Workflow Broker

Purpose

The Workflow Broker (formerly "Ensemble Broker") is a GSI web service designed to orchestrate and manage complex sequences of large computational jobs on production resources.

Features

- Expressive (XML) description logic which allows for
  - Structuring workflow in two independent layers:
    - Top level (broker-centric): a directed acyclical graph of executable nodes;
    - Individual node payloads: scripts to be run on the production resource, usually inside an Elf container;
  - Configuration of scheduling and execution properties on a node-by-node basis;
- Integration with scheduling systems, such as MOAB;
- (Potentially recursive) service-side parameterized expansion of nodes (done lazily, as the nodes become ready to run);
- Fully asynchronous staging, submission and monitoring of jobs;
- State management through a persistent object-relational layer (Hibernate);
- Inspection of workflow state enabled via event bus as well as API calls on the borker service itself;
- Cancellation and restart of nodes or entire workflows.

The broker is designed to be extensible with respect to

- Type of node payload (currently ELF/Ogrescript and C-Shell are available);
- Type of launching and monitoring protocols used (currently pre-WS GRAM, GSI-SSH to the native batch system, and local submission to a MOAB peer are available).

These extensions usually only involve implementing a special module type.

Currently, the broker can handle hundreds of simultaneous submissions to computational resources; we are working towards the necessary modifications which would enable scaling up to thousands of such submissions.

Comparison with alternative systems

Batch-system workflow

Some batch systems, such as IBM's Loadleveler, allow for simple workflow capabilities at the job dependency level. But the process for submission is command-line oriented, making the submission of multiple-node or ensemble-like workflows difficult (a complex script would probably be required, one which would in turn suffer from the need for continual modification). Monitoring the jobs, if not done by the usual manual inspection of the batch queue, would also require an *ad hoc* script to be written.

Condor

Condor has very nice scalability, especially when managing a Condor pool (usually of single processor systems). It also has an expressive language for describing DAGs of jobs to be executed remotely. In its Condor-G flavor, though – which opens the (necessary) door to parallel jobs – , it suffers from scalability and reliability concerns that arise from using GRAM, and which our broker can circumvent by virtue of its native-batch-system protocol option. Also, since Condor does not automatically provide a cluster-local component (i.e., our script container), it is not possible – in the absence of *ad hoc* scripting and logging mechanisms – , to obtain finer grained monitoring of remote jobs than what the resource manager and GRAM tell you about them.

Neither of these alternatives has a readily available, generalized mechanism capable of handling parameterization of workflows.

Interactions

This service sends and receives events via a JMS message bus, and acts as a client to the Host Information Service in order to configure the payloads to run on specific resources. For its relative position in the general client-service layout, see the infrastructure diagram; a slightly more detailed look at the service topology is illustrated by an interaction diagram for a typical simple submission.

Mode of Usage

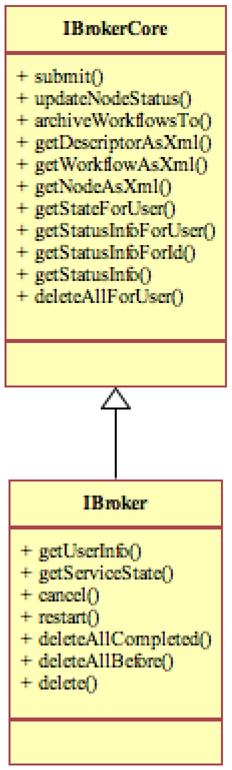
In order to run through the broker, a Workflow Description must be created and submitted (refer to the Siege Tutorial for a walk-through of this process). This description in turn will contain one or more scripted payloads, which can run from trivial to very complex, depending on the nature of the work to be accomplished through a single job launched onto the host; however, no special changes or recompilation of the underlying computational codes is necessary.

Siege also allows the user to inspect the state of workflows which have been or are being processed and to view all the events published in connection with them; this is particularly useful for debugging purposes. Once again this can be done entirely asynchronously (i.e., one can disconnect and then, upon reconnecting, retrieve a historical view of the state of the workflow). The level or granularity of information remotely propagated over the event bus can be controlled by the user on a workflow-by-workflow basis.

API

Service WSDL

The broker API, which includes both workflow-management as well as administrative methods, is as follows:

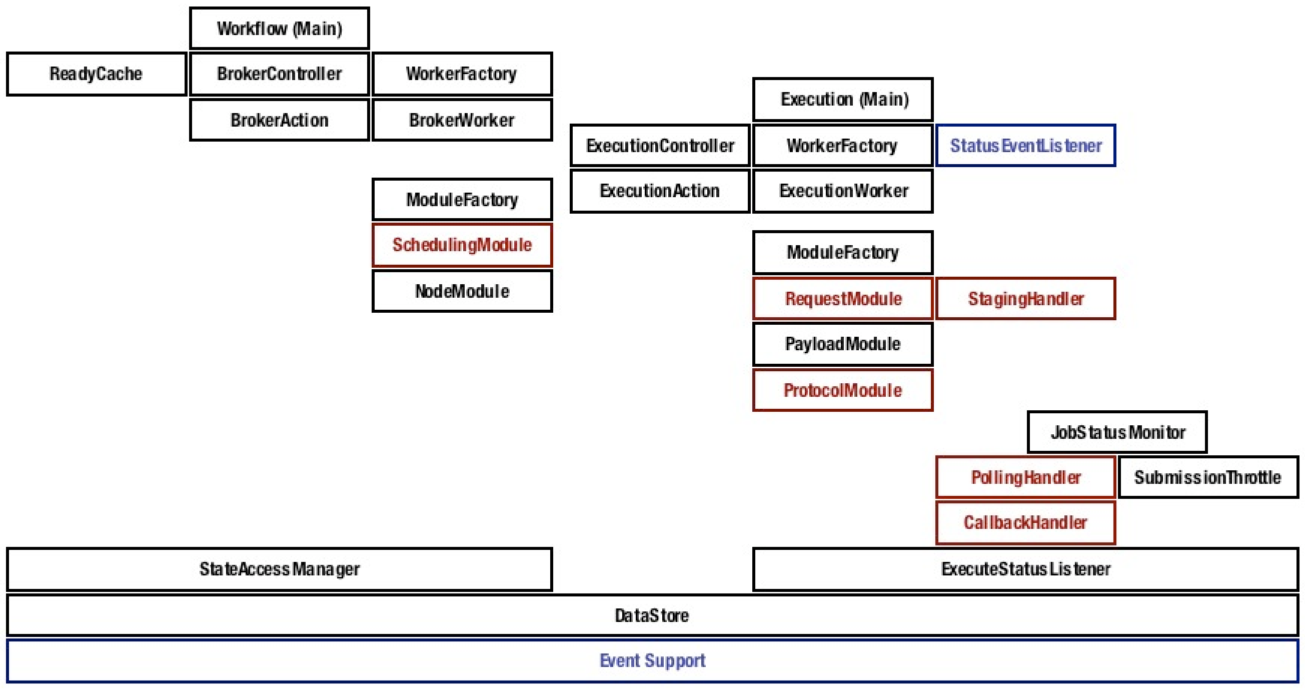


```
public interface IBrokerCore
{
   /**
    *  @param  submission description of workflow to submit.
    *  @return handle containing identifiers for workflow
    */
   public SubmissionHandle submit( BrokerSubmission submission );

   /**
    *  @param sessionId should uniquely identify the node to this service.
    *  @param status the state of the node.
    */
   public void updateNodeStatus( Integer sessionId, String status, Property[] output );

   public void archiveWorkflowsTo( String archiveURI, String configurationXml );

   /**
    * @param id the primary key of the workflow object, not the submission
    */
   public String getDescriptorAsXml( Integer id );
   public String getWorkflowAsXml( Integer id );
   public String getNodeAsXml( Integer id );
   public AggregateSummary getStateForUser( String user );
   public StatusInfo[] getStatusInfoForUser( String user, Integer lastId );
   public StatusInfo getStatusInfoForId( String globalId );
   public StatusInfo[] getStatusInfo( Integer lastId );
   public void deleteAllForUser( String user );
}

/*
 * Administrative methods.
 */
public interface IBroker extends IBrokerCore
{
   public UserSubmissions[] getUserInfo();
   public AggregateSummary getServiceState();
   public void cancel( String globalId );
   public void restart( String globalId );
   public void deleteAllCompleted();
   public void deleteAllBefore( Long time );
   public void delete( String globalId );
}
```

See further BrokerSubmission for a description of the submission objects. (We leave out discussion of the administrative query return types, since in most cases querying will take place through Siege, and those objects will remain hidden from the user; their schematic representation can nonetheless be obtained by inspecting the Broker WSDL.)

Implementation

Workflow Broker Architecture



Red signifies a component which makes external calls, either to web-service ports, file-transfer protocols or remote execution.

The workflow broker consists of three functional units:

- Graph logic management (**= Workflow**)
- Scheduling
- Submission and monitoring of jobs (**= Execution**)

The second unit interacts with external scheduling systems, such as MOAB, where the brunt of the scheduling work is done.

The other two units are both designed as follows:

1. Underlying each unit is a "controller", consisting of a queue and a subscriber. Actions are placed on the queue, and the subscriber delegates them to appropriate workers.
2. An action is typed, corresponding to some state associated with the workflow or the job (e.g., Workflow Submitted, Node Completed, etc.), and contains (minimally) an id pointing to the workflow or part of the workflow to be acted upon.
3. A worker is typed in accordance with the action it is capable of handling, and is threaded.
4. The subscriber uses a factory (Spring) to match the worker to the action, then makes a request to a threaded-worker-pool in order to run the worker.

A worker in turn can make use of modules encapsulating particular functional variants; for instance, in the graph-management sequence, there is a module for handling an executable node and a module for handling parameterized nodes; in the submission unit, there are modules for handling remote submission, local submission, for dealing with various payloads and protocols, and so forth.

Each unit also has a number of singleton pieces mediating between the persistent data and the current activity. In the graph-management unit, the **`ReadyCache`** is responsible for determining which nodes are available for running; in the submission unit, the **`JobStatusMonitor`**, with its various handlers, watches over external job state. In both units there is a manager or listener which controls access to the data store itself.

Finally, the event support component is responsible for both the sending and receiving of events; in the case of the submission unit, there is a **`StatusEventListner`** subscribed to the event support layer which listens for all of the status events produced by the remotely executed jobs which the broker knows about (for a schematic view, refer again to Events & Messaging).

For a more detailed look at how a workflow or node is processed inside the broker, refer to the Data Flow Diagrams.