GLARE: A Grid Activity Registration, Deployment and Provisioning Framework *

Mumtaz Siddiqui, Alex Villazón, Jürgen Hofer and Thomas Fahringer
Institute of Computer Science, University of Innsbruck
Technikerstraße 21a, A-6020 Innsbruck, Austria
{mumtaz|avt|juergen|tf}@dps.uibk.ac.at

ABSTRACT

Resource management is a key concern for implementing effective Grid middleware and shielding application developers from low level details. Existing resource managers concentrate mostly on physical resources. However, some advanced Grid programming environments allow application developers to specify Grid application components at high level of abstraction which then requires an effective mapping between high level application description (activity types) and actual deployed software components (activity deployments). This paper describes *GLARE* framework that provides dynamic registration, automatic deployment and on-demand provision of application components (activities) that can be used to build Grid applications. *GLARE* simplifies description and presentation of both activity types and deployments so that they can easily be located in the Grid and thus become available on-demand. *GLARE* has been implemented based on a super-peer model with support for activity leasing, self management, and fault tolerance. Experiments are shown to reflect the effectiveness of the *GLARE*.

1. INTRODUCTION

Advances in network technologies and the emergence of Grid computing have provided the infrastructure for computation and data intensive applications to run over collections of heterogeneous computing nodes. A main goal of a Grid is to provide uniform access to wide-area distributed resources. It is widely accepted that resource management which is responsible for the management of physical resources like networks, storage and computers and logical resources like replicated data files and software components, is of paramount importance for Grid infrastructures. Most existing resource management systems focus mainly on physical resources typically dealing with Grid computers and job submission systems. Some efforts like GrADS [16], AppLeS [6], GridARM [36] and GridLab [35] have been made to provide automatic management of physical resources. There is still much work to be done to effectively support deployment and management of software components that essentially may become part of Grid applications. Grid workflow applications [41] emerge as some of the most challenging and important classes of truly distributed Grid applications. Grid workflow applications require the composition of a set of application (software) components (e.g. executables or Grid/web services) which execute on the Grid in a well-defined order to accomplish a specific goal. Most existing systems require manual or semi-manual deployment of these software components, as well as force application builders to hardcode specific software components deployed on specific Grid sites into their Grid applications. In addition, currently available Grid information services are not well adapted to store complete description of software components, forcing the application builder to use only *(name,location)*-like information about available applications. As a consequence these applications are difficult to port to different Grid architectures, are sensitive towards dynamic changes of a Grid infrastructure, and often imply an avoidable failure rate during execution. Such a manual and hardcoded approach forces an application developer to deal with low level details of the Grid. Services and executables must be described along with their locations and access paths or URIs. All of that makes application development a time consuming, tedious and error prone task.

There exists several sophisticated Grid workflow programming environments and paradigms [15, 19] that allow a programmer to specify the semantics of software components as part of a workflow. But there is a gap between the description of the functionality of a component and the actual deployed services and executables that can provide such functionality. We believe that this gap can be closed or at least narrowed down by separating the description of the functionality of a component from it's deployments, and through a sophisticated mapping mechanisms that goes beyond management of physical resources. Such an advanced management system should support dynamic registration, automatic deployment, on-demand provision, and leasing of software components.

In this paper we describe *GLARE*, a Grid-level activity registration, deployment and provisioning framework that provides dynamic registration, automatic deployment and on-demand provision of software components. *GLARE* is

*The work described in this paper is partially supported by the Higher Education Commission (HEC) of Pakistan and partially supported by European Union through the FP6-IST-004617 project ASG.

SC | 05 November 12-18, 2005, Seattle, Washington, USA

designed and implemented as a distributed framework that stores information about application (software) components, called *activities*. Activities are the essential components of a Grid workflow application that may reside on different computers and execute in a well defined order to accomplish a specific goal of the application. *GLARE* provides distributed registries for *activity types*, *activity deployments*, and services which perform registration, provisioning, monitoring and automatic deployment of new *activities* on different Grid computers in a Virtual Organization (VO). Note that *activity types* refer to a functional description of activities whereas *activity deployments* relate to executables or Grid/web services that can actually be executed. Application developers can focus on *activity types* and thus must not be aware of specific *activity deployments*. *GLARE* simplifies the description and presentation of both activity *types* and *deployments* in such a way that they can easily be located in a distributed Grid environment and thus become available on-demand.

Moreover, *GLARE* provides a leasing mechanism which enables a client (such as a scheduler or enactment engine) to *lease* an activity *deployment* for a certain timeframe. *GLARE* has been designed and implemented based on the super-peer model [38] with support for self management and fault tolerance. It remains available and functional even if some of the Grid computers or services stop working. *GLARE*'s dynamic registration, automatic deployment and on-demand provision of the Grid *activities*, in combination with GridARM's resource brokerage and advanced reservation [36], provide a powerful base for the Grid workflow management system and substantially improve the usability of the Grid towards an invisible Grid. We developed *GLARE* and *GridARM* as integrated services of the ASKALON Grid application development and computing environment [18]. A *GLARE* prototype has been implemented based on the Globus Toolkit 4 [24], which is a reference implementation of the new Web-Services Resource Framework (WSRF) [3].

The rest of this paper is organized as follows: In Section 2 we discuss Grid workflow activities and motivation behind our work. Thereafter, we describe the architecture of the *GLARE* in Section 3. In Section 4, we describe our experiments about the *GLARE* and discuss the results. We present related work in Section 5 followed by conclusion and future work in Section 6.

2. MOTIVATION

A Grid workflow consists of Grid *activities* [19, 41]. A Grid *activity* is a high level abstraction that refers to a single self contained *computational task* that corresponds to an execution unit, initiated for instance by an executable or a service deployed on a certain Grid site. In this section we present motivation behind the *GLARE* system and describe its simplicity. Also, we demonstrate *activities* as generalised abstractions of the Grid tasks/jobs.

2.1 An Example Using Basic Grid Services

In order to illustrate the advantages of the *GLARE* framework, we consider a simple example of a workflow consisting of two activities: *ImageConversion* and *visualization* shown in Fig. 1. The input of *conversion* activity is a *POVray*[1] [33]

[1]POVray is a high-quality tool for creating stunning three-dimensional graphics.

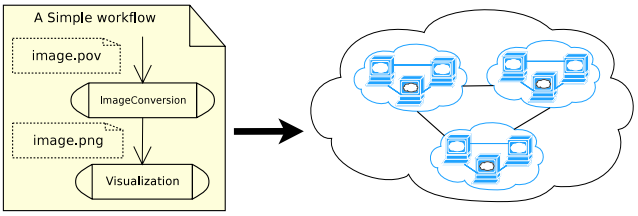

Figure 1: A simple workflow execution on the Grid.

source file containing description of a scene, which is used to generate a 3-D image file. A client who wants to initiate *conversion* activity on a Grid site (e.g. on a powerful computer), needs to deploy *POVray* on the target Grid site, then sends a request to perform the image conversion, and finally transfer the resulting image to run a *visualization* activity on his computer to analyze the resulting image. To simplify the understanding, we assume that a Java version of *POVray* (*JPOVray*) is available in the form of an executable and also a Grid/web service (*WS-POVray*) which wraps the execution of *POVray* in a web service.

The required components to deploy the *JPOVray* application are: (a) the `javac` compiler (b) some (possible required) libraries in the form of `.jar` files, (c) the `ant` build tool and (d) the source code of the *JPOVray* application itself. Once the application is (remotely) built and deployed, we need to store the information about deployed application in some information service. The *Endpoint Reference (EPR)* or URI in case the deployed application is a Grid/web service, and the *application name, path* and *home* in case the application is an executable.

The remote compilation[2] and deployment procedure requires information about the location of the compiler and built tool on the remote Grid site, URI of required libraries and *JPOVray* source code. Example 1 shows a step-by-step procedure that is needed to perform the compilation, deployment and execution of the workflow, using the basic Globus services, that is GRAM [11], MDS[3] [10] and GridFTP [2] on a target Grid site:

Example 1 (Step-by-step execution of the workflow)

```
# Preparing environment
JAVA_HOME = Query MDS for location of java on target Grid site
if java not found then
   - Query MDS for the location of JDK installation file
   - Transfer installation file to target Grid site
   - Create user-defined JDK deployment script
   - Submit installation script using GRAM
   JAVA_HOME = user-defined location used to deploy JDK
   - Update MDS with the information about the deployed JDK
endif
ANT_HOME = Query MDS for location of ant on target Grid site
if ant not found then
   - Do same steps to install ant as done for java and update
MDS
endif
povray_libs = Query MDS for libraries
# Transfer needed application data for deployment
```

[2]Notice that the compilation of Java code is for exemplar purpose, otherwise *Write once run everywhere* is ideal for Java applications.

[3]By default only physical resources are registered in MDS, but it can be used for logical resources like application components as well.

```
- Transfer the required libraries
- Transfer java application (JPOVray) source code
# Prepare build scripts
- Create script to remotely build and deploy JPOVray
using the information from MDS (JAVA_HOME, ANT_HOME
and set CLASSPATH)
- Submit deployment script through GRAM
povray_location  = user-defined location on remote Grid site
- Update MDS with information about newly deployed JPOVray
application (i.e. jpovray_location, libs_location etc.)
# Using the deployed application
- Query MDS to find JPOVRay service location
if deployed application is Grid/web service then
   - Contact the WS-POVRay service directly
elseif deployed application is an executable
   - Create script to run jpovray using
   java and libs_location
   - Submit execution script to run jpovray through GRAM
endif
# Visualization
- Retrieve result using GridFTP
- Visualize image on local station
```

In Example 1, we need to put application-specific information of the JDK and Ant in some information or registry service for (a) the deployment of the *JPOVray* and (b) the execution of JPOVray itself, i.e. there is a special need to store *activity*-specific description, so that the procedure can be automatized as much as possible. This becomes very complex for several activities, which must be orchestrated and executed as a Grid workflow application[19].

The main problem is that the information stored in the information service (like MDS) maps the name of the activity directly to its location. Therefore, the description of the workflow cannot be done independently of a given application deployment, which represents a major disadvantage of current systems. We need a service which allows the registration, deployment and provisioning of *activities*, in order to simplify the automation of service execution and composition. The information stored in such a registry service should allow to map (a) the description of the deployed application activity, and (b) the access point (EPR or host:/path/to/application). We believe that such an activity registry should work in coordination with MDS, which is well adapted to store static information about available Grid resources (e.g. available Grid sites, operating system, etc.), but not well adapted to store application-related information.

Creating an *automatic deployment* procedure for an application, as described in Example 1, using basic Grid services is non-trivial and hard to achieve in practice. We present a more practical solution for this problem based on the *GLARE* system.

2.2 A GLARE-based solution

During workflow composition we would like to specify a functional (or semantic) description (*types*) of *activities*, so that based on their descriptions, the associated *activity deployments* can be located and used. For this purpose, a fine-grained activity-based Grid resource management is required.

In order to provide this management, *GLARE* allows a user to describe *activities* in the form of *activity types* and *activity deployments*.

- An *activity type (AT)* is a functional or behavioural description, which can be used to lookup or deploy an *activity* (application component).
- An *activity deployment (AD)* refers to an executable or Grid/web service and describes how they can be accessed and executed.

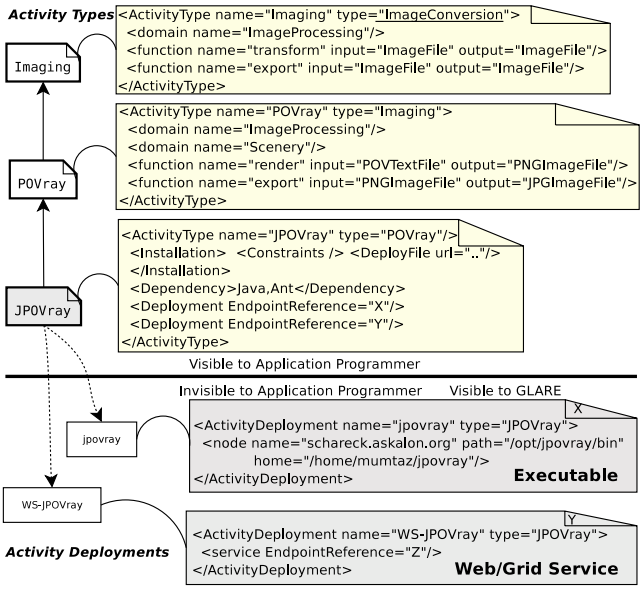

Figure 2: Activity Type Hierarchy and *type* to *deployment* mapping.

Activity Types are organized in a hierarchy of *abstract* and *concrete types*. An abstract type is one which has no directly associated deployment. A *concrete type* may have multiple *deployments* and a *deployment* may have multiple *instances*. A running process of an activity is called *activity instance*.

As shown in Fig. 2, *Imaging* and *POVray* are abstract types which perform some kind of image processing and define functionality (*render and export*) with possible inputs/outputs. *JPOVray* is a concrete type that extends *POVray* and *Imaging* and thus inherits functional description of the base types. Installed occurrences of a *concrete type* on different Grid sites, or on the same site with different options are referred to as their *deployments*. *WS-JPOVray* and *jpovray* shown in Fig. 2 are *deployments* of *JPOVray*. A developer only uses *activity types* while composing a Grid workflow application. The *GLARE* system hides *deployments* from the developer, and transparently maps *activity types* composed in the application, to the *deployments*. This is a major advantage, since the Grid workflow composer, does not want to know how and where the *POVRay* application is actually implemented (as an executable, Grid/web service, etc.) on the Grid. Activity *instances* are not shown in Fig. 2. They are specific to a given execution of the Grid application and typically handled by the execution engine [13].

Fig. 3 shows a more complete overview of the different components that are needed to deploy and execute our example workflow described in Example 1 and shown in Fig. 1. In addition to the different abstract and concrete types, the dependencies between the components are shown. *GLARE* manages these dependencies over the Grid sites.

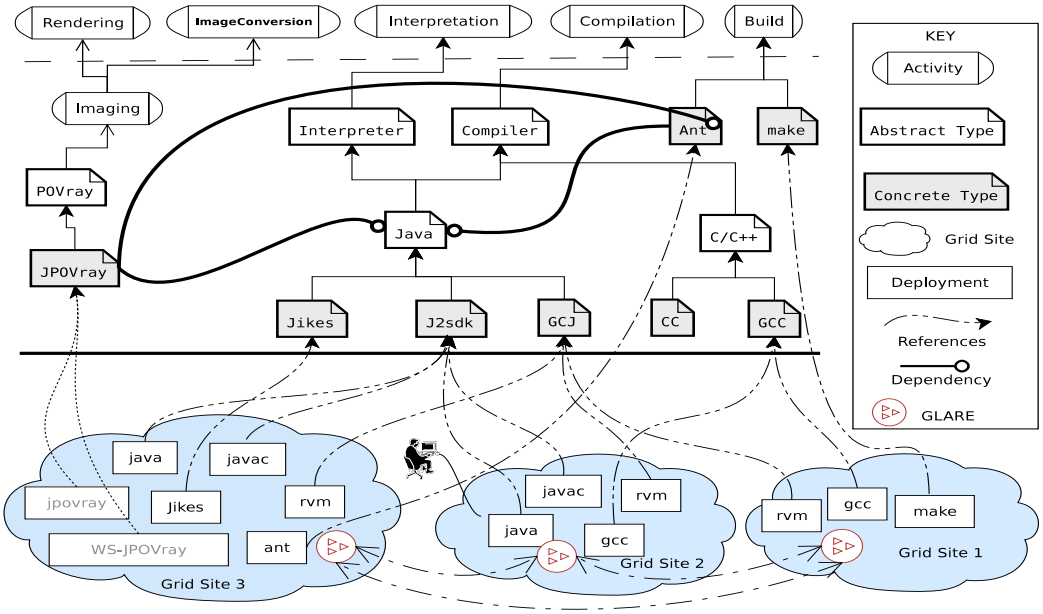

Figure 3: Example activities, type hierarchy and Deployments on different Grid sites.

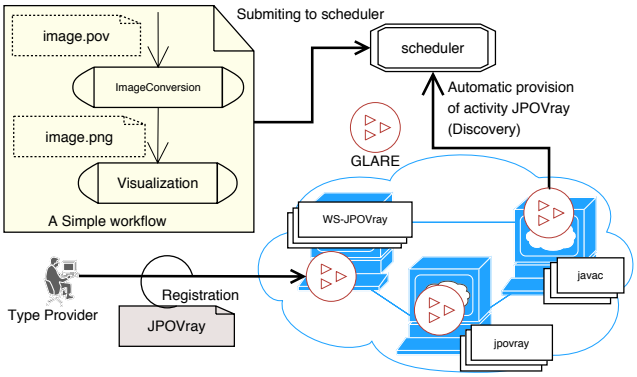

Figure 4: A simple workflow execution by a scheduler with help of *GLARE*.

GLARE framework consists of distributed *GLARE services*, which can perform automatic deployment of new *activities*. Each Grid site has a local *GLARE* service. The service provider describes the *activity types* to be registered with *GLARE*. The detailed information description that has to be provided is described in Section 3. Example 2 shows registration of *JPOVray activity type* in the *GLARE*. Notice that the registration of an *activity type* is done only on a single Grid site, and *GLARE* takes care of distributing and deploying it on other sites on-demand.

Example 2 (Registration of JPOVray type)
`JPOVray.xml` = *Define* `JPOVray` *activity type in a xml file*
`if provider` *does not know xml format* `then`
  - *Transfer* `template` *xml from local GLARE service*
  - *Modify* `template` *xml*
`endif`
- *Register* `JPOVray` *in the* `local` *GLARE service*

The workflow shown in Fig. 1 can be composed using the *activity types* stored in the *GLARE* registry. The description only specifies that a user needs an *activity* that can produce an image using as input a *POVray* scene description source file. The workflow description can then be submitted to the *scheduler*. The scheduler interacts with a *local GLARE service* and requests for an *activity deployment* capable to provide the requested service (*ImageConversion*). Example 3 shows steps involved in executing the workflow with the *GLARE* system. A client (*scheduler*) specifies *activity type* (any one in the type hierarchy) and *GLARE* returns a list of *deployment* references. Fig. 4 demonstrates registration of *JPOVray activity type* on one Grid site by activity provider and discovery of *JPOVray deployments* by the scheduler from an other Grid site. Both activity provider and scheduler interact with their local sites.

Example 3 (Execution of workflow using GLARE)
`Result` = *Get* `ImageConversion` *deployments using local GLARE*
`if Result` *is an error* `then`
  - *Retry later*
`else`
  - *Select a* `Deployment` *from the* `Result`
`endif`
`if Deployment` *is a Grid/web service* `then`
  - *Contact the service(WS-JPOVRay) directly*
`elseif Deployment` *is an executable*
  - *Instantiate JPOVray using* `jpovray executable` *as* `GRAM` *job*
`endif`
`# Visualization`
- *Retrieve result using* `GridFTP`
- *Visualize image on local station*

On a discovery-request by the scheduler, the local *GLARE* service (e.g. at Grid site 2 shown in Fig. 3) internally advances as follows:

- It looks up *ImageConversion* in the distributed *GLARE* registry service and finds (after an iterative lookup) *JPOVray*, a concrete *activity* of the required type, but without any *deployment* anywhere in the Grid or VO.

- It analyzes the *JPOVray type* and found that (a) *JPOVray* can be installed on Grid site 3 (b) *JPOVray* depends

on activities *Java* and *Ant* and (c) both *Java* and *Ant* are not available on site 3.

- It discovers *Java* and *Ant activity types* which are (a) suitable for target Grid site and (b) provide installation or *deploy-file* that describes the steps for automatic deployment.
- If *deploy-file* exists, it invokes *deployment handler* on the target site and hands over the *deploy-file* to it. Deployment handler performs all steps given in the deploy-file automatically. Otherwise, it transfers installation files and required libraries on the target site using GridFTP.
- Then it automatically connects to the target Grid site (as described in Section 3.4) to *build* and *install* both *Java* and *Ant activities* by automatizing the interactive installation process (procedure).
- It identifies *deployments* (*e.g. java, javac and ant*) associated with newly deployed *activities* and registers them in the *deployment registry* of the target Grid site along with information including executable path, home and type etc. *Deployments* are identified in an *activity type* description by the activity provider, or automatically by the *GLARE* service (e.g. by exploring *bin* sub directory of the deployed *activity* home for executables).
- Finally, it transfers *JPOVray* installation file on to the target Grid site and deploys it automatically. Furthermore, it identifies JPOVray *deployments* (i.e. *jpovray and WS-JPOVray*), registers them in the *deployment registry* of the target Grid site and returns their references to the client i.e. *scheduler*.

In this way, the *GLARE* system performs dynamic registration of new *types* and *deployments*, automatic installation and on-demand provision. The *deployments jpovray* and *WS-JPOVray* both provide the same functionality but belong to different categories, one is an *executable* whereas the other is a web service. It is also possible that both deployments of the same type belong to different Grid sites. Clients can select one of them suitable to their needs.

GLARE system hides *deployments* and the installation process of all *activities* thus shielding the client from the Grid.

3. GLARE ARCHITECTURE

The architecture of the *GLARE* system is depicted in Fig. 5. In brief, it comprises three principal components: *Activity Type Registry (ATR)*, *Activity Deployment Registry (ADR)* and *GLARE* Registration, Deployment, and Monitoring *(RDM)* Service, which handles requests and deployments (installations), and monitors different components. The system is deployed on all Grid sites in a VO to form a distributed framework. The distributed framework works based on a super-peer model. In contrast to hierarchical or centralized models, the super-peer model works well with dynamic and large-scale distributed environments such as computing Grids. Based on this model, some members (called super-peers) of smaller groups of Grid sites form a super group. A VO consists of one or more groups, who share distributed *activity types* and *deployments* with each other

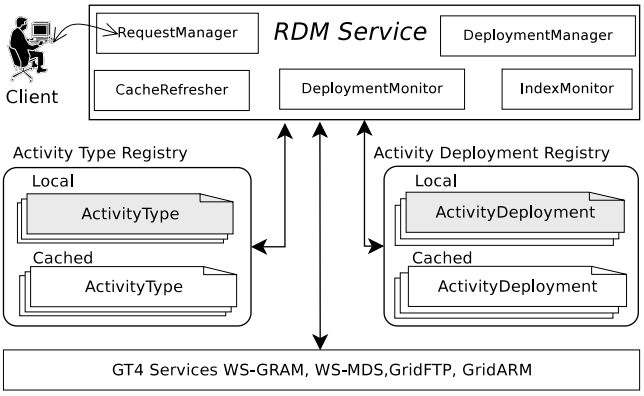

Figure 5: *GLARE* system Architecture.

through the super-peers. This model makes the *GLARE* framework more scalable and extensible. Furthermore, the automatic formation of super-peer model-based distributed framework makes *GLARE* self-managed.

3.1 Registry Services

Activity types and *deployments* are maintained in separate registries. Each occurrence of an *activity type* and *deployment* in a registry service is represented as a *WS-Resource*. A WS-Resource is a stateful web service which provides mechanisms including service lifecycle management, event registration and notification [3]. *Activity Types* are described in a hierarchy of abstract and concrete types and defined in terms of base *activity types*, *domains*, *functions*, *arguments*, *benchmarks* for different platforms and installation mechanism required for an on-demand deployment. *Abstract activity types* are used to discover *concrete activity types* and a *concrete type* identifies available *activity deployments*.

Activity Type Registry (ATR) maintains a set of named *activity types* in the form of WS-Resources organized in a hierarchy. It presents a more *abstract activity type* as root and uses it in discovering *concrete types*. *Concrete types* are installed on Grid sites and may have associated *activity deployments* and a reference to *deploy-file* describing steps involved in automatic installation.

Activity Deployment Registry (ADR) complements *Type Registry* and maintains a set of *activity deployments* of *concrete activity types* as WS-Resources. An *activity deployment* refers to an executable or a web/Grid service and provides information required for the *selection* and *instantiation* of a deployed (installed) *activity*. The Endpoint Reference (EPR) of each *activity deployment* resource is registered in its *type* resource presented in the *type registry*. Moreover, an *activity type* must be present in the *type registry* before registration of its *deployments*. The *type registry* service is responsible for discovering a matching *activity types*. In case of failure in discovering matching *activity type*, the *deployment registry* service requests the *type registry* service for the dynamic registration of a new *activity type*.

Both registry services are part of a distributed framework (explained in Section 3.3). They can access all resources registered on different Grid sites with *GLARE* services distributed all over the VO. A new *activity type* registered dynamically with one site can be discovered automatically by other sites. A resource discovered from a remote registry is

```
<DeploymentEPR>
  <Address>
    https://138.232.1.2:8084/wsrf/services/ActivityDeploymentRegistry
  </Address>
  <ReferenceProperties>
    <ActivityDeploymentKey> jpovray </ActivityDeploymentKey>
    <LastUpdateTime> Mar 30, 2005 16:41:24 </LastUpdateTime>
  </ReferenceProperties>
  <ReferenceParameters/>
</DeploymentEPR>
```

Figure 6: Deployment Endpoint Reference.

```
<ActivityDeployment name="jpovray" type="POVray">
  <executable> jpovray </executable>
  <Node name="schareck.askalon.org" home="$DEPLOYMENT_PATH"
      path="$DEPLOYMENT_PATH/bin"/>
  <function name="render" input="POVTextFile" output="PNGImageFile"/>
  <ActivityHistory>
    <LastExecution completionTime="3636" returnCode="1"
                   accessTime="Mar 30,2005 16:41:24"/>
  </ActivityHistory>
</ActivityDeployment>
```

Figure 7: A simple deployment representation.

optionally cached locally.

Both registry services provide an aggregation of all locally registered and cached resources, based on a WSRF service-group framework, in which aggregated resources are periodically refreshed. This enables the service to discover resources (*activity types* or *deployments*) by using standard XPath-based querying mechanism. In order to answer queries for *named resources* faster, the registry services use hash tables to access named resources. This eliminates XPath-based search requirements for *named resources* and significantly improves the performance.

3.2 GLARE RDM Service

The GLARE Registration, Deployment and Monitoring (RDM) service is the main frontend service which consists of components including *Request Manager*, *Deployment Manager*, *Cache Refresher*, *Index Monitor* and *Deployment Status Monitor*. The *Request Manager* receives and handles requests both from clients (in the form of queries) and from activity providers (in the form of updates). *Deployment Manager* performs on-demand deployment and installation of new *activities*.

Caching and Cache Monitoring: To ensure an efficient on-demand provision, *GLARE* supports a two-level cache; cache at normal Grid site and cache at *super-peer* Grid site, and provides a mechanism to refresh cache of updated resources. As shown in Fig. 5, cache of *activity types* and *activity deployments* is maintained in the *type registry* and *deployment registry* respectively, and the *Cache Refresher* updates cached resources if and when they change on the source Grid site. Outdated resources are discarded automatically.

The *Deployment Status Monitor* checks the status of each locally registered *activity deployment* and updates its resource and endpoint reference registered in the resource of its *type*. The *deployment Endpoint Reference (EPR)* contains an additional attribute *Last Update Time (LUT)* which is used by the Cache Refresher. Fig. 6 shows a deployment EPR in which the service *Address* and *ActivityDeploymentKey* elements are required to access *activity deployment* resource and do not change during the lifecycle of a *deployed activity*. *Last Update Time* can be changed frequently by the *Deployment Status Monitor* and each time it changes, cached *activity deployment* resources are revived.

The *Deployment Status Monitor* can register in local WS-GRAM service to get the latest metrics associated with *active* deployed *activities*. For instance, attributes like *last execution time*, *return code*, *last invocation time* etc. can be useful while scheduling and promising QoS. Fig. 7 represents a sample representation of *jpovray activity deployment*.

Deployment Leasing: The *GLARE service* provides the capability to lease an *activity deployment* with the help of *GridARM Reservation* service. A fine-grained reservation of a specific *activity* instead of the entire Grid site is supported. A user with valid *reservation ticket* is authorized to instantiate the *reserved activity*. A lease can be exclusive or shared. In case of an *exclusive lease* no one else is allowed to use the *activity*, during its leased timeframe. In case of *shared lease*, multiple clients can use the *leased activity* but *GridARM reservation* service ensures that the number of concurrent clients does not exceed the allowed limits and the required QoS are met.

Local Access: A distributed *GLARE* framework, as described in Section 3.3, ensures that clients of different Grid sites have the same view of the entire Grid or VO. An *activity* is discovered and provisioned by the local Grid site independent from the location of *activity deployments*. This is in contrast to the hierarchical model of MDS, in which a client has to contact root or the community Index service[4] in order to get the entire view of all Grid resources [10]. This enables clients to interact only with their local sites and get all distributed *activity types* and *deployments*. Clients don't have to consider or remember a centralized service and its access mechanism.

3.3 Self Management and Fault Tolerance

The *GLARE* framework is self-managed and fault tolerant. It uses Globus Toolkit 4 (GT4) built-in hierarchical aggregation and indexing mechanism to discover Grid sites and form peer groups. One member from each group becomes a super-peer and all super-peers form a super group. Within a group, a peer-to-peer interaction model is used, whereas inter-group communication is done through the super-peers. If some sites or services fail, the rest of the *GLARE* system continues working. A super-peer failure leads to the election of a new super-peer. Fig. 8 shows coherent peer and super-peer groups formed by using WS-MDS hierarchy.

Super-peer Election: *Index Monitor* is part of *GLARE RDM* service. It periodically probes the *GT4 Default Index* to see whether it is a *community index* or *local index*. A *GLARE* service on a site with *community index* becomes super-peer *election coordinator* and notifies all other Grid sites registered in the community. Notification is done twice (with a configurable time interval) and the second notification is acknowledged. A notification message includes number of registered Grid sites in the community index showing the community strength. A message from a smaller community is acknowledged in case of notifications from multiple indices. A responding site with higher *rank* is elected as

[4]In Globus Toolkit 4, terms *Default Index service* and *Community Index service* are used for local and root (VO-level) WS-MDS/MDS4 services respectively.

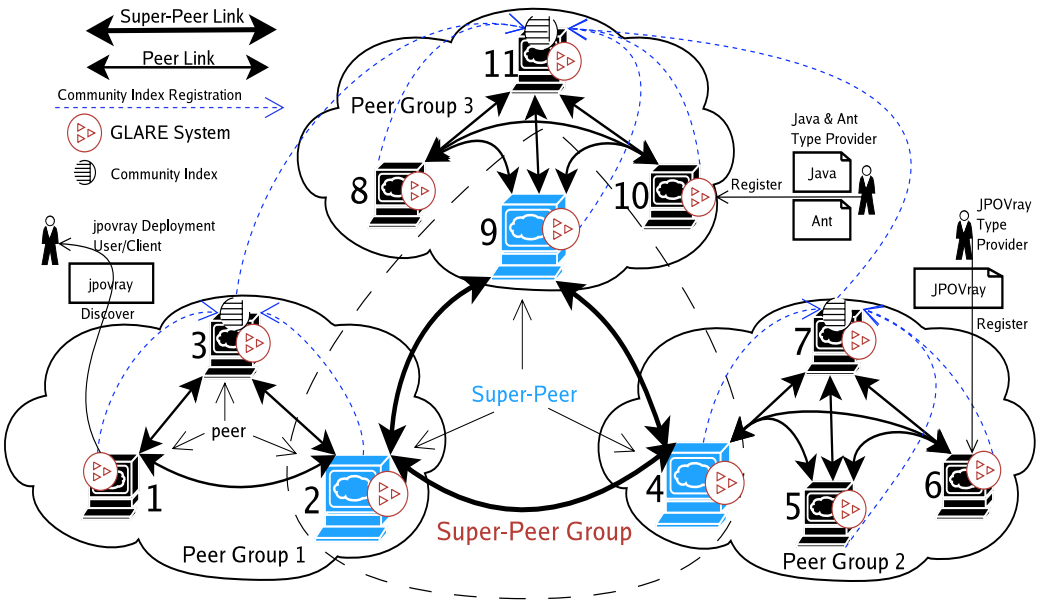

Figure 8: Peer and super-peer groups, formed after super-peer elections.

super-peer. Depending on the number of Grid sites, more than one sites can also be elected as super-peers and other members are then equally distributed among the elected super-peers. In this way each group can have exactly one super-peer. This group making is initially done by the *election coordinator* who notifies all elected super-peers about their group members after the completion of their election.

In order to rank different sites, a unique hashcode of all grid sites is calculated based on their static attributes. These attributes includes *processor speed*, *memory*, *uptime* and *site name*. Well established hashcode algorithms ensure the uniqueness when invoked by different *GLARE RDM* services residing on different sites. This unique hashcode is used as site *rank*.

Once a *GLARE* service recognizes itself as super-peer after receiving notification from the election coordinator, it does the following:

- Discovers other super-peers distributed in a larger community by interacting with community or *super community* indices.
- Handles requests from its peer members. A super-peer is contacted when other peers could not find information about some *activity types* or *deployments* within the group. It then forwards requests to other super-peers and caches the results.

Fig. 8 depicts the structure of a VO after election of super-peers with 3 peer groups and one super-group. Each member within a smaller group become peer of each other, whereas one member from each group joins a super-group as super-peer.

Once a member discovers that the super-peer is not working, it immediately calculates the *ranks* of all member sites, excluding the missing super-peer and notifies the *highest ranked member*. The highest ranked member then (a) verifies that the super-peer is missing (b) verifies its own rank and then (c) sends verification message to every other member. As a result each member again verifies the unavailability of the super-peer and acknowledges back to the highest ranked site. An acknowledgement from a simple majority confirms that the super-peer is no longer available, and the highest ranked site takes over as a new super-peer. In this way election and re-election of super-peers takes place, and high availability and scalability of the distributed *GLARE* system is ensured.

Furthermore, as both *activity types* and *deployments* are represented in the form of WS-Resources, they can be expired, refreshed or removed permanently. An activity provider can control the lifecycle of an *activity type* and its *deployments* by making a registration, cancelling it or revoking for certain time. Moreover, a provider can also specify minimum and maximum limits of *deployments* of an *activity* and the *GLARE* system ensures to fulfil the implied constraints. If an activity type expires, its deployments automatically expire, but an *active* (running) *deployment* at expiration time completes its execution. Moreover, if a deployment fails on one site, it can be moved to another site.

3.4 On-demand Deployment

Installation and deployment of scientific applications (*activities*) on different Grid sites is a time consuming and labour intensive task. *GLARE* provides a mechanism in which an activity provider can register new abstract or concrete *activity types* with an *installation* procedure in an associated deploy-file. A new *activity type* registered with one Grid site can be discovered by other sites and installed on-demand (automatically) based on constraints specified in the *type* description.

Also, simultaneous installation can be performed on multiple Grid sites, with least involvement of administrators or requesters. An installation procedure of *POVray* is depicted in Fig. 9.

Currently, installation with *autoconf* (configure, make, install) and auto build using *ant* is supported. The *deploy-file* and source URLs must be accessible by GridFTP for trans-

```
<ActivityTypeEntry name="POVray" type="Imaging">
  <Dependency>Java,Ant</Dependency>
  <Installation mode="on-demand">
    <Constraints>
       <platform> Intel </platform>
       <os> Linux </os>
       <arch> 32bit </arch>
    </Constraints>
    <DeployFile url="http://dps.uibk.ac.at/~mumtaz/deployfiles/povray.build"
                md5sum="...."/>
  </Installation>
</ActivityTypeEntry>
```

```
<Build baseDir="/tmp/papers/" defaultTask="Deploy" name="Povray">

  <Step name="Init"
    task="mkdir -p" baseDir="$DEPLOYMENT_DIR" timeout="10">
            <Env name="POVRAY_HOME" value="$DEPLOYMENT_DIR/povray/"/>
            <Env name="POVRAY_DIR" value="/tmp/povray/"/>
            <Property name="argument" value="$POVRAY_HOME"/>
            <Property name="argument" value="$POVRAY_DIR"/>
  </Step>

  <Step name="Download"
    depends="Init" task="$GLOBUS_LOCATION/bin/globus-url-copy"
    baseDir="$POVRAY_DIR" timeout="20">
            <Property name="source"   value="http://www.povray.org/ft...povlinux-3.6.tgz"/>
            <Property name="destination"  value="file:///$POVRAY_DIR/povray.tgz"/>
            <Property name="md5sum" value="....."/>
  </Step>

  <Step name="Expand depends="Download" task="tar xvfz"
        baseDir="$POVRAY_DIR" timeout="10">
            <Property name="argument" value="$POVRAY_DIR/povray.tgz"/>
  </Step>

  <Step name="Configure  depends="Expand" task="./configure"
          baseDir="$POVRAY_DIR/povray-3.6.1/" timeout="300">
            <Property name="argumen" value="COMPILED_BY=Glare Service glare@dps.uibk.ac.at"/>
            <Property name="argument" value="--prefix=$POVRAY_HOME"/>
  </Step>

  <Step name="Build" depends="Configure" task="make"
          baseDir="$POVRAY_DIR/povray-3.6.1" timeout="200">
  </Step>

  <Step name="Deploy" depends="Build" task="make install"
          baseDir="$$POVRAY_DIR/povray-3.6.1" timeout="30">
            <!--Dialog expect="Do you want to continue(Yes/no)?" send="yes"/-->
            <Dialog expect="Install as root or normal user (R/U):" send="U"/>
  </Step>
</Build>
```

Figure 9: *JPOVray* activity type description and Deploy-file with steps for automatic deployment.

fers to the target Grid site. An activity provider can specify different constraints which must be fulfilled before the installation, for example, prerequisite platform and operating system etc. An activity can be restricted to a certain number of sites or revoked temporarily. An activity provider can use default environment variables `DEPLOYMENT_DIR`, `USER_HOME`, `GLOBUS_SCRATCH_DIR` and `GLOBUS_LOCATION` in the deploy-file, and *RDM service* substitutes their values.

After successful installation, the *activity type* is marked as deployed and specified executables or services are registered in the *deployment registry* in the form of *deployment* WS-Resources. The *deployments* are identified in the *activity type* description by the activity provider, or *GLARE* service can automatically find, for instance by exploring *bin* sub directory of the *deployed activity home*. In case of failure, or installation *mode=manual GLARE* service notifies administrator of the target site by email referring to the website of the activity or contact of its provider. *Automatic* deployment eliminates the overhead of manual or *on-demand* deployment. But, in order to control unwanted installations on different sites, only constraint-based or on-demand deployment is supported. A smart scheduler can reduce overhead of on-demand deployment by providing intelligent look-ahead scheduling.

Deployment Handler: is an *Expect*[5] [17] based virtual terminal used to automatically interact with operating systems of different Grid sites and perform interactive process of local or remote installation. *GLARE* uses local shell (e.g. bash) or glogin[6] [25] to login on target site securely with the *expect* mechanism. As an alternative to *glogin*, the deployment handler can use GRAM on target Grid site and issues commands in the form of GRAM jobs. By default, system's local shell is used by the *GLARE* service running on target Grid site. We also exploit *Expect* for interactive installation. For instance, the installation of *POVray* requires human interaction and prompts for license acceptance, user type, and install path, and activity provider specifies this interaction dialog in *deploy-file* in the form of *send/expect patterns* as shown in Fig. 9.

GLARE system is designed as a set of WSRF services distributed in the Grid with platform-independent interaction mechanism. This makes it acceptable for both Grid and web services technologies. The openness of underlying infrastructure and super-peer model based design makes *GLARE* a scalable middleware that shields application developers from the Grid. Furthermore, automatic super-peer election and activity installations upgrade *GLARE* system to become self managed and fault tolerant. In the following section we show some experiments which demonstrate the effectiveness of the *GLARE*.

4. EXPERIMENTS

We have implemented a prototype of the *GLARE* system based on GT4 and integrated in ASKALON Grid environment [18]. We then deployed it on different sites of Austrian Grid infrastructure [8]. The Austrian Grid is a national computing Grid infrastructure distributed across several cities and institutions across Austria. The infrastructure is composed of more than ten Grid sites that aggregate over 200 processors. Each local Grid site system administrator independently installed his favourite local job manager and the Globus toolkit (GT2 or GT4) for integration within the Austrian Grid.

We have evaluated *on-demand* deployment of new activities and calibrated the deployment overhead for some real world scientific applications. We have selected three applications; *Wien2k* [7] (pre-compiled) which performs electronic structure calculation of solids based on density functional theory. *Invmod*, a hydrological application for river modelling which has been designed for inverse modelling calibration of the WaSiM-ETH program [28], and *counter*, a sample GT4 service that explores GT4 features and used here to demonstrate the deployment of a Grid service. Table 1 illustrates time spent in different operations and components of the *GLARE* framework. On-demand deployment is performed in two ways; with JavaCoG (using GRAM and GridFTP) and with *Expect* by programmatically acquiring local system shell and automatizing the installation process. Communication and deployment overhead depends on the size of installation files and compilation respectively. As shown in the Table 1, the registration of a new *type* and its *deployments* and notification to the site administrator imply reasonable costs. Downloads take some time but significant

[5]Expect is a method of automating interactive applications/tools like telnet, ftp, passwd etc.

[6]Glogin is a secure shell that uses standard Globus GRAM and GSI mechanism, i.e. the users can use their proxy certificates to log into a remote Grid site, without any additional server running (as gsissh).

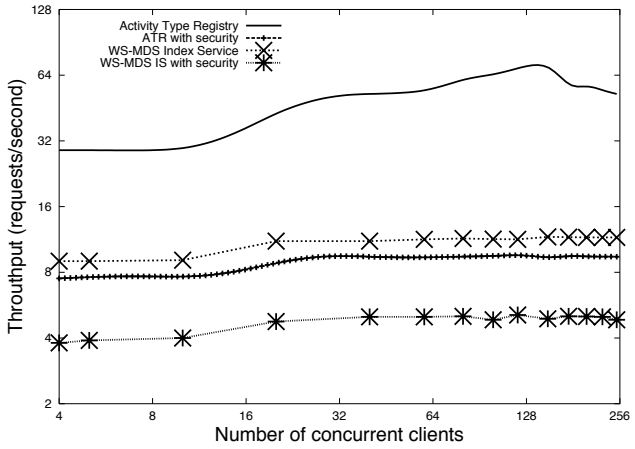

Figure 10: Comparison of Activity Type Registry and WS-MDS Index Service both with and without transport level security. Throughput with varying number of concurrent clients.

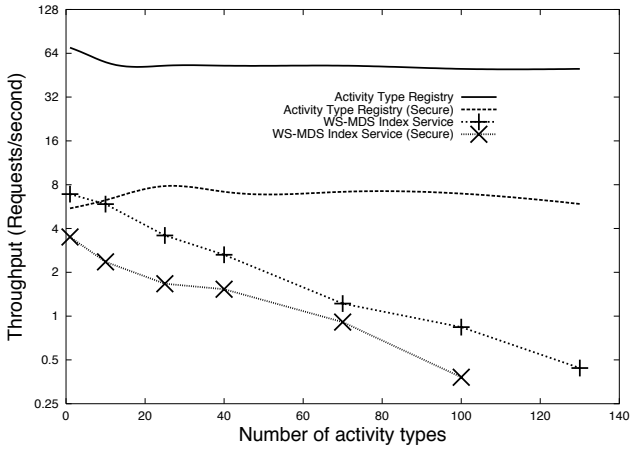

Figure 11: Performance comparison with an increasing number of activity types.

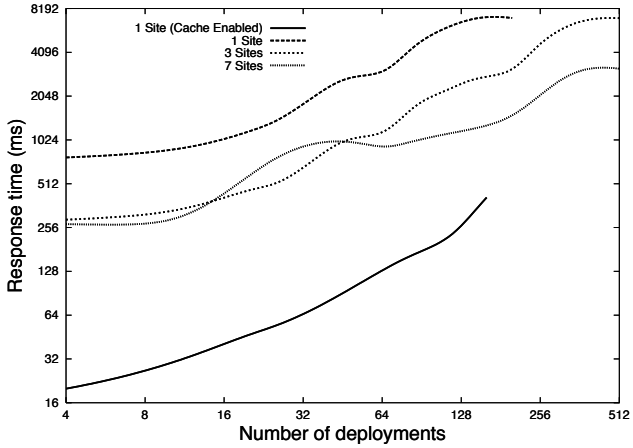

Figure 12: Response time per activity *deployment* request with cache on 1 Grid site and without cache on 1, 3 and 7 Grid sites.

time is spent in compilation and installation. Also, *Expect* is more efficient than Java CoG. The overall scheduler overhead shown in the Table, can be eliminated by employing automatic deployment, or reduced by providing a *schedule-ahead* mechanism by the scheduler.

We tested the efficiency, performance and scalability of the *GLARE* system by deploying it on up to 7 Austrian Grid sites. We compared an integral component of the *GLARE* framework, that is, *Activity Type Registry* with the GT4 *Index Service* (WS-MDS) by registering multiple *activity type* WS-Resources in both services. We performed experiments with and without transport level security enabled (i.e. with http and https). Note that, although Index Service is normally used for physical resources but the underlying aggregation framework (WSRF-based GT4 aggregation framework) is same for both GT4 Index service and *GLARE* registries. Therefore it is logical to make this comparison.

Fig. 10 shows performance of both services with and without security enabled. Throughput decreases almost by 50% for both services with transport level security. Index Service is 50% slower than Activity Registry because of its XPath-based querying mechanism. This experiment was performed with both WS-MDS *Index* and *activity type registry* services running on the same Grid site with same number of registered activity types, whereas clients were distributed among 7 other sites.

Fig. 11 shows a comparison of *Activity Type registry* with *Index Service* with a varying number of *activity type* resources in the registry and index service, again, with and without security. Throughput of *Index* Service decreases significantly with increasing number of resources whereas it can be observed that throughput of an *activity type registry* is consistent. A good performance comparison of previous versions of MDS is given in [29, 42].

We also experienced the scalability of the system on 1, 3 and 7 Grid sites, with and without cache enabled. Fig. 12 shows response time per request for a list of *deployments* associated with an activity *type*. Deployment entries are equally distributed on all involved sites. It is observed that there is a significant improvement in performance by increasing number of sites or by enabling the cache.

Fig. 13 shows the change in the 1-minute load average as the number of clients (requesters) and event notification listeners (sinks) increases; the load average is measured as the load on the *Activity Type Registry* during the last minute (using Unix *uptime* command). The load average is therefore a measure of the number of jobs waiting in the run queue. The highest load average occurs when the notification rate is 1 sec. It peaks slightly above 16 corresponding to 210 sinks. Load average is proportional to the notification rate. The load average against the number of requesters peaks just below 5, which shows consistency.

Finally, we have evaluated that sometimes *Index Service* stops responding when we register more than 130 *activity type* resources in it and number of concurrent clients exceeds 10 (Fig. 11). This is quite strange behaviour and could be a real shortcoming of the index service, which may become a bottleneck when registered number of Grid sites increases. In contrast, the *GLARE* registry services works well with a reasonable large number of registered resources.

Deployment Method	Operation/Overhead	Wien2k	Invmod	Counter
Expect	Activity Type Addition	633	632	665
	Communication Overhead	1,667	1,381	1,279
	Activity Installation/Deployment	8,068	27,776	29,843
	Activity Deployment Registration	355	350	352
	Notification	345	345	345
	Expect Overhead	2,100	2,100	2,100
	Total overhead for meta-scheduler	11,068	30,484	32,484
Java CoG	Activity Type Addition	633	632	665
	Communication Overhead	5,600	2,500	2,400
	Activity Installation/Deployment	18,068	49,700	39,756
	Activity Deployment Registration	355	350	352
	Notification	345	345	345
	JavaCoG Overhead	9,800	9,900	9,800
	Total overhead for meta-scheduler	25,001	53,527	43,518

Table 1: Time spent (in ms) in different operations.

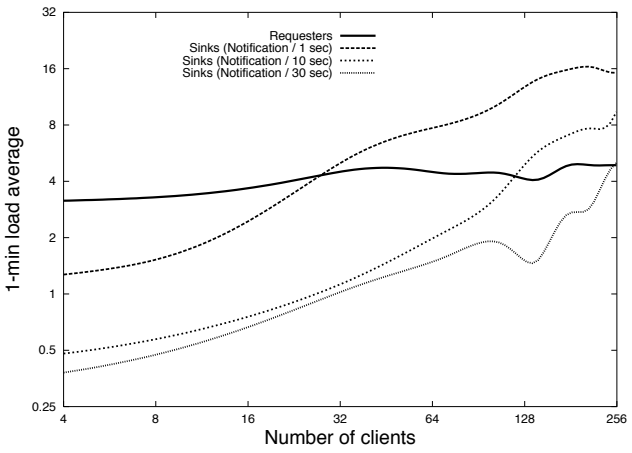

Figure 13: Average 1-min CPU load with various number of concurrent client and notification sinks.

5. RELATED WORK

A number of efforts have been made within the Grid community to develop automatic Grid resource management and brokerage solutions but very few of them are addressing the issue of resource management covering software components (activities) and their automatic deployment. A separation between meaning, behaviour, and implementation of the Grid application components is described in [31]. The work in [23] matches a high-level application specification to an optimal combination of available components. In contrast, *GLARE* provides a high-level application specification in a hierarchy of *activity types* and provides dynamic registration and automatic deployment of software components.

Pegasus [15] uses Chimera [22] and Transformation Catalog [14] for transforming an abstract workflow into concrete workflow. The transformation Catalog is used to map a logical representation of an executable (transformation) to a physical representation, which describes its functionality and accessibility. The catalog uses MySQL as a centralised backend database. Chimera Virtual Data System [22] describes and stores data derivation procedures and derived data in a central database. It provides a special language interpreter that translates user requests. This system is useful for datagrid applications, but works with a dedicated querying mechanism. Pegasus uses Globus middleware services and automates replica selection. It does not provide automatic/on-demand deployment of software components.

GrADS [12] resource selection framework [30] addresses the discovery and configuration of physical resources that match application requirements. It provides a declarative language using set matching techniques, which extend Condor matchmaking [40] and support both single and multiple resource matching. This system does not cover Grid application components. S. Decker et al describe in [39] Grid resource matching using semantic web technologies. This work proposes physical resource matching by using ontologies, background knowledge and rules. It highlights the need of semantic description of Grid resources and resource matching but does not address issues of performance and efficiency. Both systems [12, 39] do not cover software resources like Grid computational activities.

CrossGrid [9] provides a distributed component registry with peer-to-peer technology. It supports inter-registry communication for maintaining table coherency. Grimoire [27] extents UDDI [32] to provide invocable activities such as workflows or legacy programs. GridLab capability registry [35], CrossGrid component registry [9] and MyGrid Grimoire [27] provide registries for static information of the Grid applications. UDDI [32] and Handle System [20] can be used to augment our system but they have their own limitations. UDDI is a specification for distributed web-based information registries for web services but unsuitable for legacy scientific applications. Also it does not support dynamic updates. Handle System supports a very basic querying mechanism. Furthermore, it requires domain specific naming authorities to be registered in a root naming authority which is not managed efficiently. Globus MDS [10] provides a hierarchical aggregation framework for distributed Grid resources.

The main difference between *GLARE* and the above systems is that while most of the above systems focus on discovering and brokering physical resources, *GLARE* framework focuses on software components (activities). Furthermore, *GLARE* provides dynamic registration, automatic deployment and on-demand provision and leasing of logical resources. The framework is self-managed, fault tolerant, distributed and scalable. In contrast to MDS, *GLARE* pro-

vides a super-peer model based distributed framework which works well for large scale environments. It is implemented in Globus Toolkit 4 an implementation of a new Web-Services Resource Framework [3].

Ka-tools [5], LCFG [4] and Quattor [34] provide auto deployment but mostly deal with configuration of physical nodes or perform OS cloning in a fabric. SmartFrog [1] requires specific components or wrappers to support automatic deployment of software components.

Global Grid Forum CDDLM working group [21] is addressing issues of automatic deployment and provisioning of Grid services with security and fault tolerance. The focus of this group is how to describe configuration of services, deploy them on the Grid and manage their deployment lifecycle (instantiate, initiate, start, stop, restart, etc.). The group is also standardising APIs for this purpose. The focus of the group is a WSRF-based Grid services whereas *GLARE* targets both Grid services and legacy scientific applications.

6. CONCLUSION

Grid resource management systems so far have been mostly used for brokerage of Grid computers. In our work we focused on extending resource management to application components that can be part of distributed Grid applications. We introduced *GLARE*, a Grid-level application component registration, deployment and provisioning framework that provides dynamic registration, automatic deployment and on-demand provision of application components (activities) that can be used to build Grid applications. Application components are described as *activity types* and *activity deployments*. By separating *activity types* from *activity deployments* we can shield the application developer from the Grid. *GLARE* automatically relates *activity types* to a set of *activity deployments* that can then be selected for instance by the middleware to create a Grid application for execution. Moreover, *GLARE* provides a mechanism in which new activities can be registered dynamically, installed automatically and provisioned and leased on-demand. We believe that this functionality is a major step forward towards an invisible Grid from the application developer point of view.

We examined the performance of the *GLARE* system and compared its registries with GT4 index service (WS-MDS) and found it quite encouraging. We also exhaustively verified the efficiency of the registration and provisioning mechanism with varying number of *activity types*, *activity deployments* and concurrent clients.

We plan to incorporate the GGF recommendation for automatic configuration and deployment of *Grid services*, once they become standard. We are considering to add features of un-deployment and generation of wrapper services for legacy code by integrating with the Otho toolkit [26]. Also as a future work, we plan to augment *activity types* with ontological description so that activity types can be searched for based on a semantic description. Similar as proposed for physical resources [37].

GLARE simplifies the description and presentation of both *activity types* and *deployments* in such a way that they can be easily located in a distributed Grid environment and thus become available on-demand. A mechanism is supported to lease an *activity deployment* for a certain timeframe. The *GLARE* ensures that a leased *activity* remains available and provides required QoS during the leased timeframe. *GLARE* has been implemented based on a super-peer model with support for self management and fault tolerance. Experiments have been shown to demonstrate the performance and efficiency of the proposed system.

Acknowledgements

We thank our colleagues especially Radu Prodan and Jun Qin for their constructive discussions and the anonymous reviewers for pointing us to additional related work.

7. REFERENCES

[1] http://sourceforge.net/projects/smartfrog/.

[2] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002.

[3] Globus Alliance. WS-Resource Framework. http://www.globus.org/wsrf.

[4] Paul Anderson and Alastair Scobie. LCFG: The Next Generation. In *UKUUG Winter Conference*. UKUUG, Bristol, July 4-7 2002.

[5] Philippe Augerat, Wilfrid Billot, Simon Derr, and Cyrille Martin. A scalable file distribution and operating system installation toolkit for clusters. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, May 21 -24 2002. IEEE Computer Society Press.

[6] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive Computing on the Grid Using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[7] P. Blaha, K. Schwarz, G. Madsen, D. Kvasnicka, and J. Luitz. *WIEN2k: An Augmented Plane Wave plus Local Orbitals Program for Calculating Crystal Properties*. Institute of Physical and Theoretical Chemistry, Vienna University of Technology, Vienna. ISBN 3-9501031-1-2, 2001.

[8] The Austrian Grid Consortium. http://www.austriangrid.at.

[9] CrossGrid. http://www.crossgrid.org/.

[10] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE International Symposium on High- Performance Distributed Computing(HPDC-10)*. IEEE Press, 2001.

[11] Karl Czajkowski, Ian Foster, Nick Karonis, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer Verlag, 1998. Lect. Notes Comput. Sci. vol. 1459.

[12] Holly Dail, Otto Sievert, Francine Berman, Henri Casanova, Asim YarKhan, Sathish Vadhiyar, Jack Dongarra, Chuang Liu, Lingyun Yang, Dave Angulo,

and Ian Foster. Scheduling in the Grid Application Development Software Project. *Resource Management in the Grid: Kluwer*, 2003.

[13] Rubing Duan, Radu Prodan, and Thomas Fahringer. DEE: A distributed fault tolerant workflow enactment engine for grid computing. In *Proceedings of the High performance computing and communication (HPCC05)*, Sorrento, Naples Italy, 2005.

[14] Ewa Deelman et. al. Transformation Catalog Design for GriPhyN. Technical report griphyn-2001-17, 2001.

[15] Ewa Deelman et. al. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing, LNCSD9, ISSN 1570-7873*, 1:25–39, 2003.

[16] Ken Kennedy et.al. New Grid Scheduling and Rescheduling Methods in the GrADS Project. In *International Parallel and Distributed Processing Symposium, Workshop for Next Generation Software*. IEEE Computer Society Press, April 2004.

[17] Expect. http://expect.nist.gov/.

[18] Thomas. Fahringer. ASKALON - A Programming Environment and Tool Set for Cluster and Grid Computing. http://dps.uibk.ac.at/askalon, Institute for Computer Science, University of Innsbruck.

[19] Thomas Fahringer, Jun Qin, and Stefan Hainzer. Specification of Grid Workflow Applications with AGWL: An Abstract Grid Workflow Language. In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid 2005 (CCGrid 2005)*, Cardiff, UK, May 9-12 2005. IEEE Computer Society Press.

[20] Corporation for national Resource Initiatives. Handle System. http://www.handle.net.

[21] Global Grid Forum. CDDLM - Configuration, Deployment and Lifecycle Management of grid services. http://forge.gridforum.org/projects/cddlm-wg.

[22] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: A Virtual Data System For Representing, Querying, and Automating Data Derivation. In *14th International Conference on Scientific and Statistical Database Management (SSDBM 2002)*, September 2002.

[23] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of component-based applications within a grid environment. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM Press, November 10-16 2001.

[24] The Globus Alliance. http://www.globus.org.

[25] GLogin. http://www.gup.uni-linz.ac.at/glogin/.

[26] Juergen Hofer, Alex Villazon, Mumtaz Siddiqui, and Thomas Fahringer. The Otho Toolkit: Generating tailor-made scientific grid application wrappers. In *Proceedings of 2nd International Conference on Grid Service Engineering and Management (GSEM'05)*, Erfurt, Germany, September 19-22 2005.

[27] IT Innovation. Workflow enactment engine, October 2002. http://www.it-innovation.soton.ac.uk/mygrid/workflow/.

[28] K. Jasper. Hydrological Modelling of Alpine River Catchments Using Output Variables from Atmospheric Models. Phd thesis, eth zurich, 2001 diss. eth no. 14385., 2001.

[29] H.N. Lim, Choi Keung, J.R.D. Dyson, S.A. Jarvis, and G.R. Nudd. Performance Modelling of a Self-Adaptive and Self-Optimising Resource Monitoring System for Dynamic Grid Environments. In *AHM2005, Fourth All Hands Meeting, Nottingham*, 2005.

[30] Chuang Liu, Lingyun Yang, Ian Foster, and Dave Angulo. Design and Evaluation of a Resource Selection Framework for Grid Applications. In *HPDC-11, the Symposium on High Performance Distributed Computing*, Scotland, 2002.

[31] A. Mayer, S. McGough, M. Gulamali, L. Young, J. Stanton, S. Newhouse, and J. Darlington. Meaning and behaviour in grid oriented components. In *3rd International Workshop on Grid Computing, Grid 2002, volume 2536 of Lecture Notes in Computer Science, Baltimore, USA*, November 2002.

[32] OASIS. UDDI: The Universal Description, Discovery and Integration. http://www.uddi.org/about.html.

[33] POVray. http://www.povray.org/.

[34] Piotr Poznanski, German Cancio Melia, Rafael Garca Leiva, and Lionel Cons. Quattor - a framework for managing grid-enabled large-scale computing fabrics. In *In workshop of the 4th Cracow Grid Workshop 2004*, December 2004.

[35] Ed Seidel, Gabrielle Allen, Andre Merzky, and Jarek Nabrzyski. Gridlab: A grid application toolkit and testbed. *Future Generation of Computer Systems*, 18(8):1143–1153, 2002.

[36] Mumtaz Siddiqui and Thomas Fahringer. GridARM: Askalon's Grid Resource Management System. In *European Grid Conference (EGC 2005)*, Lecture Notes in Computer Science. Springer Verlag, February 2005.

[37] Mumtaz Siddiqui, Thomas Fahringer, Juergen Hofer, and Ioan Toma. Grid resource ontologies and asymmetric resource-correlation. In *Proceedings of 2nd International Conference on Grid Service Engineering and Management (GSEM'05)*, Erfurt, Germany, September 19-22 2005.

[38] Domenico Talia and Paolo Trunfio. Toward a synergy between p2p and grids. *IEEE Internet Computing*, 7:94–96, 2003.

[39] H. Tangmunarunkit, S. Decker, and C. Kesselman. Ontology-based Resource Matching in the Grid–The Grid meets the Semantic Web. In *Second International Semantic Web Conference, Sanibel-Captiva Islands, Florida*. October 2003.

[40] The Condor Team. http://www.cs.wisc.edu/condor/.

[41] Jia Yu and Rajkumar Buyya. A Taxonomy of Workflow Management Systems for Grid Computing. Technical report grids-tr-2005-1, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, Mar 2005. http://www.gridbus.org/.

[42] Xuehai Zhang and Jennifer M. Schopf. Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2. In *Proceedings of the International Workshop on Middleware Performance (MP 2004), part of the 23rd International Performance Computing and Communications Conference (IPCCC)*, 2004.