The Managed Computation and its Application to EGEE and OSG Requirements

Ian Foster, Kate Keahey, Carl Kesselman, Stuart Martin,
Mats Rynge, Gurmeet Singh

DRAFT of June 19, 2005

DRAFT

Abstract

An important model of Grid operation is one in which a virtual organization (VO) negotiates an allocation from a resource provider and then disperses that allocation across its members according to VO policy. Implementing this model requires the ability for a VO to deploy and operate its one resource management strategy. In this white paper, we argue that an approach in which VO management functions are mapped into the creation and operation of *managed computations* provides a simple yet flexible solution to the VO resource management problem in general, and EGEE's VO computation management problems in particular. We present the basic architectural framework, describe an initial implementation based on existing components and present performance results.

Table of Contents

1 Introduction 2
2 Managed Computations 3
2.1 Managed Computation Factory Interface 3
2.2 Managed Computation Factory Architecture 4
3 GRAM: A Managed Computation Factory 4
4 EGEE Use Case 6
5 Example Implementation: GRAM+Condor 9
5.1 Configuring Condor as a Resource Manager 10
5.2 Implementing Infrastructure Services 12
5.3 Implementing a Condor-based VO Scheduler 12
GT4 WS GRAM Performance 13
6 13
7 Open Issues and Next Steps 14
References 14

1 Introduction

A common requirement in Grid deployments is that *resource providers* be able to allocate resources (e.g., nodes on a compute cluster, disk space on a storage system or CPU fraction on a single processor) to *virtual organizations* (VOs). Each VO may then in term wish to disperse its assigned resources among its members as it sees fit: for example, giving analysis activities higher priority than simulation, and group leaders higher priority than students. In effect, we want to allow a resource provider to delegate to a VO (or, more specifically, to *VO administrators*) the right to control the use of a resource allocation.

One approach to implementing this VO decision making function is to create a *VO scheduler*. VO users submit requests to the VO scheduler which in turn meets these requests by submitting them to the appropriate resource provider(s) to execute against the VO's resource allocation Implementing this model requires the following basic abilities:

- Resource owners need a mechanism for allocating resources to VO administrators, and for ensuring that the VO cannot consume more than its allocation.
- The VO needs mechanisms for allocating resources to VO users, and for ensuring that no user can consume more then they have been allocated.
- The VO needs a mechanism for allowing users to consume resources allocated to the VO by a resource provider.
- The VO needs a mechanism for creating capabilities that can be considered part of the VO infrastructure, with the reliability and persistence characteristics that one expects of an infrastructure service.

We propose here both an architectural framework and an implementation approach that together address these requirements. The framework is based on a construct that we call the *managed computation*, a computational activity that a client can create with specified persistence properties and resource constraints, and then monitor and manage, via operations defined within a managed computation factory interface (Section 2).

The implementation uses a combination of the GT4 WS GRAM service and various resource managers to implement this concept within a service-oriented architecture, with the resource manager providing local monitoring and management functions and GRAM providing network access, security, policy callouts, and other related functions (Section 3).

We show how the managed computation construct can be used to implement an EGEE use case in which a "VO scheduler" deployed as a managed computation on a "head node" submits jobs to a "backend cluster," with the resources consumed by the VO scheduler constrained (Section 4). We also report on initial experimental results intended to evaluate an GT4+Condor implementation of this use case, from the perspectives of both performance and management effectiveness (Section 5). We conclude in Section 6 with a discussion of open issues.

2 Managed Computations

A *managed computation* is one that we can start, stop, terminate, monitor, and/or control. As we discuss below, we define these management functions in terms of a *managed computation factory interface* that defines operations that a management client can use to request the creation, monitoring, and control of managed computation.

This managed computation construct can be applied pervasively to the design of the VO management environment discussed in the introduction:

- The VO scheduler runs in a shared environment provided by the resource owner. To ensure that the VO scheduler does not consume more resources then it is allowed to, it is structured as a *service managed by the resource provider*. As a managed computation, the VO scheduler may also request that the manager ensure that it is restarted in the case of failure.
- The VO scheduler services requests for the VO user community. Hence, it can be viewed as providing managed functions such as job staging and submission to the VO user community.
- As part of its operations, the VO scheduler may request that operations take place on other resources (e.g., running a compute job on a cluster). These requests can be structured as requests to other entities to create new managed computation, such as compute tasks.

The important point to take away from this discussion is that the entire VO scheduling architecture can be rendered in terms of the creation and operation of managed computations, without the need to introduce any other special architectural or implementation concepts.

2.1 Managed Computation Factory Interface

Both VOs and VO user tasks may be created dynamically. Thus, services that can create managed computations are a critical component of our architecture. We refer to such services as *managed computation factories.* A managed computation factory's interface defines operations that a client can use to request:

- *Deployment*: creation of a managed computation on a computational resource or "host";
- *Provisioning*: Specification of how many resources and in what way may be consumed by the computation, determined via client and/or host; and
- *Persistence*: Specification of what to do if the environment in which the computation is executing, or the computation itself, fails, determined via client and/or host; and
- *Monitoring*: Client monitoring of managed computations.

A computation created by a managed computation factory may itself define a network interface and thus operate as a service, in which case we may refer to it as a managed service. In addition, we note that a service created by a managed computation factory

may itself be a managed computation factory. Indeed, this is a good way to think about the function of a VO scheduler.

2.2 Managed Computation Factory Architecture

Figure 1 illustrates the major architectural components of a managed computation factory. It consists of:

- *The factory service proper*. This service provides the interface to the network, formatting and exchanging messages, and implements the overall control of managed computation factory function. The interface to the managed computation factory defines the functions available to a management client. These functions include at a minimum the ability to create, configure, and destroy a managed computation.
- *A policy module* that is used to determine authorization, resource constraints, and persistence policy. In particular, the policy module is used to determine if a managed computation request (e.g., to create a VO scheduler) is consistent with the operational policy of the resource provider that is to host the managed computation.
- *A resource manager*. This is a resource-specific management system that does the heavy lifting associated with creating the managed computation, enforcing persistence and policy, and implementing management operations, such as service termination. The managed computation factory interacts with the resource manager via manager-specific commands and protocols.

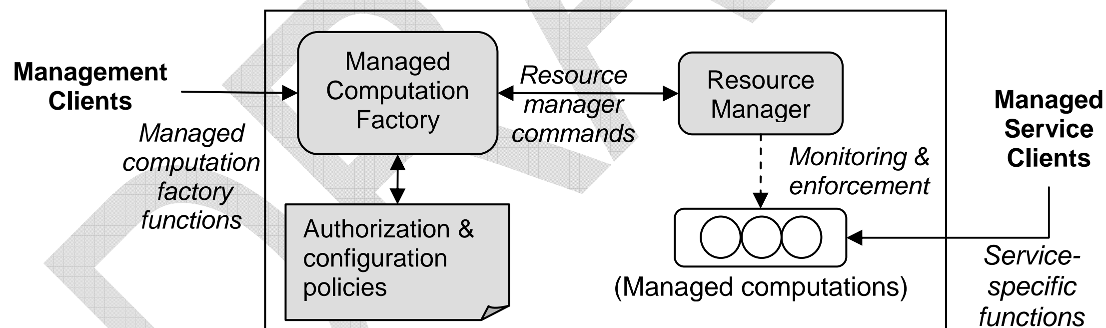

Figure 1: The managed computation factory

3 GRAM: A Managed Computation Factory

As we now describe, we can implement a managed computation factory via the *GT4 GRAM service*, working in conjunction with a *resource manager*.

The Globus Toolkit's Grid Resource Allocation and Management (GRAM) service is often viewed as a "job creation service." However, it is more properly viewed as a managed computation factory, in which the managed computation being created is an executing job.

In the GT4 WS GRAM, the factory service proper is implemented via a WSRF-compliant Web service. The managed computation factory functions are defined by the GRAM interface protocol combined with the Job Description Schema. Authorization and policy

are implemented via a GGF-standard authorization callout. The resource management function can be implemented via any of a variety of resource management systems. For example, a common configuration is for GRAM to use a system such as Condor, PBS, SGE, or LSF to implement a space sharing resource management approach. These systems also typically implement computation persistence via an automatic restart mechanism. Note that these systems can apply and enforce their own policy, which is usually rendered by defining properties associated with a queue such as memory consumption and maximum execution time.

In the case of space sharing, which is how compute clusters are typically operated, the enforcement mechanisms are straightforward. The managed computation (job) is allocated a fixed number of nodes that it is allowed to hold for a fixed period of time, at which point the managed computation may be terminated.

In non-space shared systems, GRAM has typically been configured to use a best effort resource management strategy (GRAM's "fork gatekeeper") by which jobs are simply forked and scheduled by the operating system. However, it is important to realize that these shared resources can also be managed by a resource manager (e.g., Condor, SGE, PBS, LSF)o provide more fine-grained policy and enforcement. These resource managers provide various functions that can be used to constrain the amount of CPU, memory, and/or network bandwidth consumed by a service. In other words, a *resource manager can provide precisely the management functions that we require to implement the managed computation abstraction*.

The previous observation is central to our approach. In brief:

- We view all computations as managed regardless of the resource on which they execute, be it a service node or a computational cluster.
- We use standard interfaces to managed computation factories (primarily GRAM, but possibly other specialized interfaces such as RFT or the workspace service) to request the creation of managed computations, regardless of whether those computations are services for schedule jobs for VO users (VO schedulers) or jobs to be run on computational clusters.
- Managed computation factories are not resource managers, but rather provide standard interfaces for interacting with resource managers. The interfaces remain consistent, while the resource managers may vary.
- There may be a number of different resources types (e.g., single nodes, multi-processor nodes, clusters). While each resource type may have a different resource manager, they are all accessed uniformly via managed computation factory interfaces.

We note that the term "head node" is sometimes used as an alternative to "service node," to denote a service node associated specifically with a cluster. However, this term tends to imply that there is only one such node and that its functionality is bound to a specific physical location. The term service node is more general. There may be multiple service nodes, and they need not be either collocated physically with the computational cluster nodes or distinct from the set of computational cluster nodes.

DRAFT

4 EGEE Use Case

To reiterate the requirements we seek to address:

- A VO should be able to cause the creation of a VO scheduler service. (The term compute element, or CE, is sometimes also used.) This is an infrastructure service and as such, this service should be able to continue to exist across a variety of failures, such as: failure of the environment hosting the service, failure of the network and even failure of the service itself. Initial behavior should be that in the case of any of these failures, execution of the service should be resumed assuming the execution of the underlying environment can be resumed. Further, the execution should be resumed in a way defined by policies.
- The VO scheduler may be run on a shared resource, hence the resource owner must be able to control the resources that it consumes. The purpose of this control may be to enforce fair-sharing, but also to simply ensure that the load on the system is within acceptable limits.
- As part of its operation, the VO scheduler may need to perform tasks on other resources or with other services. These may include transferring files, submitting jobs for execution, generating accounts and others. These tasks will be initiated using the credentials of the VO administrator. Note, that VO credentials may be mapped into local accounts on the resource via any of the standard mechanisms, including dynamic account creation.

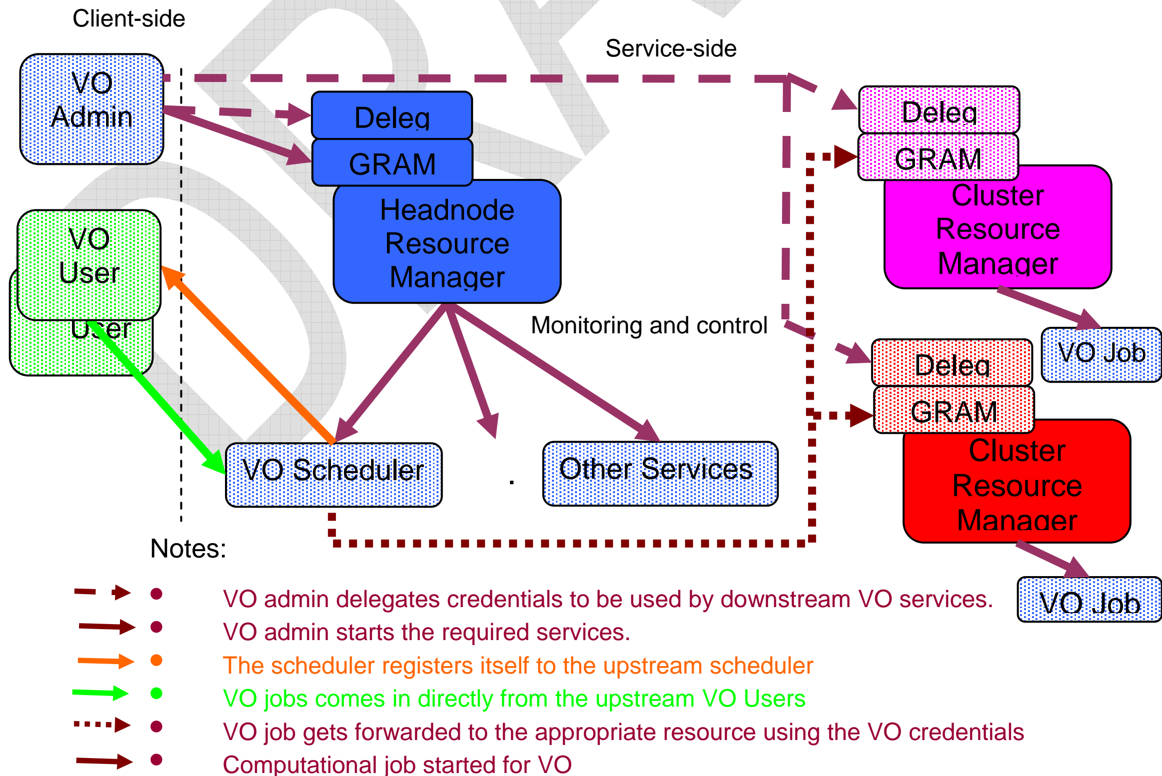

Figure 2: Managed service based architecture for VO resource management.

EGEE requirements can be addressed by the system architecture illustrated in Figure 2. This diagram shows two types of managed computation factories, one executing on the so-called service node and one on the computational nodes. The VO scheduler is created by a managed computation factory (GRAM) responsible for creating jobs on the service node. As described above, part of the job of the resource manager on the service node is to provide service persistence via restart after failure and to protect the service node by enforcing the policy of the service node owner, such as the number of VO schedulers that may execute at once, how much load each VO scheduler and processes created by the scheduler may place on the execution host, or how much memory they have consumed. A number of options are available for resource management of the service node, but below we will describe a method in which Condor can be used as a local, single-node resource manager.

As part of its operation, the VO scheduler may cause managed jobs to execute on a (potentially remote) computational cluster. Creation of this managed computation is the typical GRAM usage scenario in which the GRAM protocol is used to submit a job via a GRAM gatekeeper. In this case, the resource manager is most likely a typical cluster management system, such as PBS, which manages cluster nodes in a space sharing mode via a queue management mechanism.

In general, the resource manager configured on the service node may be different than the resource manager managing the cluster, even if they are both deployed on the same computer. If such is the case, two GRAM services will be deployed on the service node, one serving as the contact point for services executing in the service node environment (i.e., the VO scheduler) , the other for services running on the computational cluster (i.e., the VO jobs) However, it is important to note that these managed computation factories are creating computation on two different managed environments and that the collocation of the managed computation factory (GRAM) with the platform on which the managed computation is executing (i.e., the service node or computational cluster) is solely a matter of deployment and not architecture. This is an important feature of the service oriented approach.

In both cases, managed computations are created using credentials delegated by the VO administrators to the factory, as indicated by the blue shading. To facilitate this delegation, the VO administrator uses the GT4 delegation service to make delegated VO credentials available to the two GRAM services. Normally a delegated credential cannot be used to submit a job to GRAM so as to prevent an implicit trust relationship to be created. However, by providing a handle to the credentials in the computational GRAM delegation service (i.e. an end point reference), we can effectively federate the resources via the users proxy, and the VO scheduler is able to submit jobs from the service node to the computational node as the VO using the standard GT4 GRAM interfaces.

The user authenticates with the VO scheduler using their own credentials (indicated by the green shading) allowing the VO scheduler to assert VO policy to the users request. When accompanied by the appropriate delegation from the user, the VO scheduler may also be granted access to resources belonging to the user, for example staging files that are owned by the user.

Pulling this all together, we see the following sequence of events:

DRAFT

1. The VO administrator uses the delegation service interface to make delegated credentials for the VO available to the GRAMs managing the service node and the computational cluster.
2. The VO administrator uses the GRAM interface to submit a VO scheduler job to the service node. As part of its arguments, the VO scheduler may specify desired policy including if it should be restarted, and is given a reference to the delegation service on the compute nodes to which it will be submitting jobs. The VO administrator is given a handle with which it can manage the VO scheduler (observe its state, kill, etc.) The resource manager on the service node will enforce a resource consumption policy (i.e., how much CPU or memory the VO scheduler may consume) and restart the VO scheduler if the node fails.
3. The VO user authenticates to the VO scheduler using their own credentials and submits jobs using a VO scheduler specific submission protocol such as Condor (or even GRAM). The VO scheduler is responsible for queuing the job.
4. The VO scheduler selects which job it is ready to execute and performs any setup that may be required, such as staging files. Any activity that the VO scheduler process performs on the service node will be constrained by the policy enforced by the service node resource manager. Any operations that are performed by the VO scheduler on remote nodes will be constrained by the resource managers (if any) operating on those nodes.
5. The VO scheduler uses the GRAM interface to submit the VO job to the compute cluster. The job is submitted with VO credentials and is executed under the local account as determined by whatever mapping functions have been put in place by the resource owner (i.e. a mapfile, dynamic account creation, etc). The VO scheduler uses the delegation service for the compute cluster to delegate credentials to the job.

This approach has several distinct advantages:

- It appears to meet all the stated requirements for the VO scheduling architecture as defined by EGEE, subject to verification of the available resource managers' ability to monitor and control the resources consumed by VO schedulers and load generated by them in a satisfactory way.
- It leverages existing GT4 components virtually unmodified with the exception of a job description extension to specify persistence
- It can leverage the GT4 security implementation without modification, including GSI, delegation service, and interfaces to SAML-based policy processing engines.
- It is a pure service oriented architecture, enabling the components to be easily composed into higher level components and configured into a wide variety of physical deployment strategies such as supporting multiple resources for VO scheduler execution, execution of the VO scheduler on a cluster node, creation

DRAFT

of a remote VO scheduler, and extension of a VO scheduler to meta-schedule across multiple compute clusters.

In the next section, we will provide details of a specific implementation of this architecture based on GRAM and Condor.

5 Example Implementation: GRAM+Condor

To evaluate our premise that a pure managed computation approach can address EGEE requirements, we experimented with a configuration with the following properties:

- There is a single service node
- A single GT4 WS GRAM instance runs on the service node which provides two managed computation factories, one for computations running on the service node and one for computations running on an associated computational cluster
- Condor is used as the resource manager for computations running on the service node.
- LSF is used as the resource manager for computations running on the cluster.
- Condor-C is used as the VO scheduler.

It is important to distinguish the two distinct uses of Condor in this configuration, as otherwise confusion can result:

1. *Condor as a resource manager for jobs running on the service node*. To use Condor in this mode, we create a standalone Condor pool whose role is only to accept jobs from the GRAM running on the service node. Condor then is responsible for figuring out which jobs to run, when to run them, enforcing service node policy and restarting jobs if they fail. Note that the interface between GRAM and the Condor resource manager on the service node is the existing GRAM/Condor adapter that is a standard part of GT4.
2. *Condor as a VO scheduler implementation*. This is a completely separate Condor deployment and has no connection what so ever with the Condor used as the resource manager. Indeed, we could use, say, Sun Grid Engine as the resource manager for service node service and still use Condor as the VO scheduler, or we could use some other VO scheduler implementation, such as a dynamically created GRAM service. To provide the VO scheduler functionality we use Condor-C, which essentially sets up a submission point for VO user jobs. Thus a VO user can submits a Condor job as normal using existing Schedd protocols. The VO scheduler will then submit VO jobs to the compute cluster by using the Globus universe: that is, the existing Condor-G mechanism.

Note that other resource managers could be used instead of Condor. We have briefly looked at Sun Grid Engine, which also has a very extensive policy engine, but presented in a different way to the resource manager administrator.

5.1 Configuring Condor as a Resource Manager

Our use of Condor as a resource manager for the service node is solely an issue of configuration and does not require any modification to Condor. However, this use of Condor is somewhat unconventional, and thus we provide some details so as to provide better understanding of our proposed implementation.

Condor was deployed so that all Condor services (e.g., schedd, negotiator, collector, startd, etc.) ran on the service node. In other words we created a stand alone Condor pool, including submit host and compute resource, on a single compute resource. Note that if more than one service node is desired, the configuration could be trivially modified to support additional compute resources. We used the following Condor configuration parameters.

```
NUM_CPUS = 10
VIRTUAL_MACHINE_TYPE_1 = cpus=10%, ram=10%, swap=10%, disk=10%
NUM_VIRTUAL_MACHINES_TYPE_1 = 10

# Allow a maximum load of 40, so each VM gets a load share of 4
# considering 10 VMs, the maximum threshold at which to evict a
# job is twice of the allocated load amount 4 x 2 = 8
THRESHOLD  = 8

# Temporary requirement for distributed job submissions
WANT_SUSPEND = True
WANT_VACATE  = False

# Do not start a job if the current total load on the host
# is 40 (maximum allowed load)
START        = TotalLoadAvg < (( $(THRESHOLD) * $(NUM_CPUS))/2)

# If the total load is above 40 then suspend jobs that are generating
# more than the threshold for suspension (i.e., 8)
SUSPEND      = (((CondorLoadAvg/TotalCondorLoadAvg) * TotalLoadAvg) \
                      > $(THRESHOLD) ) && \
               ( TotalLoadAvg > (( $(THRESHOLD) * $(NUM_CPUS))/2))

# continue suspended jobs when the total load comes down below 20
CONTINUE     = TotalLoadAvg < (( $(THRESHOLD) * $(NUM_CPUS))/4)
PREEMPT      = False
KILL         = True
PERIODIC_CHECKPOINT     = False
PREEMPTION_REQUIREMENTS = False
PREEMPTION_RANK = 0
```

Our first policy decision is to limit the total number of managed computations that can execute at one time. (Note that in practice, the number of processes is always limited, although that limit may be a large number). Condor only allows one job to execute per compute node, but does allow the creation of multiple “Condor virtual machines” (CVMs), each of which can run a job. (We use the term “job” to refer to a task submitted to GRAM or passed on by GRAM to the underlying resource manager. A job may consist of many dynamically created processes, which may run sequentially, or concurrently.)

In our experiment, we configured Condor to create 10 virtual machines, hence allowing up to 10 VO schedulers to be executing simultaneously. Note that while an arbitrary number of VO scheduler creation requests may be submitted, only 10 may be executing at any point in time. The mapping of a VO scheduler to a CVM is achieved by the standard Condor matchmaking mechanisms.

The bulk of the policy addresses specifying how much load any individual managed computation can place on the service node:

1. No new VO services are started if the system load is more than 40. This is implemented by defining a START policy for each VM. System load is determined by standard operating system mechanisms for reporting load average and checking the load prior to starting a job is directly expressed in the standard Condor policy language.
2. If the system load is below 40, jobs are not restricted as to how many system resource they consume.
3. If the system load goes above 40, then jobs that generate a load of more than 8 are suspended. The load of a job is determined internally by the Condor implementation by measuring overall system load, and then apportioning that load to the jobs (see www.cs.wisc.edu/condor/manual/v6.7/3_11Setting_Up.html). All processes associated with a job are suspended.
4. If the system load comes down below 20, resume the suspended VO services are resumed.

Though relatively simple, this policy has been shown to be effective at both stabilizing total system load and allowing jobs that are compliant those that are not. Note that for this strategy to be effective, it will be important for managed computations to both (a) know what policy they are operating under, so they can moderate their resource consumption, and (b) have a way of being notified when they are in violation so they may adapt their behavior. (Memory allocation is a good analog. A process can call getrlimits() to determine how much memory it has available, and if it asks for too much, a malloc call returns NULL. At that point, it is up to the application to do something about the limited resource, or fail.)

More advanced policies can easily be supported by extending the rules for SUSPEND and CONTINUE in the above configuration file.

The policy file presented here is expressed only in terms of system load average and the contribution of jobs to that load. However, the policy can be extended to include other factors, such as memory consumed. In the default configuration, Condor only monitors system load average and the contribution to that load average for jobs running under Condor. However, Condor provides a general mechanism by which the Hawkeye monitoring component can be used to incorporate additional machine parameters into policy statements. For example, the amount of memory on a per process basis can be determined via the linux **ps** command, or by looking at **/etc/proc**. By interfacing these commands to the Hawkeye monitor, these values can be made available to the Condor system as ClassAd values, which in turn can be included in policy statement, such as the SUSPEND statement in the above example. Condor currently includes Hawkeye monitors for total virtual memory and this will have to be generalized if memory utilization on a per VO basis is required. However, it remains to be determined what types of policy are desired by EGEE, and how the current load based mechanisms should be paramterized and extended to include additional factors

DRAFT

5.2 Implementing Infrastructure Services

Infrastructure services are those services that are critical to the operation of an organization. Thus, GRAM may be a critical service to a resource provider, while a VO scheduler may be considered critical for a VO. Thus we need to be able to ensure that the managed computation factory (GRAM), underlying resource manager (e.g., Condor), and the managed computation (VO scheduler) persist beyond a failure.

To address the persistence of the factory, GT4 was designed such that GRAM is able to be restarted and reinitialize its state. Hence if the platform on which GRAM executes should fail, or the GT4 execution environment should fail, GRAM can reconnect itself to all its active and submitted tasks.

For GRAM restart to be effective, it must be the case that the resource manager being used by GRAM also is persistent. Fortunately, Condor also has this property. There is however, one final piece to this puzzle, which is that ultimately infrastructure services being managed by the resource manager should also be restarted. This however places responsibility on the managed computation, requiring it to checkpoint its state in such a way as to able to restart. For this reason, not all managed computations are automatically restarted, but rather, the need for restart is indicated as part of the jobs description provided to GRAM. This is achieved by extending the GRAM job description to include a PERSISTANCE option.

This entire process can be bootstrapped by placing appropriate entries into the systems initialization table (i.e., initab) to ensure that the GT and RM services are always running and in turn the RM will restart any managed process as determined by the PERSISTANCE option in the job description.

Persistence properties may be driven by policy statements that specify, for example, how often a service should be restarted. Such policies can be partially accommodated within the resource manager by providing Condor with additional options that specify how many times to retry if the VO service enters the Held state. The Held state can indicate a temporary failure, like timing out while connecting to a GRAM service. During testing, we added the following to the job description created by Globus (lib/perl/Globus/GRAM/JobManager/condor.pm):

```
periodic_release = (NumSystemHolds <= 3)
periodic_remove  = (NumSystemHolds > 3)
```

This text specifies that the job should be restarted three times in case of a temporary failure, but on the fourth time the job should be removed. A reasonable default for the number of job restarts can be set in the condor.pm file. This default can be easily modified by a sys-admin if desired. In addition, GRAM could be enhanced to allow the specification of restart behavior in the job description document. As always, the RM Owner would retain control of accepting or rejecting requests via the condor.pm file.

5.3 Implementing a Condor-based VO Scheduler

We are in the process of evaluating a VO scheduler based on the Condor schedd (i.e., Condor-C). As outlined above, in this scenario, the VO admin uses GRAM to submit a managed computation job (via GRAM+Condor) to start the schedd. The schedd is provided a Condor configuration file that configured the dynamically created schedd into

DRAFT

a VO wide Condor pool. A VO user can then submit a job to the local Condor pool, and the job would flow from the VO Condor pool to the schedd using Condor-C. Given our service oriented approach, the schedd will use Condor-G to submit the job to the computational resource for eventual execution. The following shows the submit file used by the VO scheduler to submit the job to the computational nodes.

```
universe = grid
grid_type = condor
executable = /bin/hostname
output = out/out.cc.$(Cluster).$(Process)
error = out/err.cc.$(Cluster).$(Process)
log = out/log.cc.$(Cluster).$(Process)
notification=NEVER

remote_schedd = TEST_VO@nimrod.isi.edu
+remote_jobuniverse = 9
+remote_jobgridtype="gt4"
+remote_globusresource="https://somehost/wsrf/services/ManagedJobFactoryService"
+remote_jobmanager_type="Fork"
+remote_x509userproxy="/tmp/x509up_u4017"
Queue
```

The VO service and the computational jobs both run under the VO administrators credentials. To enable this, we can leverage the delegation service, which is a new feature of GT4. One of the basic assumptions of GSI is to prevent implicit transitive trust relationships, hence it is not possible to use a delegated credential to create a job. However, the delegation service allows the VO administrator to place a delegated GSI proxy into the managed service environment for both the service node and the computational nodes. By having the VO administrator provide reverences to the prestaged credentials, it is therefore possible for the VO scheduler to submit a job on behalf of the VO.

The VO user authenticates to the CE using their own credentials. Once authenticated the VO scheduler can enforce any applicable VO policy. It then takes the submitted task and using its reference to the VO credentials in the computational server submits the job for execution. (There is currently a limitation in the Condor-C implementation that does not make it possible to alter the credentials used to submit the jobs. Thus, the job is submitted to the compute resource using the user’s credentials. Condor-C is being augmented to remove this limitation.)

6 GT4 WS GRAM Performance

We present the results of experiments designed to measure GT4 GRAM performance.

All tests were done on ruly.mcs.anl.gov, running Red Hat Linux release 7.3 with two Intel(R) Xeon(TM) CPUs at 2.20 GHz, and sporting 2 GB of RAM.

Results were obtained by sustaining a job load N from 1 to 128 by 2n (1,2,4,...,128) for 15 minutes. The number of client threads M were also varied from 1 to 128 by 2n giving actual sustained loads of N * M. For example, if 4 client threads (M) are started with a job load value of 64 (N), then in total there would be in general 4 * 64 = 256 jobs being operated on by the service through the duration of the test. The results below show *submission rates in jobs per minute*.

The first set of results are for simple sleep-for-five-seconds jobs with no delegation done. The second set are for simple jobs that share a delegated credential. The third set are for jobs which stage in the echo tool to a new job directory, run the echo command with boring output, and then stage out their stdout and stderr files (all using a shared credential to contact RFT). The cells labeled "**Serv. OOM**" are tests which failed to complete because of a java.lang.OutOfMemoryError on the service with the default heap size. The cells labled "**TMOF**" are tests which failed to complete because of a "Too many open files" error on the service when it attempted to fork too many jobs. The cells labled "**N/A**" are tests which were not attempted because of assumed failure.

Throughput Without Delegation for Simple Job

		Number of Client Threads (M)							
		1	2	4	8	16	32	64	128
Sustained Job Load Per Client Thread (N)	1	7	15	29	57	80	69	69	70
	2	15	29	58	79	74	70	70	64
	4	29	58	78	77	68	69	52	69
	8	59	77	77	72	65	27		69
	16	77	77	75	64	27			50
	32	76	75	68	64	67			N/A
	64	75	73	70	66	65		N/A	N/A
	128	80	72	64	63	71	TMOF	N/A	N/A

7 Open Issues and Next Steps

TBD

References

TBD

DRAFT