# PWE2008

## NCSA Workflow Management Infrastructure (as of October 21, 2009)

**The NCSA Workflow Management Infrastructure** is an end-to-end system for generating, editing, submitting and monitoring high-performance computing [HPC] workflows which have the following characteristics:

- each node in the workflow graph comprises a payload, or script, which is to be executed, usually by being submitted to a Distributed Resource Manager [DRM], or batch system; the script can be given input values and can return (small) output values as well;
- the graph itself is directed and acyclic, meaning there are no conditionals or loops at this level (however, conditional or repeated submissions of a workflow with varying values can be achieved through the use of the Trigger service, described below);
- any node or subgraph in the workflow could be subject to "parameterization", meaning multiple submissions with varying input values.

The infrastructure consists of:

1. a front-end desktop client, **Siege**;
2. a **Parameterized Workflow Engine [PWE]**, responsible for workflow management;
3. information and data transfer services (**Host Information**, **Event Repository**, **Tuple Space [VIZIER]**);
4. a transient, compute-resource-resident container for running the payload scripts (**ELF**);
5. a service for triggering actions, most typically submissions of workflows to PWE, on the basis of events or as cron jobs;
6. a message bus.

The following diagram schematizes these components:

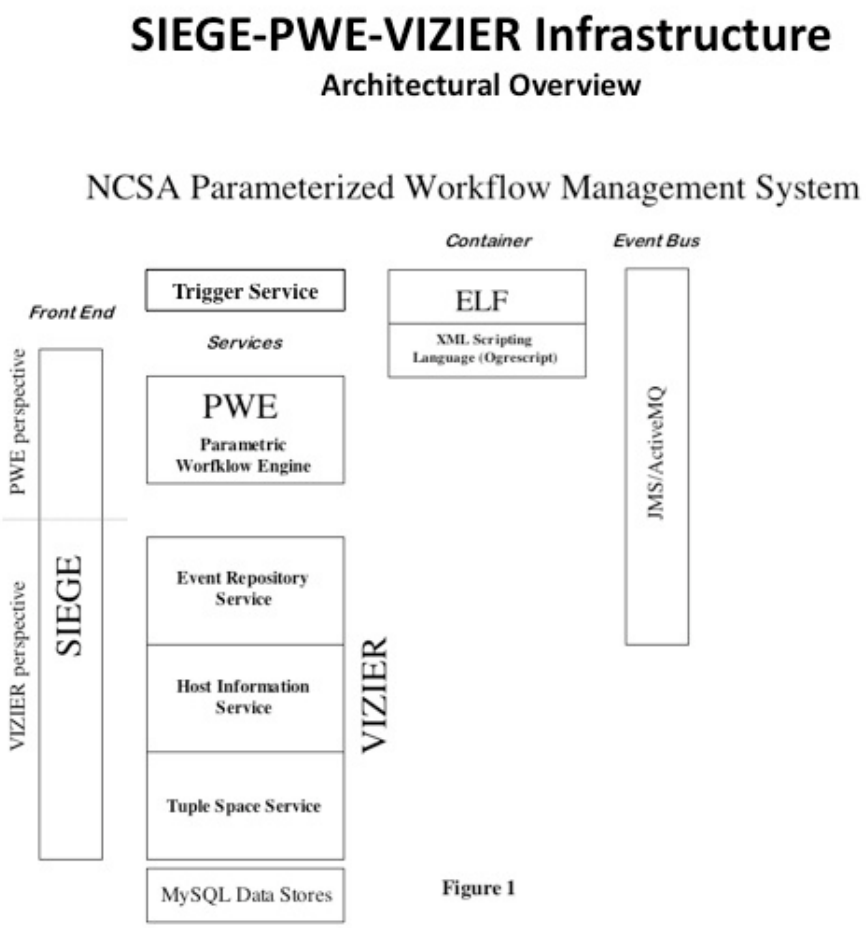



Figure 1

The most powerful feature of this system is its ability to launch efficiently through batch systems very wide parameterized workflows - that is, where a single node may be replicated hundreds or even thousands of times (one current user has been submitting workflows with a graph consisting of 1 + 700 + 1400 + 700 + 2 + 1 nodes). The design objective in this regard is twofold:

1. avoid unnecessary overhead/latency costs in the submission;
2. achieve scalability by offloading the control of each set of nodes from PWE onto the resource-resident container, ELF.

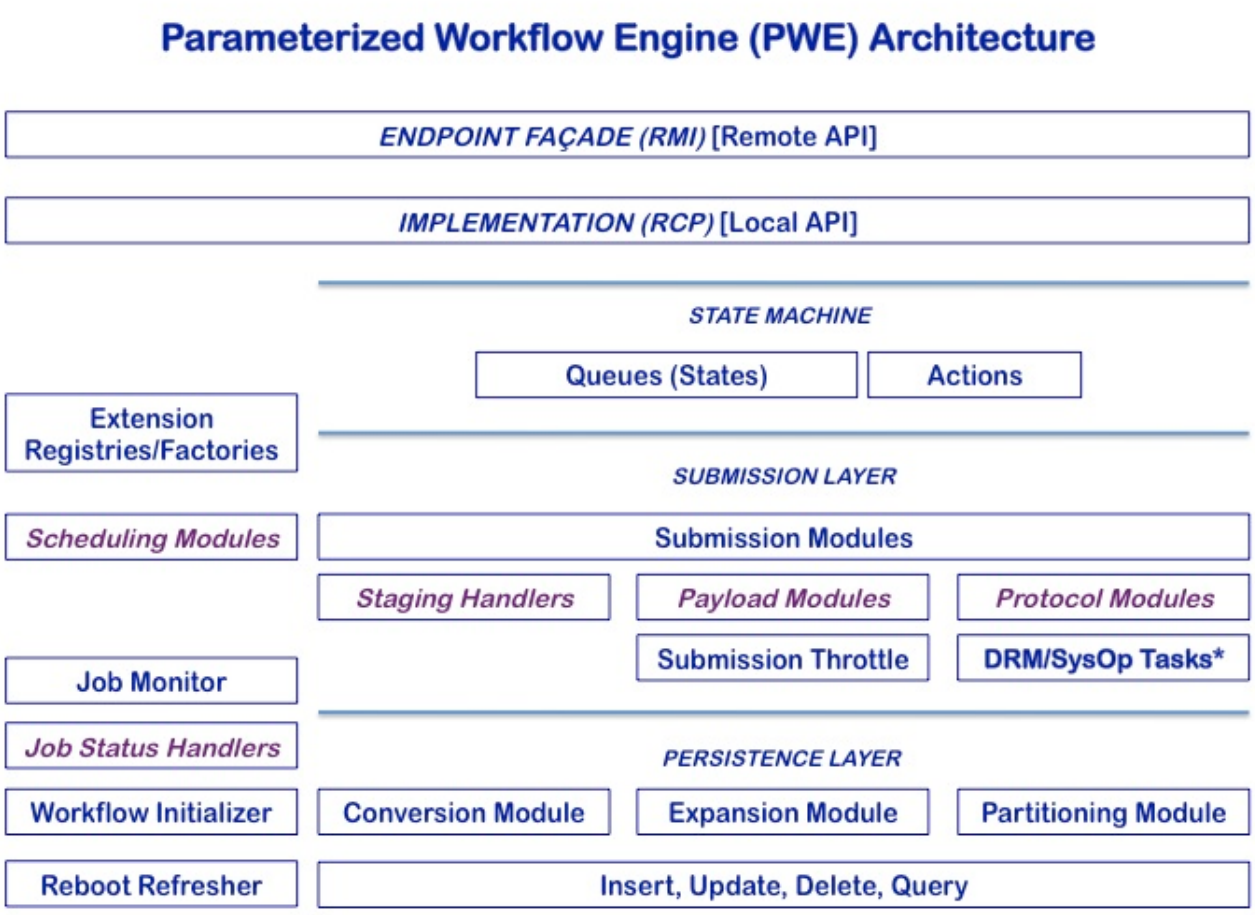
To accomplish this, the specifics of the host platform and batch system must, of course, be taken into account. On a distributed-memory cluster (such as Abe or Mercury here at NCSA), a single submission to the batch system brings up the ELF container, and this container in turn uses SSH to launch the parameterized scripts onto the compute nodes it knows about (from a PBS node-file); on an SMP-like machine (such as Cobalt), on the other hand, the single ELF container relies on *dplace* to distribute the work among cpus (a fuller illustration of this sequence of operations is given below in section §3). We have called this management pattern ***glide-in***, a term we have adopted from Condor (where it has, however, a somewhat different meaning).

---

## Detailed Discussion of Implementation, with addition of modules to support AIX/LoadLeveler.

### 1. Additions to PWE (Parameterized Workflow Engine)

This is the main service in our infrastructure, responsible for the management of workflows through their entire lifecycle, from submission through completion. A full description of this service is not included here; the following architectural diagram is provided merely to be able to situate the extensions we have provided.




The top two layers typify all the services in our infrastructure: a remote (Java-RMI) endpoint which embeds a headless RCP (Eclipse) implementation. Since the core layer uses the Eclipse RCP extension mechanism (itself implemented as OSGi-Equinox bundles), we have exposed as extension points the components which depend on the specifics of external operating systems, resource managers or types of payload (scripting language): the Java classes so supported are indicated in violet.

Anticipating that the principal deployment of PWE at NCSA will have to run against Linux, PBS, AIX and LoadLeveler (as well as perhaps other systems), we saw no immediate reason to organize the core's plug-ins according to a system-oriented typology; hence the IBM-specific extensions have been directly incorporated into the current plug-in structure. If, however, in the future the need for system-specific distributions arises, it would be very easy to refactor these pieces into such a set of plug-ins.

Moreover, it is entirely possible to provide new plug-ins to PWE containing the implementation of all necessary modules for a given OS, DRM or scripting language without requiring the restructuring of the core. For instance, if one wanted to be able to run Python scripts directly (rather than using ELF, our script container), one could write a plug-in implementing the PayloadModule extension point; for interactive submissions to a MacOS or Solaris box, one could write a plug-in implementing the ProtocolModule and JobStatusHandler extension points, and so forth. In the first case, the workflow description submitted to the service would specify the payload script as Python; in the second case, a property in the Host Information service (see the first diagram above) would map the resource to a submission and polling contact whose labels would include the tag of the protocol extension to use. To enable these extensions, the deployment of PWE would merely need to include these additional plug-ins in its plug-in set.

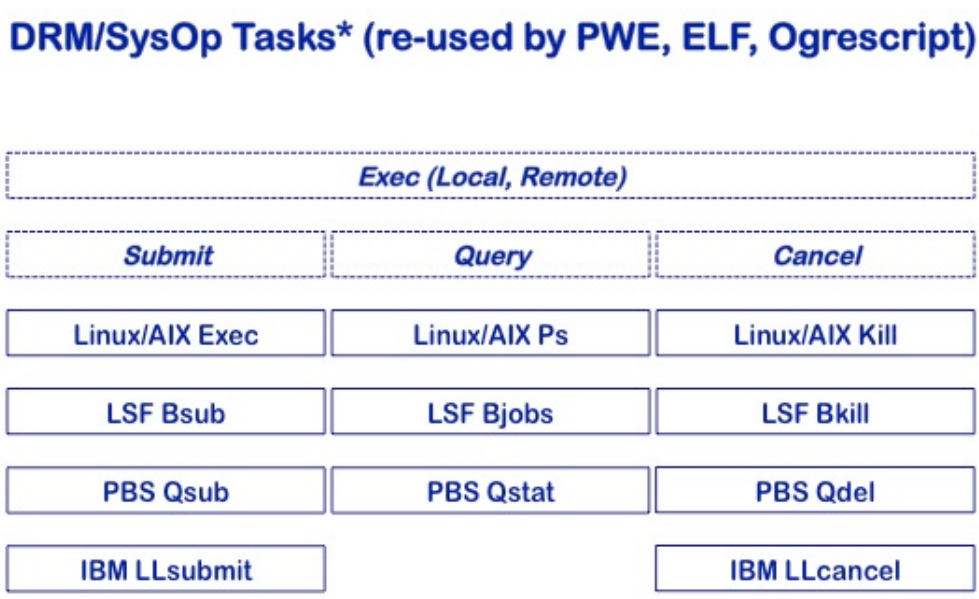
We added the following extensions to PWE's core plug-ins:

1. ***Submission Protocol Module***
    a. AIX (Interactive) - for launching (via [GSI]SSH) directly onto the head node of an AIX machine or cluster (very specialized use, usually for preprocessing actions which are short-lived);
    b. IBM Loadlevler - for submitting (again via [GSI]SSH) to the LL Scheduler.
2. ***Job Status Handler***
    a. AIX Polling Handler - for monitoring the status of interactively submitted jobs (polls the head node via [GSI]SSH using 'ps');
    b. IBM Loadlevler Job Status Handler - this is just a façade for receiving the callback events sent by the User Exit Trigger (see below).

Both the modules and the handlers are very light-weight child classes of a parent containing most of the necessary functionality. In the case of the modules, the extension merely defines which implementation of the *Submit* task (see below) to use.

## 2. Additions to DRM/SysOp Tasks

These embody the mechanism whereby our RCP applications communicate remotely with interactive resources, or locally and remotely with distributed resource managers. The tasks are shared by various components in the system -- PWE's protocol modules, the ELF Agent's member access classes and *Ogrescript*'s Batch tasks (see below) --, and currently reside in a single plug-in. The Java class hierarchy defining them is as follows:




The interactive Linux tasks were generic enough for reuse with AIX, so separate implementations were not necessary. The newly added tasks are:

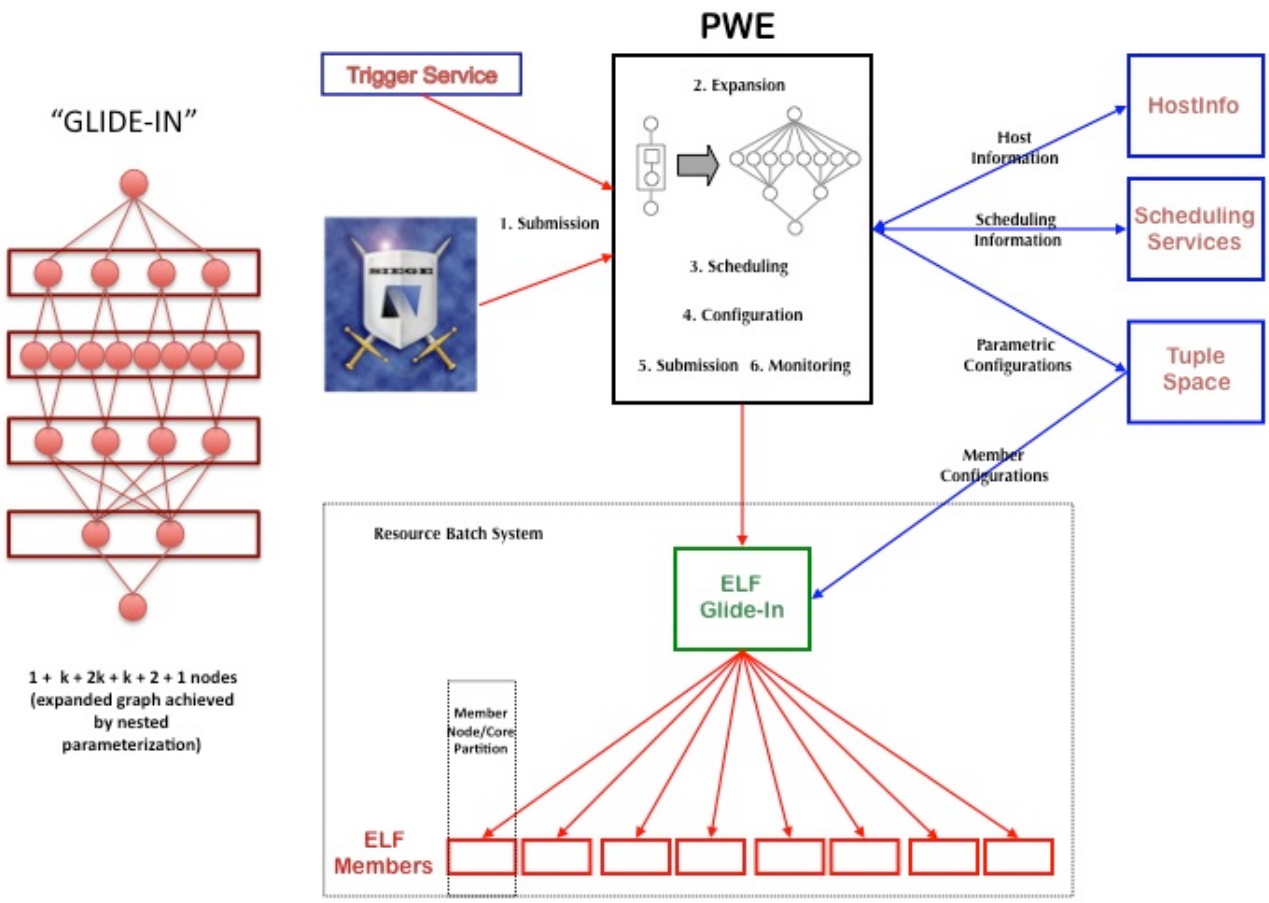
- ***IBM LLsubmit***
- ***IBM LLcancel***

The brunt of the adaptation effort required to make PWE interact with a new batch system, in fact, consists largely in implementing the above. In particular, the *Submit* task is responsible for converting PWE's internal resource request / job definition into a command to be issued to the batch system, usually in the form of an `echo` statement containing the script contents (including initial environment variables) piped to the batch command resident on the host, and followed by arguments specific to that system. In the case of LoadLeveler, this command is issued to our native wrapper (*llsubmit_ncsa*; see below), which, among other things, turns it into a full-fledged batch script (this is not necessary, for instance, on PBS systems, where resource requirements, account, etc., can be specified via command-line flags).

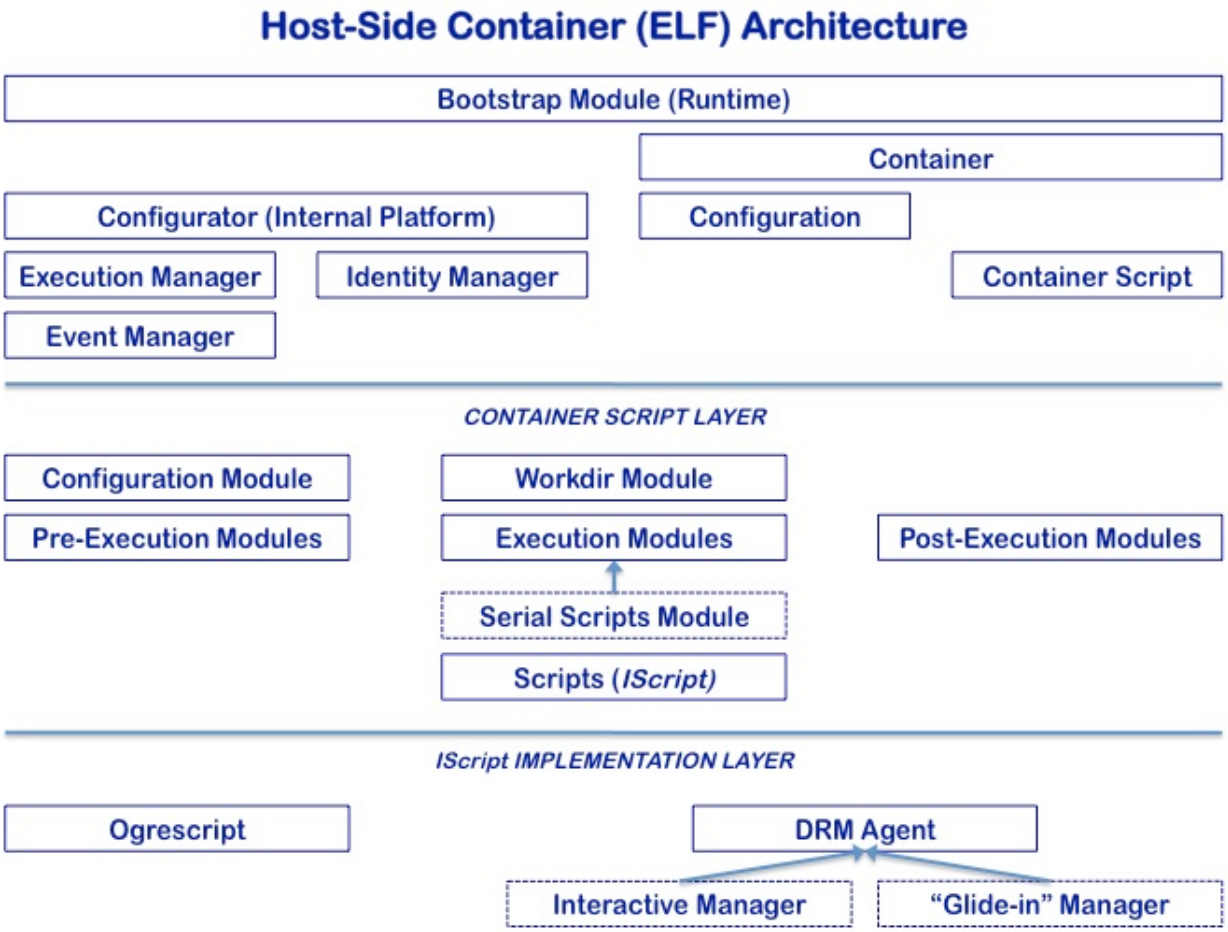
In implementing the *Submit* module, we have adopted an agile approach, exposing only the most-used features of the batch API; we await fuller user requirements to drive any ulterior refactoring which would make fuller access to all features available through PWE.

## 3. Additions to Host-Resident Script Container (ELF)

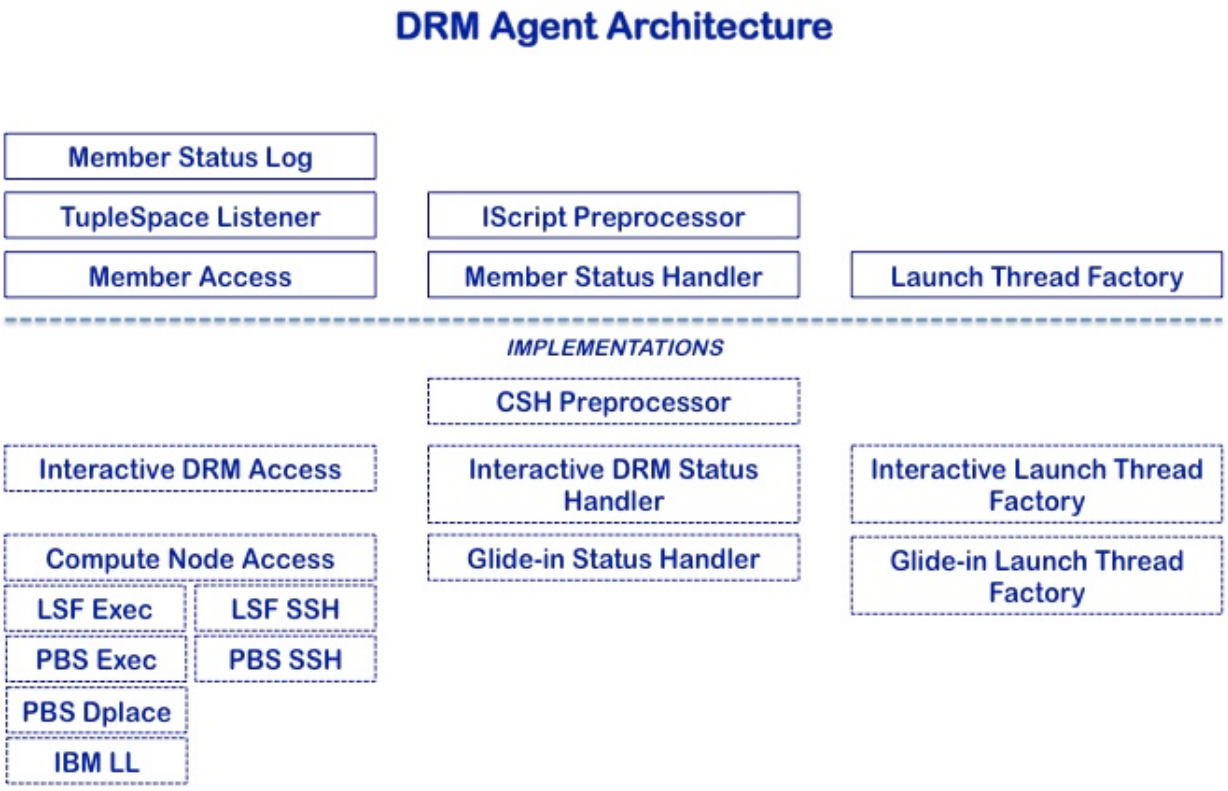
Where the back-end architecture allows for the execution of Java processes, the preferred way of running each node in the workflow graph (a node consisting of a payload, or script, along with initial properties, input and required output, if any) is through our Eclipse headless RCP application, ELF. For singleton nodes in the graph (that is, no duplication through parameterization), the ELF wrapper serves as a "container" whereby the execution of the script is set up and monitored, and failures or success are reported back to PWE. Where parameterization is involved, a second kind of ELF wrapper, referred to here as the DRM Agent, intervenes to handle, host-side, the actual parameterization, with each member then having its own singleton ELF wrapper. This is illustrated at the bottom of the following diagram.




These two ELF "wrappers" share the same basic architecture; the difference is at the level of the embedded script itself; singleton ELF (usually) runs our XML scripting language, *Ogrescript*, whose base class implements the ELF-internal interface *IScript*, whereas the DRM Agent, which provides the additional layer of container functionality, is simply a distinct implementation of the same interface:




There are actually two kinds of DRM Agent, as indicated above; the *Interactive Manager* is akin to the pre-WS Globus *job-manager*, in that it is meant to run on a head node of a system where back-end Java is unavailable, and monitors arbitrary jobs (not necessarily ELF) put through the resident batch system. Our concern here, however, is with the second type of manager, which is itself a job (step) submitted to the scheduler or resource manager, and which, when it becomes active, distributes the work among the given number of ELF singleton members. Similar to PWE internals, this agent has a system-specific component, as can be seen from the following diagram of the agent's architecture.




We thus needed to provide an implementation of *Member Access* for LoadLeveler (= ***IBM LL Access***). Once again, this child class relies heavily on its parent for much of the basic functionality, but in the case of the LoadLeveler extension, there were some peculiarities with the job set-up for which it is responsible, deriving from the fact that our mode of interaction with LoadLeveler is to resubmit through the scheduler using a reservation id, rather than to control the distribution of members among the resources made available to the glide-in job, as we do in the case of LSF or PBS (using either *exec* or *dplace* on [pseudo-]SMP machines or SSH to the compute nodes in the case of distributed memory). The access module is reponsible for

- creating a "nodefile" from the queue of available cores/cpus
- generating the necessary `bootstrap.properties` and `container.xml` files for ELF to run the member script
- generating a command file (*bash*) to be used in conjunction with member launch
- releasing the resources when the member has completed
- issuing an appropriate *Cancel* command when necessary

In the case of LoadLeveler, however, the command file is bypassed, and a *Job* object is created, much in the way PWE handles submission; the *DRM /SysOp* task ***LLsubmit*** is re-used here, and the command it generates is issued locally to the *llsubmit_ncsa* wrapper around LoadLeveler's *llapi.h*. Resource accounting ends up being for internal purposes only.

### 4. Host-Resident Platform Support

In addition to the extensions provided for by the PWE-ELF architecture, two other components also had to be implemented. This code resides on the head node and is called by PWE and LoadLeveler, respectively.

### 4.1 *llsubmit_ncsa.c*

This native C wrapper around LoadLeveler's API serves two purposes:

1. To translate the command issued by PWE via [GSI]SSH into a job script;
2. To provide to LoadLeveler the arguments necessary for reporting job state back to PWE (i.e., the "User Exit" parameters in the *llsubmit* signature).

As previously mentioned, the command-line for batch submission produced by the protocol module / submit task takes the form of an echo piped to the batch system command executable. For LoadLeveler, this looks like:

```
echo " ... ; ... ... ;" | llsubmit_ncsa 'callback_uri' llsub_arg0 llsub_arg1 ...
```

When *llsubmit_ncsa* is invoked with this command, the following steps are taken:

- a job command file name is generated based on the callback_uri;

For example:

`rmi://otfrid.ncsa.uiuc.edu:1099/pwe?groupId=pweIntegrationTests&level=DEBUG&node=run&user=/C=US/ONational_Center_for_Supercomputing_Applications/OU_People/CN=Albert_L._Rossi&workflow=test-singleton-hg-2845851136136799036`

contains the location of the service to call back with state updates, but also unique identifiers for the worklow; these identifiers are used to generate the command file name - in this case, "`test-singleton-hg-2845851136136799036-run`".

- the job command file is created with this name;
- the command-line arguments are translated into LoadLeveler's job properties pragmas (using the '# @ property = value' syntax; the current environment is also passed on to the job step using '# @ environment=COPY_ALL');
- the stdin redirected to this wrapper is written to the file as its contents;
- a properties file is created using the job command name as prefix; this file contains arguments necessary for the callback trigger;
- *llsubmit* is called using the job command file, the path to the trigger script (see below), and the properties file as arguments;
- if the call is successful, the job step id is returned to PWE on stdout.

### 4.2 PWE Notification Agent [= LLUserExitTrigger]

This is the Java trigger called by LoadLeveler to report edge events ("User Exit"). It is implemented as another RCP headless application. Though designed specifically for this purpose, we have nonetheless made the LoadLeveler implementation a concrete instance of an abstract class, *PweNotificationAgent*.

Just as the ELF executable on the host resides in the area defined by the ELF_HOME property or environment variable, so the script which launches this trigger, *llexit_ncsa*, resides in the area designated LL_SUPPORT (as does the *llsubmit_ncsa* executable). LoadLeveler thus calls the "User Exit" as follows:

```
$LL_SUPPORT_HOME/llexit_ncsa job_id properties_file job_state job_exit_status
```

The underlying Java code is extremely simple: it uses the properties file to construct a callback client to PWE, then generates a STATUS event using the `job_id, job_state` and `job_exit_status`, and sends it to PWE, exiting when the event has been received. This is essentially the same mechanism utilized by ELF to call back to PWE, but the latter can distinguish between the sources of the two events inasmuch as their origin is identified through a 'producer' header attribute. PWE prioritizes ELF events over edge events, but in the absence of the former, will rely on the latter to indicate what steps to take next in the processing of the workflow.

## 5. Refactoring of the Job Description

Having to take into account LoadLeveler's job properties API forced us to reconsider the way we had been specifying jobs in the workflow description schema. We believe this to have been a fruitful exercise, as there had been much dead weight accreted from various extensions, first to the Globus RSL properties, then to the SSS Job schema. As a consequence, we decided to do a major refactoring of our Job description along lines which seemed more rational to us, and which seemed to generalize better over the several languages of the DRMs we must interact with. This resulted in the production of our own Job .xsd, a number of changes to the `<scheduling>` properties in the workflow, and the rebuilding of five plug-ins.

The Job XML Schema is provided as an attachment for the sake of completeness; more important is the specification of the scheduling property names which may be used in the workflow description:

```
<!-- ********************************************************************** -->
<!--      These are the properties available to be set in a profile         -->
<!--      used for <scheduling> in the workflow description.                -->
<!-- ********************************************************************** -->
<profile>
        <!-- REQUIRED -->
        <property name="submissionType" type="string" /><!-- batch / interactive -->

        <!-- REQUIRED for batch submission types-->
        <property name="account" type="string" />
        <property name="maxWallTime" type="long" />

        <!--
             OPTIONAL
             - if cpus are not set, they will (both) default to the number of cpus
```

```
       needed to satisfy all members (=1 for non-parametric workflow nodes)
     - ranks is the MPI parallelism, on a member-by-member basis (default = 1)
     - threads is number of threads/cpus assignable to each rank (default = 1)

     For non-mpi workflow nodes to which you wish to assign k > 1
     cpu (per member), either ranks or threads can be set to k (though
     it perhaps makes more sense to view this as a case of rank = 1,
     threads = k); in any case, on non-LoadLeveler systems,
     only the total cpus per member has any meaning
-->
<property name="maxCpus" type="int" />
<property name="minCpus" type="int" />
<property name="ranksPerMember" type="int" />
<property name="threadsPerRank" type="int" />

<!-- OPTIONAL, defaults 'std[].log' in the initial directory -->
<property name="stdout" type="string" />
<property name="stderr" type="string" />

<!-- OPTIONAL, defaults to parameterized member count -->
<property name="maxMembers" type="int" />

<!-- OPTIONAL, defaults to no later than submission time + 30 minutes,
          where applicable; date format: yyyy/MM/dd HH:mm:ss -->
<property name="requestedStart" type="string" />

<!--
     OPTIONAL, defaults to number of cores per node on scheduled machine
     example rule:   ABE:7,BLUEPRINT:16, where the name corresponds
     to HostInfo key for the machine, and the number is cores
     per node to be used
-->
<property name="coreUsageRule" type="string" />

<!--
     OPTIONAL, defaults to true
     indicates that unused cores on a node allocated to this
     submission should be used if possible; i.e., if you are
     running 4 16-wide members on nodes with 32 cores, you'd ideally
     like to run on 2 nodes, not 4; set to false if you
     want to enforce only 16 per node (for LL).
-->
<property name="shareNodes" type="boolean"/>

<!-- OPTIONAL, defaults to maxWallTime -->
<property name="minWallTime" type="long" />
<property name="maxWallTimePerMember" type="long" />
<property name="minWallTimePerMember" type="long" />

<!-- OPTIONAL, defaults to <null>, false or undefined -->
<property name="queue" type="string" />
<property name="stdin" type="string" />
<property name="maxTotalMemory" type="long" />
<property name="minTotalMemory" type="long" />
<property name="dplace-trace" type="boolean" />
<property name="nodeAttributes" type="string" />

<!-- OPTIONAL, currently unimplemented -->
<property name="maxCpuTime" type="long" />
<property name="minCpuTime" type="long" />
<property name="maxCpuTimePerMember" type="long" />
<property name="minCpuTimePerMember" type="long" />
<property name="maxMemoryPerNode" type="long" />
<property name="minMemoryPerNode" type="long" />
<property name="maxSwap" type="long" />
<property name="minSwap" type="long" />
<property name="maxDisk" type="long" />
<property name="minDisk" type="long" />
<property name="maxBandwidth" type="long" />
<property name="minMandwidth" type="long" />
<property name="maxOpenFiles" type="long" />
```

```
        <property name="maxStackSize" type="long" />
        <property name="maxDataSegmentSize" type="long" />
        <property name="maxCoreFileSize" type="long" />
        <property name="checkpointable" type="boolean" />
        <property name="suspendable" type="boolean" />
        <property name="restartable" type="boolean" />
        <property name="priority" type="int" />
</profile>
```

## 6. Host Information Service

In our infrastructure, available resources must be mapped in our Host Information Service, which records user names and homes, node names, protocols to use when submitting, querying, or moving files, along with any other properties or environment variables necessary for the proper execution of our codes.

Along with the appropriate mapping for Blueprint, which includes the LL_SUPPORT_HOME path, we have also added a SHARE_NODES property to all resources so that the proper default behavior will be enforced ('true' for SMP-like resources, otherwise 'false').

## 7. PWE Scheduling & Reservation Request Modules

PWE currently has a slot for scheduling modules, but uses a NOP module which requires the user to specify the resource(s) on which to run. In essence, there are two aspects to "scheduling" inside PWE:

1. determine the resource(s) on which to run;
2. procure a reservation on those resources, if possible.

These two aspects should ideally be coupled in a kind of feed-back loop; in the current state of development, however, we do not yet have a clear idea for the more general design, particularly of the first function, which is of course the harder issue to solve.

In keeping with agile practices, our immediate approach will be to map to each host a scheduling protocol key indicating to PWE how to go about requesting a reservation (point 2 only), and implement only that feature for the moment. In the case of other systems, this would continue to be a NOP, whereas for Bluewaters/Blueprint we would use the LoadLeveler API to procure a reservation id to be passed on and included in the appropriate job step (s) submitted to the scheduler.

NOTE: these capabilities have now been implemented. See Siege_3.0 for a guide to their usage.

---

10/21/2009, by Albert L. Rossi, Principal Architect and Developer