All our components should provide a runtime and a build-time package**
  - e.g. wms-common and wms-common-devel
  - Required to have a proper gLite SDK (important goal for year II)

Code and Build portability

- **For C/C++**
- **Maintain M4 macros**
  - Need to assign responsibility to someone
- **For Java?**

Build Integration is not enough

- **A successful integration at build time is necessary but not sufficient**
- **What is built needs to work**
  - Successful deployment, successful run, ...
- **Extend the "interface" concept to include deployment time and run-time**
  - Packages keep installing the same things, components use the same files, log the same events with the same format, etc.
  - Need to precisely define and document what the interface is for each component
- **Same constraint on backwards compatibility**
  - Given gLite Y, version X+1 of a component cannot replace version X if it breaks the backwards compatibility according to the definition of interface (in the broadest sense) for that component

/opt/glite

- Is there any reason why we should complicate our life and install in /opt/glite?
- Why not simply in /usr?

CVS replacement

- **The CVS service at CERN is planned to end at the end of 2009**
- **Replaced by SVN**
- **Do we really want to use (only) SVN?**
- **The world is going towards Distributed Version Control Systems**
  - *See Akos's and Vincenzo's presentations*
- **Need to keep a central repo for the centralized operations (quality assurance, releases, …)**
- **Working Group to define guidelines for the migration**
  - Volunteers (apart from Akos and Vincenzo)?

Major and critical bugs

- **The management of major and critical bugs is not satisfactory, starting from their classification**
  - What is a critical/major bug?
- **For example:**
  - Critical bug: the system is unusable for many users and there is no workaround
  - Major bug: the system is usable but its usability is severely limited
  - …
- **The severity can be suggested by the bug submitter, but it's the EMT that ultimately decides on it**
  - If the submitter does not agree, the issue is escalated to the TMB
- **Fixes to critical and major bugs go in a patch by themselves**
  - A critical fix cannot be delayed by another bug fix (even critical)