Workload management in gLite 3.x - MPI

P. Nenkova, IPP-BAS, Sofia, Bulgaria

Some of materials are used from presentations of

Mike Mineter, Training Outreach and Education, University of Edinburgh, UK

www.eu-egee.org

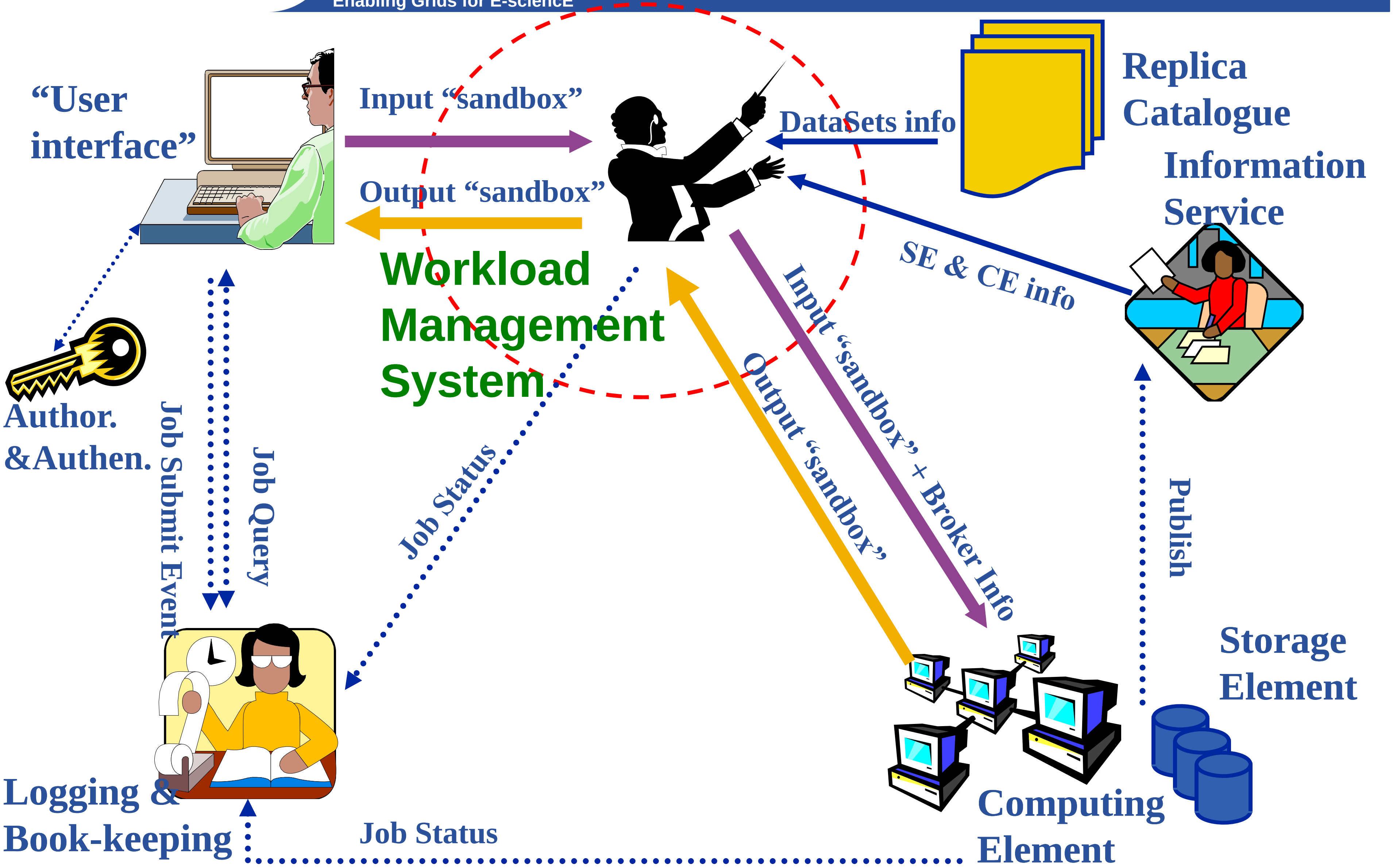
Workload Management System

- **Helps the user accessing computing resources**
  - resource brokering
  - management of input and output
  - management of complex workflows

- **Support for MPI job even if the file system is not shared between CE and Worker Nodes (WN) – easy JDL extensions**

- **Web Service interface via WMProxy**

gLite components



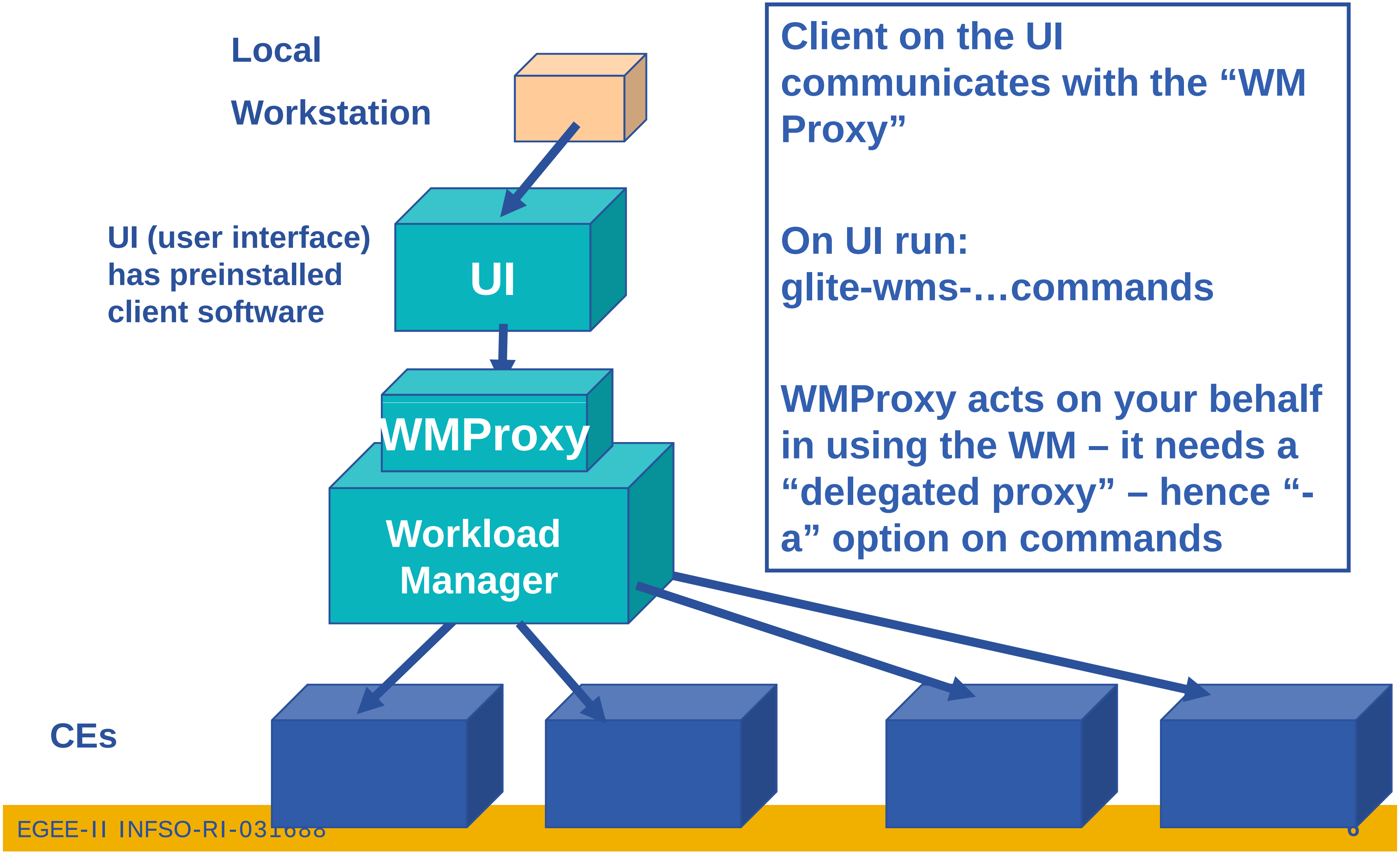
WMS functionality

- **The** Workload Management System **(WMS) is the gLite component responsible for the management of user's jobs : their**
  - submission
  - scheduling
  - execution
  - status monitoring
  - output retrieval
- **Its core component is the** Workload Manager **(**WM**)**
- **The** WM **handles the requests for job management coming from the WMS clients**
  - The submission request hands over the responsibility of the job to the **WM**.
    - **WM** will dispatch the job to an appropriate **Computing Element** for execution
      - *taking into account requirements and the preferences expressed in the job description (JDL file)*
- The choice of the best matching resource to be used is the outcome of the so called *match-making* process.

gLite WMProxy

- WMProxy (Workload Manager Proxy)
  - is a new service providing access to the gLite Workload Management System (WMS) functionality through a simple Web Services based interface.
  - has been designed to handle a large number of requests for job submission
    - gLite 1.5 => ~180 secs for 500 jobs
    - goal is to get in the short term to ~60 secs for 1000 jobs
  - it provides additional features such as *bulk submission* and the support for *shared and compressed* sandboxes for *compound jobs*.
  - It’s the natural replacement of the NS in the passage to the *SOA approach*.

WMS: role of WMProxy



Local

Workstation

UI (user interface) has preinstalled client software

CEs

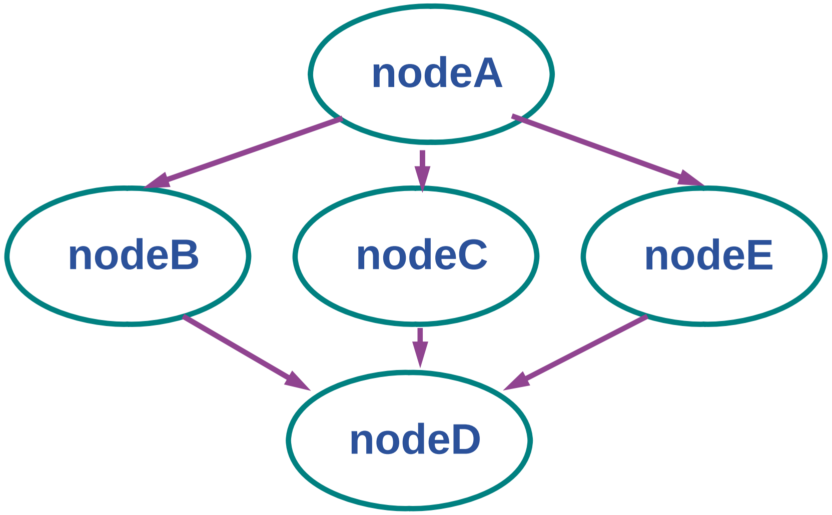
Client on the UI communicates with the “WM Proxy”

On UI run: glite-wms-…commands

WMProxy acts on your behalf in using the WM – it needs a “delegated proxy” – hence “-a” option on commands

Complex Workflows

- Direct Acyclic Graph (DAG) is a set of jobs where the input, output, or execution of one or more jobs depends on one or more other jobs



- A Collection is a group of jobs with no dependencies
  - basically a collection of JDL’s
- A Parametric job is a job having one or more attributes in the JDL that vary their values according to parameters
- Using compound jobs it is possible to have one shot submission of a (possibly very large, up to thousands) group of jobs
  - Submission time reduction
    - Single call to WMProxy server
    - Single Authentication and Authorization process
    - Sharing of files between jobs
  - Availability of both a single Job Id to manage the group as a whole and an Id for each single job in the group

WMProxy: users' view

- **glite-wms-job-submit will supersede glite-job-submit (which is superseding edg-job-submit)**
- **Its support for compound jobs will simplify application software**
  - WMProxy manages sub-jobs
  - Shared Input and Output "sandboxes"
- **MUST establish proxy delegation before this can be used!**

Submitting jobs to lcg-RBs

- Jobs run in batch mode on traditional gLite grids.
- Steps in running a job on a gLite grid with a lcg-RB:

1. Create a text file in “Job Description Language”
2. Optional check: list the compute elements that match your requirements (“edg-job-list-match myfile.jdl” command)
3. Submit the job ~ “edg-job-submit myfile.jdl”
   Non-blocking - Each job is given an id.
4. Occasionally check the status of your job (“edg-job-status” command)
5. When “Done” retrieve output (“edg-job-get-output” command)
6. Or just cancel the job (“edg-job-cancel” command)

Submitting jobs to a gLite WMS

- Jobs run in batch mode on traditional gLite grids.
- Steps in running a job on a gLite grid with WMS:

1. Create a text file in “Job Description Language”
2. Optional check: list the compute elements that match your requirements (“glite-wms-job-list-match myfile.jdl” command)
3. Submit the job ~ “glite-wms-job-submit myfile.jdl” Non-blocking - Each job is given an id.
4. Occasionally check the status of your job (“glite-wms-job-status” command)
5. When “Done” retrieve output (“glite-wms-job-output” command)
6. Or just cancel the job (“glite-wms-job-cancel” command)

Job management example

Delegate proxy and set delegation id

```
> glite-wms-job-delegate-proxy -d del_id
```

Submit job and save job id

```
> glite-wms-job-submit -d del_id -o job_id
  compress.jdl
```

Check job status

```
> glite-job-status -i job_id
```

Get job output when done

```
> glite-wms-job-output -i job_id
```

WMproxy: submitting a collection of jobs

- Place all JLDs to be submitted in a directory

  (for example ./Collect)

- voms-proxy-init --voms gilda
- glite-wms-job-delegate-proxy –d DelegString
- glite-job-submit –d DelegString –o myJIDs --collection ./Collect
- glite-wms-job-status -i myJIDs
- glite-wms-job-output –i myJIDs

WMS - user interaction: JDL file

- **Write your Job Description File (JDL file)   (classAds)**

```
Type = “Job”;
JobType = “Normal”;
Executable = “/bin/bash”;
Arguments = “mySimulationShellScritp.sh”;
StdInput = “stdin”;
StdOutput = “stdout”;
StdError =  “stderr”;
InputSandbox = {“mySimulationShellScritp.sh”,“stdin”,“data-card-
    1.file”,”data-card-2.file”};
OutputSandbox = {“stderr”, “stdout”,“outputfile1.data”,”histos.zebra”};
Environment = {“JOB_LOG_FILE=/tmp/myJob.log”};
Requirements = Member(“EGEE-preprod-1.2.4-
    1.2”,other.GlueHostApplicationSoftwareRunTimeEnvironment);
```

Workload Manager Service

- The JDL allows the description of the following request **type**s supported by the WMS:
  - **Job: a simple application**
  - **DAG: a direct acyclic graph of dependent jobs**
    - **With WMSProxy**
  - **Collection: a set of independent jobs**
    - **With WMSProxy**

E.g. Type = "Collection"

Some JDL-file attributes

- `Executable` – sets the name of the executable file;
- `Arguments` – command line arguments of the program;
- `StdOutput, StdError` - files for storing the standard output and error messages output;
- `InputSandbox` – set of input files needed by the program, including the executable;
- `OutputSandbox` – set of output files which will be written during the execution, including standard output and standard error output; these are sent from the CE to the WMS for you to retrieve
- `ShallowRetryCount` – in case of grid error, retry job this many times (“Shallow”: before job is running)

JDL: Relevant Attributes

- JobType (optional)
  - o *Normal* (simple, sequential job), *Interactive*, *MPICH*, *Checkpointable*, *Partitionable, Parametric*
  - o Or combination of them
    - ✓Checkpointable, Interactive
    - ✓Checkpointable, MPI

E.g. JobType = “Normal”;

JDL: Relevant Attributes (cont.)

- **Executable (mandatory)**
  - This is a string representing the executable/command name.
  - The user can specify an executable which is already on the remote CE
  - Executable = {“/opt/EGEODE/GCT/egeode.sh”};

  - The user can provide a local executable name, which will be staged from the UI to the WN.
  - Executable = {“egeode.sh”};
  - InputSandbox = {“/home/larocca/egeode/egeode.sh”};

JDL: Relevant Attributes (cont.)

- **Arguments (optional)**

  This is a string containing all the job command line arguments.

E.g.: If your executable sum has to be started as:

```
$ sum N1 N2 –out result.out
```

```
Executable = “sum”;
Arguments = “N1 N2 –out result.out”;
```

JDL: Relevant Attributes (cont.)

- **Environment (optional)**
  - List of environment settings needed by the job to run properly

    E.g. Environment = {“JAVA_HOME=/usr/java/j2sdk1.4.2_08”};

- **InputSandbox (optional)**
  - List of files on the UI local disk needed by the job for proper running
  - The listed files will be automatically staged to the remote resource

  E.g. InputSandbox ={“myscript.sh”,”/tmp/cc.sh”};

- **OutputSandbox (optional)**
  - List of files, generated by the job, which have to be retrieved from the CE
  - E.g. OutputSandbox ={ “std.out”,”std.err”, “image.png”};

JDL: Relevant Attributes (cont.)

- **Requirements** **(optional)**
  - Job requirements on computing resources
  - Specified using attributes of resources published in the Information Service
  - If not specified, default value defined in UI configuration file is considered

    Default. Requirements = *other.GlueCEStateStatus == "Production“;*

    Requirements=other.GlueCEUniqueID == “adc006.cern.ch:2119/jobmanager-pbs-infinite”

  - Requirements=Member(“ALICE-3.07.01”, other.GlueHostApplicationSoftwareRunTimeEnvironment);
  - Requirements = Member("MPICH", other.GlueHostApplicationSoftwareRunTimeEnvironment);

MPI Applications in Grid Environment

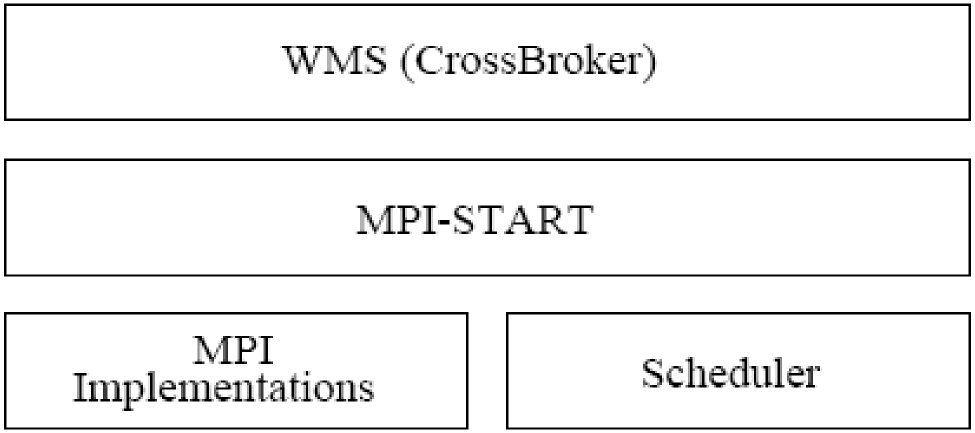
- The current release of the EGEE middleware (gLite 3.0.1) provides the MPICH job type to support MPI applications. Users wishing to submit MPI jobs set the JobType variable to “MPICH” and set the variable “NodeNumber” to indicate how many nodes they require:

```
JobType = "MPICH";
NodeNumber = 8;
Executable = "mpi-start-wrapper.sh";
Arguments = "mpi-test MPICH";
InputSandbox = {"mpi-start-wrapper.sh","mpi-hooks.sh","mpi-test.c"};
Requirements = Member("MPICH", other.GlueHostApplicationSoftwareRunTimeEnvironment);
```

- Although it is not required to compile the MPI locally, it is highly recommended. Many compilation options are specific to the software installed or hardware installed on a site.

MPI-start

- MPI-start, a script developed within the int.eu.grid project provides a layer of abstraction between the user and the various different implementations of MPI.
- Detects the configuration of a site at runtime and runs the user’s executable using the requested version of MPI



Hide differences of MPI implementations

Hide differences between scheduler implementation

Support for different schedulers

Support for different MPI implementations

Support for simple file distribution

MPI-Start Abstraction Layer

Wrapper script (mpi-start)

- **In order to hide some of the complexity, wrapper script for submitting MPI job is used.**
- **Requires the user to define a "wrapper script" and a set of "hooks“:**

```
# Setup for mpi-start.
export I2G_MPI_APP=$MY_EXECUTABLE
export I2G_MPI_TYPE=$MPI_FLAVOUR
export I2G_MPI_PRE_RUN_HOOK=mpi-hooks.sh
export I2G_MPI_POST_RUN_HOOK=mpi-hooks.sh
# Invoke mpi-start.
$I2G_MPI_START
```

  – "pre-hook" can be used to compile the executable itself or download data.
  – "post-hook" can be used to analyze results or to save the results on the grid

MPI job submission

```
JobType = "MPICH";
NodeNumber = 8;
Executable = "mpi-start-wrapper.sh";
Arguments = "mpi-test OPENMPI";
InputSandbox = {"mpi-start-wrapper.sh","mpi-hooks.sh","mpi-
test.c"};
Requirements = Member("OPENMPI",
other.GlueHostApplicationSoftwareRunTimeEnvironment);
```

JDL

```
 # Setup for mpi-start.
 export I2G_MPI_APP=$MY_EXECUTABLE # ($1)
 export I2G_MPI_TYPE=$MPI_FLAVOUR  # ($2)
 export I2G_MPI_PRE_RUN_HOOK=mpi-hooks.sh
 export I2G_MPI_POST_RUN_HOOK=mpi-hooks.sh
 # Invoke mpi-start.
 $I2G_MPI_START
```

wrapper

```
pre_run_hook () {
mpicc -o ${I2G_MPI_APP} ${I2G_MPI_APP}.c
}
```

hooks

More information

- gLite 3.0 User Guide
  - https://edms.cern.ch/file/722398/1.1/gLite-3-UserGuide.pdf
- GLUE Schema
  - http://infnforge.cnaf.infn.it/glueinfomodel/
- JDL attributes specification for WM proxy
  - https://edms.cern.ch/document/590869/1
- WMProxy quickstart
  - http://egee-jra1-wm.mi.infn.it/egee-jra1-wm/wmproxy_client_quickstart.shtml
- WMS user guides
  - https://edms.cern.ch/document/572489/1