DISTRIBUTED TESTING INFRASTRUCTURE AND PROCESSES FOR THE EGEE GRID MIDDLEWARE

D. Bosio, D. Collados, L. Guy, M. Reale, M Theile, CERN, Geneva, Switzerland
D. Groep, D.Salomoni, J. Templon, NIKHEF, Amsterdam, The Netherlands
S. Traylen, CCLRC, RAL, Oxon, UK

Abstract

Extensive and thorough testing of the EGEE middleware is essential to ensure that a production quality Grid can be deployed on a large scale as well as across the broad range of heterogeneous resources that make up the hundreds of Grid computing centres both in Europe and worldwide.

Testing of the EGEE middleware encompasses the tasks of both verification and validation. In addition we test the integrated middleware for stability, platform independence, stress resilience, scalability and performance.

The EGEE testing infrastructure is distributed across three major EGEE grid centres in three countries: CERN, NIKHEF and RAL. As much as is possible the testing procedures are automated and integrated with the EGEE build system. This allows for continuous testing together with the incremental daily code builds, fast and early feedback to developers, and for the easy inclusion of regression tests.

INTRODUCTION

The objective of the testing of the EGEE Middleware Re-engineering and Integration Research Activity is to discover and report as many bugs and deficits in the gLite middleware as is possible prior to release to the Grid Operations Activity, SA1. To this end, the testing activity will:

- Test all middleware components that form part of the integrated middleware system to ensure a production quality release that fulfills the requirements of the applications,
- Assess that all software requirements have been correctly and completely implemented and are traceable to system requirements,
- Test the integrated software for scalability, platform independence and stress resilience.

The testing activity is distributed across across three major EGEE sites: CERN, NIKHEF and RAL. Such a distributed environment is considered essential to ensure that assumptions valid at only one site have not been made. Three sites is considered the minimum required to test all the basic Grid functionality.

This paper describes the scope of the gLite middleware testing, the testing infrastructure and procedures used.

GLITE TEST SCOPE

The gLite middleware follows a *Service Orientated Architecture (SOA)*, as described in detail in [4], [5]. The various services and components are initially made available individually together with installation and user guides and are tested independently. Full system testing is carried out after some initial successful individual component based testing.

The testing of the gLite middleware is based on and is driven by the requirements, specifications and interfaces described in [5], [6], [8], [9], [10]. Table 1 lists all the gLite services and components that are tested, organized according to service decomposition.

Table 1: gLite services and components that are tested.

Services	**Components**
Information and Monitoring (R-GMA)	Producer Service Consumer Service
Job Management Services	Computing Element Workload Management
Data Services	Storage Element Replica Catalog Metadata Catalog File Catalog Catalog services File IO File transfer service File placement service
Security	VOMS Server Services with security
Access	Grid Access Service
Accounting	Accounting HLR server
Packaging	Package Manager

Features of the middleware services and components that are in the testing scope are:

- Functionality
- Security
- Performance
- Error handling and recovery
- Installation, configuration, upgrade and uninstallation
- Public user interfaces
- Conformance to API specifications and schemas
- Various deployment and configuration scenarios
- System resilience
- Platform independence

Features of the middleware services and components that are not in the testing scope are:

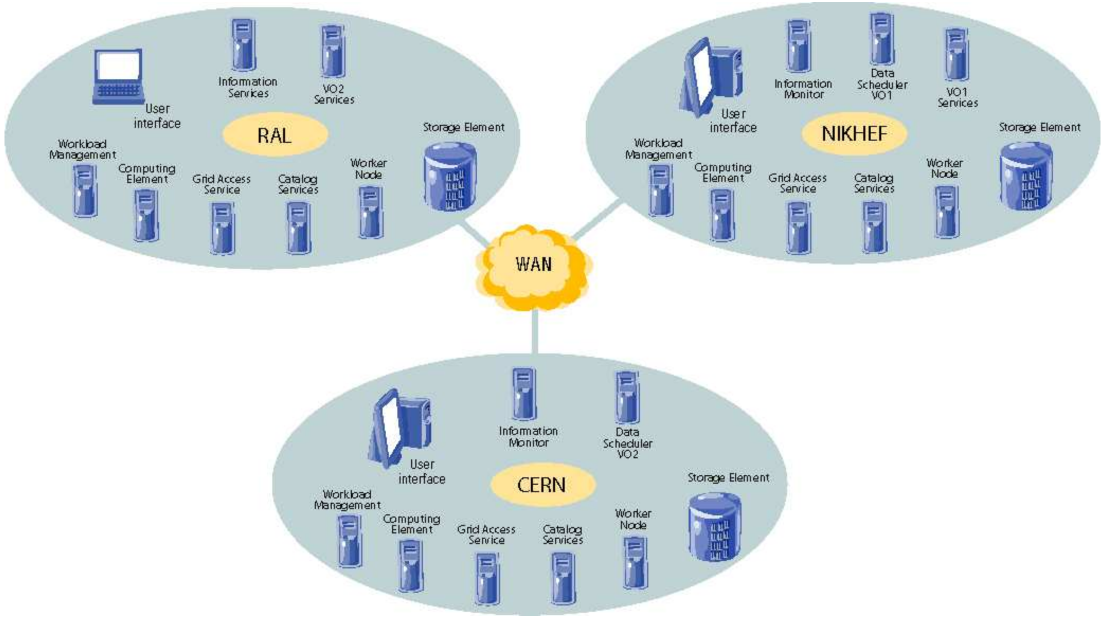

Figure 1: The distributed gLite testing infrastructure.

- Application servers (Tomcat, Oracle, etc)
- External Grid toolkits (Globus, VDT, LCG, etc)
- Condor, PBS and other scheduling and batch systems
- LDAP servers and databases (MySQL, Oracle)
- Network infrastructure
- Underlying fabric infrastructure and management systems (Quattor, etc)
- Underlying storage systems (Castor, HPSS, dCache, etc)
- Application specific metadata catalogues
- Conformance to standards (WSDL, WSI, etc)
- SRM interface

The various Mass Storage Systems underlying a Storage Element will exhibit different errors, failures or latencies. These aspects of the Mass Storage System are not tested. Metadata catalogs that are application specific are not considered part of the core gLite middleware and are not tested. The SRM interface, to be adopted as an interface to all Storage Elements, is considered to be an external dependency and is not explicitly tested.

THE DISTRIBUTED TESTING INFRASTRUCTURE

A schematic of the distributed testing testbed, showing the VO specific and site specific gLite services to be deployed at each site is given in figure 1.

The gLite middleware is developed and tested on two platforms, referred to as the *main deployment platform* and the *secondary platform* respectively. The main deployment platform is any binary compatible version of Redhat Enterprise Linux 3.0 with gcc 3.2.3 and the secondary platform is Windows XP Server 2003 with icc8.0. The CERN site runs Scientific Linux CERN 3.0, NIKHEF runs CentOS 3.1 and RAL runs Scientific Linux 3.0. Such a combination of distributed platforms provides additional checks for dependencies on explicit versions of external packages of the middleware and the overall portability of gLite.

gLite middleware is extensively tested and validated on the main deployment platform. Middleware testing on the secondary platform is only to demonstrate that the middleware can be successfully ported to another platform.

Configurations of the gLite middleware, that reflect realistic deployment scenarios are identified, configured and tested. Multiple VO configurations, where two or more VOs are configured to use the same test environment are setup.

TEST METHODOLOGY AND PHASES

Middleware testing is performed in overlapping phases to avoid unmanaged ad-hoc and random testing. Entry and exit criteria are applied to each phase of the testing to determine whether the system is stable enough to proceed to the next phase of testing. Such criteria are essential to avoid wasting time by deploying unstable baselines on the testing testbed that have not passed basic tests.

Tagged baselines of the gLite code base produced by the weekly integration builds, in which all unit and integration tests are required to pass, form the basis for all regular testing and validation activities. gLite components are first tested on a component by component basis according to the priorities set in the EGEE middleware release plan, [7] and then the fully integrated system is tested.

The testing cycle from development through to integration and testing is shown in figure 2 together with the use of the Savannah bug report and tracking system.

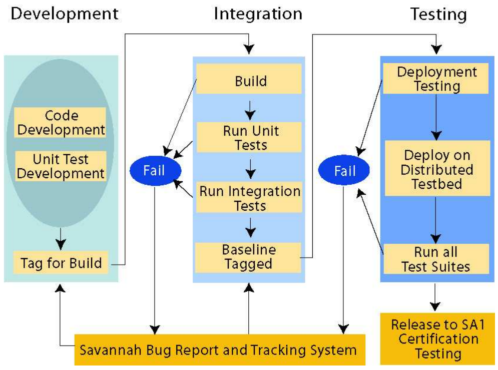

Figure 2: Testing cycle from development through to integration and testing.

Unit testing: Unit tests are written by the respective development clusters and run as part of the build system. All components must pass the associated unit tests to be accepted for further testing. In addition, code coverage tools are run by the build to ascertain what percentage of code has been covered by unit tests. Code that is not deemed to be sufficiently unit tested may not be accepted for further system testing.

Interface testing: The conformance of all interfaces to the agreed architectural specifications, given in [6], is verified by linking to libraries and associated dependencies during the build with a simple client that calls all functions. Binaries are tested by calling all exposed public interfaces. Erroneous output is parsed to determine if the error is due to non compliance with the APIs

Deployment testing: Installation and configuration testing is carried out on all system components without the use of an automated tool to ensure that installation is both correct and complete and that all installation documentation is correct. A service that can be started and responds to a ping request is considered to have passed the installation and configuration tests. If a ping operation is not supported some other test to verify that the service is alive is done.If successful, the full baseline is deployed across the distributed testing infrastructure.

Testing is carried out on all gLite packaging formats, namely RPMS for all binary compatible versions of Red Hat Enterprise Linux, source and binary tarballs and MSIs for Windows. The correct installation, location of files as defined in the developers guide [12], relocatability, uninstallation where appropriate, and consistency between the different packaging formats is tested.

Deployment testing is done on a single machine to ensure that all sources install and configure correctly and that the services run correctly.

All gLite components are required to be fully deployable on one single machine. This requirement is primarily to ensure that the installation of multiple services on one machine doesn't clash and that the number of machines required by a site to deploy gLite is not excessive.

A large number of different deployment configurations are possible for gLite components and services. We test various deployment scenarios in order to provide deployment recommendations for sites with limited numbers of machines, i.e. for a site having only three machines, which services should be deployed together on which machines.

Functionality testing Functional testing based on application requirements and use cases is carried out on both individual gLite components and the fully integrated system to validate gLite against system requirements.

Security testing: Focuses on issues of data access and resource access rights to Grid services and ensures that unprivileged users are not able to access or modify data. Biomedical application data is highly sensitive and tests ensure for example that members of one VO cannot access or modify data belonging to another VO, and that VOMS groups and roles provide the correct granularity of access to data.

The inclusion of security in a system often has adverse effects on a system's performance. The performance of the security enabled system will be measured and compared to the non security enabled system.

Resilience testing: Covering the areas of stability, error recovery, failover and stress testing as well as testing boundary conditions, such tests will identify conditions at which the system or individual components breakdown. Such tests include: Submit a job storm and measure the system response in terms of job wait time and success rate; Purposely bring down one of the system components needed by the current job only and check whether the whole system crashes and all jobs in the queue are lost, or if the current job is terminated nicely and the next job executed.

Performance testing: Ascertains whether the system or individual components meet the system performance requirements as stipulated by the user requirements, e.g. a maximum acceptable access time for a dataset for example or the volume of data or request rate that the system should be required to handle. In the absence of any performance requirements, the test team can make systematic performance measurements and present them for acceptance to the user community.

Portability testing: gLite components are deployed and tested on the secondary platform and tested as part of the whole gLite system in order to demonstrate the portability of the gLite middleware. Given that the gLite architecture is comprised of a set of independent services, the strategy for testing services running on the secondary platform is simply to configure gLite services to use certain other services running on the secondary platform. For example, to test WNs running on the secondary platform, we configure one CE to execute jobs on secondary platform WNs while leaving another CE to submit jobs to WNs running on the main deployment platform. For R-GMA we deploy servlet nodes at each site on the secondary platform and have them publish and consume information from an Information Catalogue running on the main deployment platform.

DEFECT REPORTING AND TRACKING

The Savannah bug reporting system provided by CERN is used for reporting and tracking defects found in the testing process. Figure 3 shows the process of tracking bugs.

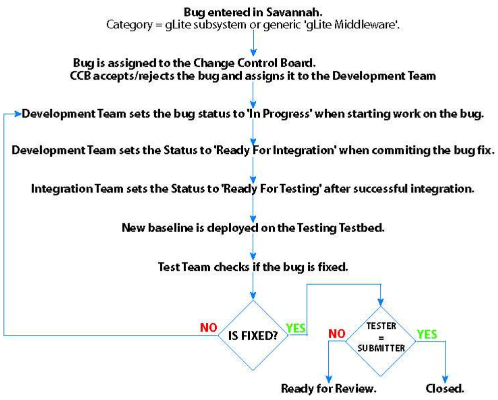

Figure 3: Life cycle of a gLite bug.

Bugs identified either by the test team or any other users are registered in Savannah specifying a severity and priority. The bug is assessed by the Change Control Board and if accepted is assigned to a development team. When a developer has fixed a bug, the bug status will be changed to *Ready for Integration* and committed. If the subsequent integration including unit and interface tests are successful, the bug status is changed to *Ready for Testing*. The new baseline is deployed on the distributed testing testbed and the fix is tested.

TEST REPORTING

We have defined XML Schemas for the output of all tests. The output format of some testing varies depending upon the test objective. Tools to convert the output of tests to HTML and PDF format have been written and incorporated into the testing process to automatically publish test results upon test completion on the JRA1 web site.

CONCLUSIONS AND FUTURE WORK

In this paper we have presented the testing infrastructure, scope and strategy for the gLite middleware as well as outlining the defect tracking and test reporting system.

We have successfully deployed a distributed testing testbed for the gLite middleware and are currently deploying and testing individual components as they are released into testing. we are developing test suites that will be publicly released.

We are currently investigating two automatic testing frameworks within which all gLite tests could be run. The advantages of such frameworks are that they can be used to set up functional test dependencies, establish a hierarchical structure or possibly facilitate testing Grid jobs that will run over a long period of time.

ACKNOWLEDGMENTS

We would like to acknowledge the contributions to the EGEE middleware testing activity of the following people: S. Burke, M. Barroso Lopez, M. Begin, O. Koeroo

We would also like to very gratefully acknowledge the assistance of Rosy Mondardini in producing the CHEP poster associated with this paper.

REFERENCES

[1] EGEE JRA1. EGEE Configuration Management Plan for EGEE Middleware, July 2004, https://edms.cern.ch/document/446241

[2] EGEE JRA1 Test Team. EGEE Software Testing and Validation, Middleware Test Plan , EGEE Milestone MJRA1.3, August 2004, https://edms.cern.ch/document/473264

[3] A. Di Meglio *et. al.* A Pattern-based Continuous Integration Framework for Distributed EGEE Grid Middleware Development , In *Proceedings CHEP04*, September 2004.

[4] E. Laure *et. al.* Middleware for the Next Generation Grid Infrastructure, In *Proceedings CHEP04*, September 2004.

[5] EGEE JRA1. EGEE Middleware Architecture. EU Deliverable DJRA1.1, July 2004, https://edms.cern.ch/document/476451

[6] EGEE JRA1. EGEE Middleware Design. EU Deliverable DJRA1.2, September 2004, https://edms.cern.ch/document/487871

[7] EGEE JRA1. EGEE Middleware release plan, September 2004, https://edms.cern.ch/document/468699

[8] F. Carminati *et. al.* HEPCAL Prime, Common Use Cases for a HEP Common Application Layer, http://cern.ch/project-lcg-gag/LCG_GAG_Docs/HEPCAL-prime.pdf

[9] EGEE JRA3. Security User Requirements, July 2004, https://edms.cern.ch/document/485295

[10] EGEE NA4. Requirements database, September 2004, http://egee-na4.ct.infn.it/requirements/

[11] EGEE SA1. EGEE SA1 Requirements, September 2004, https://edms.cern.ch/document/456865

[12] EGEE JRA1 Developers Guide, September 2004, https://edms.cern.ch/document/468700