**Early Design Plans for Generic Pilot Provisioning**

This document is a copy of the body of gittrack ticket #2377, as of 2011-08-12 13:00.

On July 29, Miron, Igor, Brian, Jaime, and Dan met to begin coming up with a plan for first-class support for "resource provisioning." Concretely, what this means is managing generic pilot jobs in a distributed computing environment such as OSG. This document attempts to present the results of that discussion.

**Overview of Desired Design**

A "pilot broker" service will make pilot allocation decisions based on demand + policy. The broker will communicate with pilot factories running at one or more grid sites. It is a design goal to have the broker make aggregate requests rather than single-pilot requests.

Although the broker should not make decisions about individual pilots, failure details of pilots need to be propagated back to the broker for debugging purposes.

The requests that may be made by the broker to the factories:

* I want N idle pilots of type X, but no more than M max total (i.e. periodically submit/retire pilots until N and/or M are reached). This request replaces previous requests for pilots of type X. A lease is associated with the request, so it must be periodically renewed or the pilots will be stopped/drained.
* define pilots of type X: what command to run, files to transfer, arguments to pass, environment to set (i.e. a job description)
* tell me the aggregate status of my requested pilots: running, idle, successes and failures in recent timeframes (or this info could be periodically pushed to the broker)
* give me error diagnostics (e.g. logs) for pilots that have failed (or this info could be pushed to the broker)

**Implementation A - glideinWMS**

The glideinWMS could be extended to meet the design goals. The framework already provides a protocol for making requests, but the requests are not for generic pilots of type X.

The factory component could be implemented using something very close to the existing glideinWMS factory. It would run at the target site and would submit pilots into the batch system, either directly or via a schedd that forwards jobs to the local batch system via the blahp.

The broker component would be another instance of the glideinWMS factory, but instead of submitting pilot jobs, it would send aggregate requests to the pilot factories at the sites. The existing protocol for sending requests from the frontend to the factory could be used, but it would need to be extended to support arbitrary types of pilots.

Retrieval of pilot failure diagnostics would need to be implemented. This is already on the wish-list for glideinWMS as it exists today.

**Implementation B - Condor**

This is an exercise to think about Condor components (existing or missing) that could be used to implement the design goals.

We considered implementing the factory as a schedd plus additional appendages that implement the factory logic. The broker could communicate with the schedd using the Condor-C protocol.

**B.1 scheduler universe factory**

Pilot jobs could exist in the schedd queue with the expression OnExitRemove=False, so they keep running until removed. These jobs are forwarded to the local site batch system via the blahp.

The broker could manipulate these pilot jobs directly, but that violates one of our design goals of having the broker only deal in aggregate requests. Alternatively, the broker could submit scheduler universe jobs that translate aggregate requests into operations on individual pilots. There are two models for how this could work:

1. the scheduler universe tasks submitted by the broker could be crondor tasks that periodically evaluate state and perform actions; the broker can modify the request by updating attributes of the crondor job's ad or by removing/replacing the job. This way of doing things sounds similar to Derek's Campus Grid Factory, as pointed out in Brian's comment below.
2. the scheduler universe tasks are long-running jobs that have their own internal event loop. For example, an early-stage prototype that modified dagman to keep N jobs running could be adapted to this role.

One problem with implementing pilots as jobs that automatically resubmit themselves with OnExitRemote=False is how to collect failure diagnostics for run attempts that fail. One solution would be to have failed glideins go on hold until their logs are harvested. Another solution would be to make Condor support logging to files with names that change with each run attempt (e.g. support $$([RunCount]) in the stderr/stdout names).

**B.2 JobRouter factory**

This idea is based on Brian's patch, which is attached to this ticket. The JobRouter could be modified to do 1-to-N transformations instead of 1-to-1 job transformations. This allows it to become a pilot factory. The source job would serve as a pilot request. The destination jobs would be the pilots. Attributes of the source job specify the desired number of pilots to instantiate. The routing table (i.e. the transformation description) would be designed to do whatever is necessary to make the jobs run on the local site (e.g. use the blahp or Condor-C or Globus).

Aggregate stats about the pilots could be fed back into attributes of the pilot request ad, which could be conveniently queried by the broker. We would need to work out how error diagnostics could be fed back to the broker.