An Abstraction Model for a Grid Execution Framework

Kaizar Amin,[c,a] Gregor von Laszewski,[*,a,b] Mihael Hategan,[b,a]
Rashid Al-Ali,[d] Omer Rana,[d] and David Walker[c]

[a]Argonne National Laboratory, Argonne, IL 60439, USA

[b]University of Chicago, Chicago, IL 60637, USA

[c]University of North Texas, Denoton, TX 76203, USA

[d]Cardiff University, Wales, UK

Abstract

Computational Grids have been identified as one of the paradigms revolutionizing the discipline of distributed computing. The contributions within the Grid community have resulted in new Grid technologies and continuous improvements to Grid standards and protocols. Though crucial to the success of the Grid approach, such an incremental evolution of Grid standards has become a primary cause of frustration for scientific and commercial application communities aspiring to adopt the Grid paradigm. Motivated by our rich experience and the need to decouple the application development and the Grid technology development processes, we propose an abstraction-based Grid middleware layer as part of the Java CoG Kit. In this paper, we showcase our abstraction model and verify its extensibility by integrating it with an advanced quality-of-service–based execution framework.

Keywords: Grid computing; Java CoG Kit; Grid abstractions; Grid execution patterns; Grid quality of service (QoS)

With the recent advancements in Grid technologies, a majority of the Grid community has focused on the development of advanced Grid protocols and sophisticated Grid services. These Grid services include execution services, data management services, information services, workflow services, quality-of-service (QoS) management services, fault-tolerance services, security services, and Grid infrastructure management services. At the same time, a large number of elementary commercial and scientific applications have been identified and implemented reusing available Grid services. Intuitively, it should be sufficient for Grid applications to directly invoke Grid services and employ their functionality. However, applications developed with such a two-layered model are severely restricted in their adaptability, extensibility, and stability for several reasons. First, Grid protocols and services are not standardized yet. Hence, integrating Grid protocols within applications increases their maintenance because of continuous changes in the "evolving" Grid standards. Second, changes in the Grid computing models [1, 2, 3] can render applications using a particular technology obsolete. Third, incorporating advanced Grid semantics and implementation dependencies directly within the applications can introduce unnecessary complexities, thereby shifting the focus from its primary commercial or scientific objective. Hence, there is an uncontested need for an additional layer bridging the Grid services layer (such as provided by the Globus Toolkit [4]) and the application layer (see Figure 1). This intermediate layer represents application development frameworks such as the Java Commodity Grid (CoG) Kit [5]. We refer to this layer as the Grid abstraction layer [6].

The Grid abstraction layer offers high-level abstractions[7] to Grid applications, shielding them from the technical and semantic complexities of various Grid implementations [3, 4, 8, 9, 10]. The Grid abstraction layer can adapt to many changes and enhancements to Grid protocols and the Grid architec-
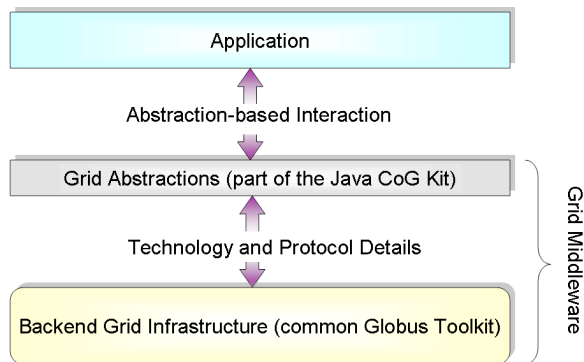
Figure 1: The Grid middleware layer decouples the application development process and the Grid service development process.

ture without reflecting them in the offered abstractions. As a result, applications developed with such abstractions do not have to be actively maintained or updated to become compatible with developing Grid technologies [2, 11], thereby providing long-term investment protection.

This paper describes the design and implementation of the Grid abstractions offered by the Java CoG Kit. We outline the key components of our abstractions and describe the important patterns in our abstraction-based execution framework. Our Grid abstraction also provides pluggable functionality extensions, allowing the Grid community to incrementally interface to new Grid middleware once they become available, and to develop new abstractions if desired. We demonstrate the extensibility of our framework by showcasing the integration of Quality of Service adaptors for job submission, allowing our execution framework to support QoS-enabled Grid execution without requiring modifications in the applications employing our abstractions.

The rest of this paper is organized as follows. Section 1 presents a brief overview of several research projects working toward provisioning advanced Grid middleware. Section 2 gives a detailed overview of the various abstractions offered by the Java CoG Kit and their intended use. Section 3 discusses the extension of our Abstractions with QoS adaptors. Section 4 summarizes the current status of the project.

1 Related Research

The Java CoG Kit has its origin in a Grid like application development framework dating back to 1994 [12, 13, 14]. Since then a variety of Grid and Grid abstraction frameworks have been discussed in literature [15, 16, 17, 18, 19, 20, 21]. However, most of the available frameworks concentrate on providing an integrated API that is built to support a particular Grid tool, rather than focusing on an abstraction model. Frameworks that do not adopt an abstraction model are tightly bound to the semantics of a specific Grid technology and exhibit the same shortcomings described in the previous section, such as increased maintenance, additional complexity, and lack of investment protection. In this section we discuss some of the most prominent Grid middleware frameworks that share our philosophy on the importance of an abstraction-based execution framework.

The Grid application framework for Java (GAF4J) [20] is a prototype developed as a part of IBM's Grid initiative [22]. It abstracts several Grid semantics from the application logic and provides a high-level programming model. At the base of its abstraction model is the notion of a generic Grid task. Grid-unaware applications formulate a Grid task and a corresponding task definition. This task and its definition are then submitted to a task execution client, which queues the task with the task dispatcher and updates its status with a console component. The task dispatcher services the dispatching queue with a first-in-first-out (FIFO) policy. It also communicates with a resource broker to identify suitable Grid resources for the execution of the task. Once the appropriate matchmaking is accomplished, the task and the Grid resource are submitted to the job-starter. The job-starter prepares the task based on its definition and submits it to the remote Grid resource, monitoring its status. In its current implementation, the job-starter component submits Grid jobs to Globus Toolkit [4] v2.4-enabled resources using an older version of the Java CoG Kit (version 0.9.13). In case GAF4J would be modified to use our newer versions of the Java CoG Kit, one could enable support for various other Grid implementations without extensive changes in the application. However, most of its abstractions (and more) are already included in the Java CoG Kit.

The Grid Application Toolkit (GAT) [21] of the GridLab project [23] is a set of extensible, abstract APIs for performing various Grid functions such as job execution, file transfer, information query, and

2

data management. The GAT abstraction model is based on the concepts of capability, capability adaptors, and capability invocation. The GAT model allows for the creation of several capability adaptors, which provide the necessary bridge between some backend functionality, referred to as capability, and a set of abstract GAT APIs. During the startup phase of the GAT framework, each of the available capability adaptors registers the abstraction APIs that it supports with a central capability registry. On invocation of any GAT API, the functionbinding component queries the capability registry for a list of relevant adaptors that support the API. Once the appropriate adaptor is chosen, the API invocation is delegated to the adaptor to access the specific capability. The GAT model offers an extensible plug-and-play solution, whereby the utility of the framework can be incrementally improved by the provision of additional capability and corresponding adaptors. The main advantage of GAT is that its abstractions are based on the C programming language and appropriate Java bindings are available. However, this is also its disadvantage as the Java CoG Kit is explicitly developed to match the framework provided by Java. The CoG Kit effort seeks to implement similar abstraction between Python and Java in the future.

The Simple API for Grid Applications (SAGA) research group of the Global Grid Forum (GGF) [24] is combining the benefits offered by several Grid abstraction frameworks into a well-defined GGF standard. Members from different research and commercial projects are participating in this activity. We believe that several novel features of our architecture will contribute significantly toward this effort, as evident from our presentations at various invited participations at the Global Grid Forum [25, 26].

2   Grid Abstractions

The Grid abstractions offered by the Java CoG Kit have been designed with a top-down approach [27, 14, 28]. Rather than exposing the complex Grid functionality to an application in a nonintuitive fashion, we outline the most general and basic functionalities desired by a number of applications and provide appropriate interfaces. With this approach, applications do not have to be penalized for limitations in the existing Grid implementations. Instead, the approach decouples the application development and the Grid development processes. In the rest of this section we describe important abstractions of our

framework.

2.1   Task

At the core of our abstraction model is the notion of a *task*. It is an atomic unit of Grid execution. Examples of Grid tasks include remote job execution, file transfers, file operations, information query, and resource reservation. Every task is annotated in a way specific to its functionality. A task acts as a container for several related abstractions:

- *Identity:* A uniform resource name (URN) that uniquely identifies the task.

- *Specification:* A detailed description specifying the details of a task including parameters and other requirements. The semantics of the specification depend on the type of the task. For example, a job execution specification describes important parameters required for job execution, whereas a file transfer specification specifies the source and destination files along with several other relevant attributes.

- *Security Context:* A security credential associated with the Grid task. This credential is used to authenticate the task to the backend Grid implementation. Grid implementations may use different security mechanisms. Thus, the security abstraction in our framework represents a generalized credential container that can be cast to a specialized representation relevant to a particular Grid implementation.

- *Service Contact:* The endpoint address of a backend service that will be used to execute this task. For example, if this task is a job execution task, its service contact represents a job execution service. Similarly, for a file transfer task, it represents a file transfer service.

- *Status:* The progress of the task execution, allowing the application to monitor its execution. The task can be in any one of the following states: unsubmitted, submitted, active, suspended, canceled, failed, or completed. Figure 2 shows the state transition diagram of the task status in our abstraction model.

- *Provider:* The provider backend used for this task. The provider attribute plays a key role

in translating the abstract task into protocol-specific constructs. For example, an application can formulate an abstract Grid task and assign it a provider as *GT2*, implying that it is to be executed with a Globus Toolkit v2 (GT2) service. The service contact, security context, and the specification are translated into GT2 protocol-specific constructs and executed accordingly. For the same task to be executed with a Globus Toolkit v3 (GT3) service or a Unicore service, the application merely needs to change its provider attribute accordingly. Thus, the same abstract task can be executed by multiple Grid implementations and architectures based on the application requirement and implementation functionality.
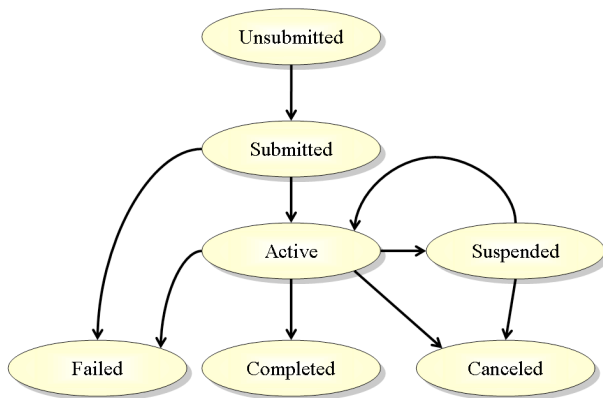

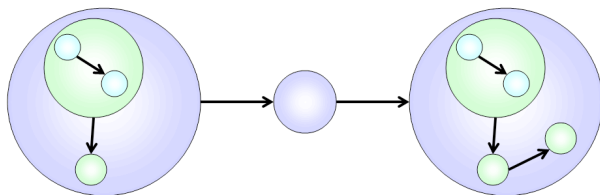Figure 2: State transition diagram of the task and task graph status.

Figure 3: Hierarchical task graph.

2.2 Task Graph

A Grid task provides the necessary ability to represent a unit of execution on the Grid. Nevertheless, advanced applications require a more sophisticated execution framework that facilitates complex execution patterns and dependencies. The task graph represents a directed acyclic graph (DAG) that allows applications to express execution (control) flows between multiple tasks modeled as DAGs.

Task graphs can support arbitrary levels of hierarchy. In other words, every node in the task graph can be either a task or another task graph (see Figure 3). Hierarchies in a task graph enable the Grid applications to conveniently modularize the execution components without introducing unnecessary complexity.

A task graph has an associated status that represents the collective execution status of all its subnodes. The task graph status is determined by the logic depicted in Figure 4. Additionally, we provide extensions to the basic task graph abstractions to form some common utility objects such as queues and sets. A queue is a task graph with a predefined FIFO execution ordering. As shown in Figure 5, every element of the queue can be either a task or a task graph, hence the concept of hierarchical queues. A set is a special task graph with no dependencies, thereby offering a collective mechanism of parallel execution of a hierarchy of tasks.

Figure 4: Pseudocode to determine the status of a task graph

```
if (any Task in the TaskGraph has failed) {
  status = failed
} else if (all tasks are unsubmitted) {
  status = unsubmitted
} else if (any task is suspended) {
  status = suspended
} else if (any task is active) {
  status = active
} else if (any task is submitted) {
  status = submitted
} else if (every task is completed) {
  status = completed
}
```

Another notable aspect of a task graph is that it is a *checkpointable* entity. That is, the execution of every node of the task graph evolves into a new checkpoint status. Re-execution of a failed task graph will resume the execution from the last valid checkpoint status (see Figure 8). This type of graph checkpointing not only allows applications to restart failed execution flows but also allows them to suspend executions
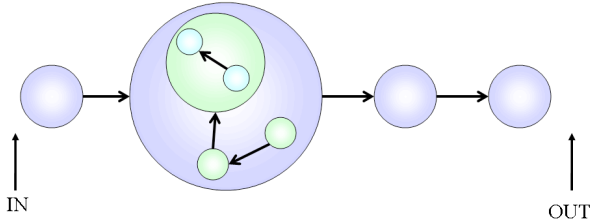
IN   OUT

Figure 5: A hierarchical queue is a special case of hierarchical task graph with a FIFO execution dependency.
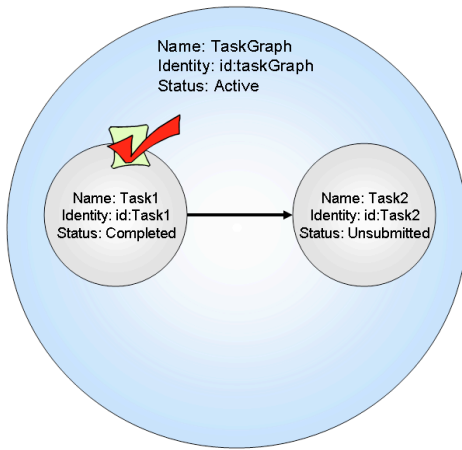

Figure 6: An active task graph containing two tasks. One task is "completed" while the other is "unsubmitted". Hence the overall status of the task graph os "active"

and restart them after migrating the suspended task graphs to a different host (see Figure 9). Thus, the checkpointable task graphs in our abstraction model support the notion of fault-tolerance and migratable execution. Checkpointing of task graphs is not analogous to checkpointing of tasks. In task checkpointing, the corresponding execution program is instrumented with special checkpoint-instructions that facilitate the rollback and restart of the executable from any checkpoint-instruction. Hence, the restart of any failed task will resume the execution from the last checkpoint-instruction that was successfully processed. In the task graph checkpointing scheme, however, rather than checkpointing the progress of a particular task, we checkpoint the progress of the task graph. Our checkpointing scheme does not support task checkpointing. We assume that this is provided

by the underlaying framework or by the application that is wrapped into a task. To keep our task graph checkpointing model simple, we maintain the checkpoint status in XML notation. Listing 7 shows the checkpointed XML notation for the task graph depicted in Figure 6.
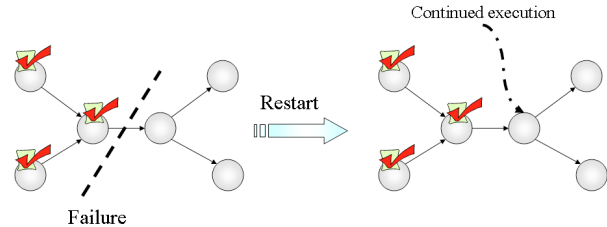

Figure 8: Checkpointable task graphs allow applications to resume failed execution flows from the point of failure.
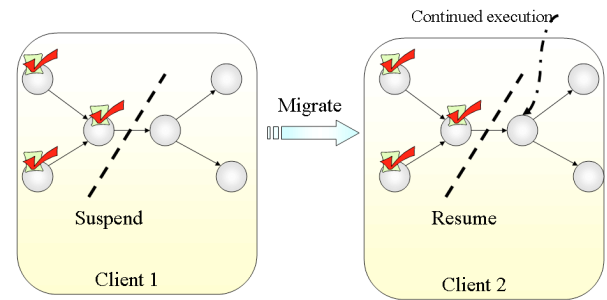

Figure 9: Checkpointable task graphs allow applications to suspend the execution of a task graph, migrate the suspended task graph to a different host, and resume the execution from the point of suspension.

2.3 Handler-Exectution Pattern

In our abstraction model, tasks and task graphs are static container entities that describe the semantics of the execution. The actual execution, however, is carried out by the handlers defined in our model. The handlers transform the abstract representation of tasks and task graphs into protocol-specific objects.

The task handler, as the name suggests, handles and processes abstract tasks. It is responsible for translating the specification, security context, and service contact into artifacts that are understood by the backend Grid implementation.

Figure 7: A sample checkpoint status with an "active" task graph. The task graph has two subtasks, one of them "completed" and the other "unsubmitted". Resubmission of this checkpoint status will execute only the unsubmitted task.

```
<taskGraph>
<identity> id:taskGraph </identity>
<name> Main Graph </name>

  <task>
    <identity> id:task1 </identity>
    <name> Task1 </name>
    <type> File Transfer </type>
    <provider> GT2 </provider>
    <specification>
      <fileTransferSpecification>
        <source> gsiftp://hot.anl.gov:2811//source.txt </source>
        <destination>
          gsiftp://cold.anl.gov:2811//dest.txt
        </destination>
        <directoryTransfer> false </directoryTransfer>
        <thirdParty> true </thirdParty>
      </fileTransferSpecification>
    </specification>
    <status> Completed </status>
    <submittedTime> 2004-06-23T16:27:29.606 </submittedTime>
    <completedTime> 2004-06-23T16:27:47.819 </completedTime>
  </task>

  <task>
    <identity> id:task2 </identity>
    <name> Task2 </name>
    <type> File Transfer </type>
    <provider> GT2 </provider>
    <specification>
      <fileTransferSpecification>
         <source>gsiftp://new.anl.gov:2811//source</source>
        <destination>
          gsiftp://old.anl.gov:2811//dest
        </destination>
        <directoryTransfer> true </directoryTransfer>
        <thirdParty> true </thirdParty>
      </fileTransferSpecification>
    </specification>
    <status> Unsubmitted </status>
  </task>

  <dependencyList>
    <dependency from="id:task1" to="id:task2"/>
  </dependencyList>
  <status> Active </status>
  <submittedTime> 2004-06-23T16:27:20.606 </submittedTime>
</taskGraph>
```
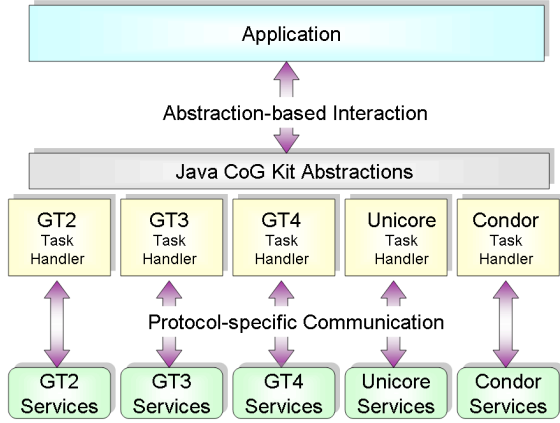
Figure 10: The task handler provides the mapping between abstract tasks and protocol-specific entities. We currently provider GT2 and GT3 handlers. Other handlers are either under construction or planned for future development.

Hence, for every protocol provider there must be a corresponding task handler providing the necessary abstract-to-protocol binding. The handler is the only component in our abstraction model that is backend implementation-specific, thereby minimizing the dependency on technology-specific functionality. Therefore, our framework can support and adapt to backend implementation as long as the appropriate handlers for it are available (see Figure 10).

A task graph handler, on the other hand, enforces the execution ordering on task graphs. It also manages the checkpoint status for the task graph and its successful resumption after execution failure or suspension. We note that the task graph handler is independent of the protocol provider. As shown in Figure 11, it simply delegates the execution of its nodes to the appropriate task handlers or task graph handlers in a recursive manner. Since the base elements of a task graph are tasks, this recursive execution of the task graph is terminated when all the base tasks are executed.

We refer to the pattern of executing arbitrary Grid tasks using our handler model as the *handler-execution pattern*. With the handler-execution pattern, applications create a task, assign a detailed specification for the task, associate a security context and service contact with the task, and designate a provider. In the case of a task graph, the application prepares multiple tasks and imposes execution
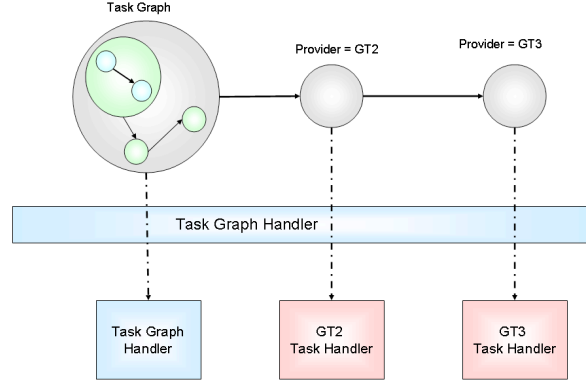

Figure 11: The task graph handler recursively delegates the execution of every node of the task graph to an appropriate task handler or another task graph handler.

dependencies on these tasks. The task or the task graph is then submitted to an appropriate handler that executes it on the Grid, allowing the application to monitor its execution status.

2.4 Resource-Execution Pattern

Tasks, task graphs, corresponding handlers, and the handler-execution pattern focus on abstracting units of execution in a Grid. Additionally, one can abstract Grid resources. We know that computational Grids aggregate several services dispersed across geographic boundaries [6]. Our resource abstractions allow applications to map, aggregate, and classify arbitrary Grid services into abstract execution resources. Such an abstract arrangement of services provides a mapping from the Grid service space to a user perspective of those services in the application space.

In the execution resource pattern, services of different types (job execution, file transfer, or information query) for various providers (GT2, GT3, Unicore, and Condor) are aggregated into a single execution resource (see Figure 12). The classification of the service aggregation is application- or user-defined. Every execution resource contains the following entities:

- *Security Context Mapping:* Every provider supported by the execution resource needs a corresponding security context. This security context is used by the execution resource to authenticate the task with the remote service.
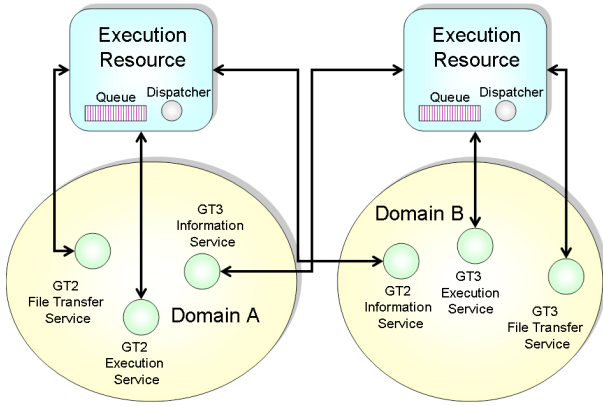
7

Figure 12: The execution resource aggregates geographically distributed Grid services into an abstract entity capable of task execution.

- *Service Contact Mapping:* The execution resource maps each provider and service type combination to a specific service contact capable of offering the required functionality following the provider protocol.

- *Resource Queue:* Every task submitted to the execution resource is appended to the resource queue, which manages the execution requests.

- *Dispatcher:* The resource queue is serviced by the resource dispatcher. The dispatcher extracts tasks from the queue and inspects the task type, protocol provider, and several other attributes such as the task priority and the estimated execution time. Based on these parameters, it associates an appropriate service contact and security context with the task and submits the task to an appropriate handler for execution. The dispatcher has a dispatching policy that decides the selection of the next task. This can be a classical policy such as first-in-first-out, shortest job first, or a user-defined customized heuristic policy [29] [30].

Such an execution pattern allows the application to partition available Grid services into logical or virtual resources and execute Grid tasks with abstract resources. We refer to the pattern of executing Grid tasks as the *resource-execution pattern*. Unlike the handler-execution pattern, the resource-execution pattern does not require the knowledge of service contacts on a per task basis. Applications classify their

Grid services and partition them into Grid resources, selecting a specific dispatching policy for each of these resources. The scheduler selects the tasks submitted to a resource and executes them with an appropriate service, internally employing the handler-execution pattern.
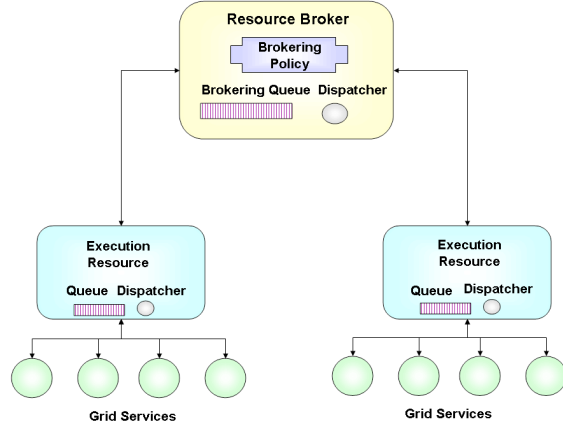

Figure 13: A resource broker aggregates multiple execution resources and performs the necessary resource mapping to execute the submitted tasks on the appropriate Grid resources.

2.5   Broker-Execution Pattern

The broker-execution pattern allows an application to aggregate a set of services into a single resource. The resource broker model, on the other hand, allows the application to aggregate several execution resources to represent a distributed Grid. The application can create multiple execution resources, thereby forming a working context of a computational Grid. The execution resources are then associated with a resource broker that functions as a matchmaking entity between submitted tasks and their appropriate execution services (see Figure 13). We refer to the pattern of submitting Grid tasks as the *broker-execution pattern*.

Every resource broker has the following associated entities:

- *Execution Resource Pool:* A set of execution resources capable of handling different service type and provider combinations.

- *Broker Queue:* An input queue for managing the task (task graph) execution requests submitted to the resource broker.

- *Dispatcher:* A means for servicing the broker queue. The dispatcher has an associated policy that selects the next brokered task from the brokering queue. Several policies can be supported, such as first-in-first-out, shortest job first, or random.

- *Brokering Policy:* The policy responsible for the matchmaking decisions between the dispatcher-selected tasks and the available set of execution resource. A number of different brokering policies can be formulated with the resource broker, including round-robin, least loaded, or random.

Tasks or task graphs, when submitted to the broker, are appended to the input queue. The dispatcher selects the next task to be brokered from the input queue based on its internal selection policy. For every selected task, the broker consults its policy to choose a resource from the pool of available execution resources, thereby internally adopting the resource-execution pattern.

Any of the three abstract execution patterns discussed thus far (handler-execution pattern, resource-execution pattern, and the broker-execution pattern) can be used by applications. However, the handler-execution pattern offers maximum control over task execution with reduced flexibility, whereas the broker-execution pattern allows maximum flexibility with the autonomy delegated to the resource broker rather than the application itself.

3 QoS Execution Extensions

To support our philosophy of an extensible Grid abstraction model, we integrate it with an advanced Grid quality-of-service (QoS) execution backend. Our objective is to showcase the fact that several Grid frameworks can be interfaced via our abstractions without modifications to existing applications that employ our abstraction model. First, we provide a brief overview of the Grid QoS Management project that provides the necessary backend for a possible prototype QoS handler.

3.1 Grid Quality-of-Service Management

Execution requests from Grid applications are serviced by remote execution services on a best-effort basis rather than with deterministic guarantees. Nevertheless, some applications need to obtain results for their tasks within strict deadlines. Hence, a nondeterministic scheduling approach without guarantees is often not suitable. For these applications, it is often necessary to reserve Grid resources and services at a particular time (in advance or on demand) to ensure deterministic bounds for execution latencies. Such QoS features are highly desirable, indeed required, if the Grid execution management service is to be able to handle complex scientific and business applications.

Grid Quality-of-Service Management (GQoSM) is a framework to support QoS management in computational Grids [31, 32]. GQoSM consists of three main operational phases: establishment, activity, and termination. During the establishment phase, a client application states the desired service and QoS requirements. GQoSM then undertakes a service discovery, based on the specified QoS properties, and negotiates an agreement for the client application. During the activity phase, additional operations such as QoS monitoring, adaptation, accounting, and possibly renegotiation may take place. During the termination phase, the QoS session is ended as a result of resource reservation expiration, agreement violation, or service completion; resources are then freed for use by other clients.
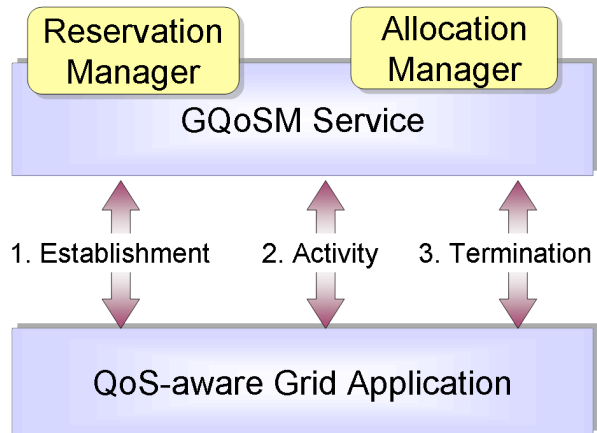

Figure 14: GQoSM service architecture.

As shown in Figure 14, one of the key component of the GQoSM framework is the GQoSM service. The GQoSM service provides QoS functionalities such as negotiation, reservation, and resource allocation with certain quality levels. Each QoS-enabled Grid resource is accessed through a GQoSM service. It publishes itself to a registry service, so clients and

QoS brokers can discover the existence of the different GQoSM services. The GQoSM service primarily performs two functions: resource reservation and resource allocation.

When a reservation request is received, the GQoSM service undertakes an admission control to check the feasibility of granting such a request. This feasibility check is undertaken by the reservation manager. If such a reservation is possible, the requested resources are reserved, the reservation table is updated, and an agreement consisting of the reservation specification is generated and returned to the client. On the other hand, when a resource allocation request is received, the GQoSM service verifies that the user has indeed made a reservation based on the supplied agreement. If this test is successful, then the GQoSM service submits the specification of the job to be executed, along with its reservation agreement to the appropriate execution manager, such as the Globus Resource Allocation Manager (GRAM) [33] on that particular Grid resource.

3.2   GQoSM Handler Integration

The GQoSM framework offers the requisite functionality for QoS-related features that allows the transformation of any arbitrary Grid resource into a QoS-aware Grid entity. However, to take advantage of such QoS-aware Grid resources, applications must be able to interact with such entities without significant changes in logic and implementation. Hence, we provide interactions with the GQoSM framework via our abstraction model, making it seamless for Grid applications to benefit from the GQoSM architecture. The only additional effort required is the "one-time" development of a GQoSM handler for our model.

As depicted in Figure 15, a possible implementation of the GQoSM handler is split into two modules: the QoS reservation module and the job execution module. When an abstract Grid task is delegated to the GQoSM handler, it invokes the reservation module to communicate with the peer reservation manager of the GQoSM service and obtain its reservation token. After the token is retrieved, the execution module of the handler communicates with the resource allocation manager of the corresponding GQoSM service and submits the Grid task based on the task parameters (start time and end time).

All Grid applications that currently use our abstraction model can attain QoS assurances without being aware of complex QoS discovery, QoS negotiation, and QoS execution phases. Applications sim-
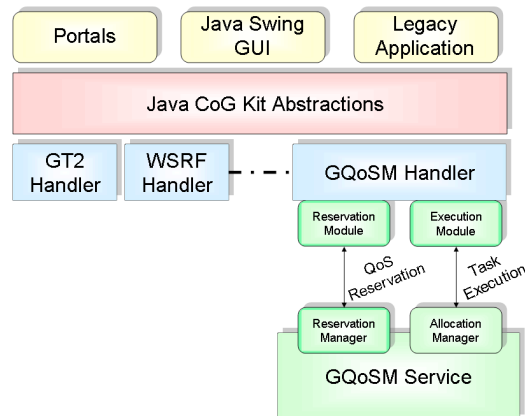

Figure 15: GQoSM handler integration.

ply formulate an abstract Grid task and assign it a provider as "GQoSM". Further, if the application requires asynchronous job startups, additional attributes such as task startup and end times can also be specified. Based on the degree of autonomy and flexibility, the application can leverage from several execution patterns described in the previous sections while interfacing with an advanced QoS architecture.

4   Conclusion

The Grid community under the auspices of the Global Grid Forum is moving in the right direction toward standardizing the Grid architecture. However, the Grid standardization is an incremental process subject to ongoing refactoring. Although such enhancements in the Grid architecture will ultimately prove beneficial for the entire Grid community, they impose an undue overhead on Grid applications to continuously keep abreast of these changes. Motivated by the need to provide an application development framework that shields the Grid users and application developers from the technological complexities of the Grid implementation, we present a suite of pattern-based Grid abstractions. Applications using our framework can concentrate on their objectives while remaining compatible with the latest Grid technologies.

The basic elements in our Grid abstractions are tasks and the task graphs. A Grid task is an atomic unit of work in our framework. Our model can support an execution flow ordered as a hierarchical directed acyclic graph, referred to as a task graph.

The task graph is a checkpointable entity facilitating fault-tolerance and mobility in execution flows. An alternative perspective on Grid abstractions is also presented focusing on the notion of execution resources. Abstract services can be aggregated into execution resources capable of servicing execution requests. Higher levels of execution abstractions are offered via the resource broker constructs that aggregate multiple execution resources providing advanced matchmaking functionality.

We have tested the utility and ease of use of our abstraction model by integrating it with the GQoSM execution framework with an older version of the Java CoG Kit. Although the GQoSM handler is not publicly distributed with the current Java CoG Kit v4 [34], it validates our hypothesis that advanced backend systems can be seamlessly integrated in our abstraction model without changes to the application.

A prototype implementation of the elementary Grid abstractions is available as a part of the Java CoG Kit v4. In the current distribution, we have implemented task handlers for Globus Toolkit v2.4, Globus Toolkit v3.02, Globus Toolkit v3.2.0, Globus Toolkit v3.2.1, and secure shell (SSH). Community members have also provided handlers for the Unicore framework. A port to Globus Toolkit 4 is under development. Furthermore, the Globus Toolkit v4 contains already several abstractions that have its origin in the Java CoG Kit.

Acknowledgments

References

[1] Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. *Grid Computing: Making the Global Infrastructure a Reality*, chapter The Physiology of the Grid, pages 217–249. Wiley, 2003. `http://www.globus.org/ogsa`.

[2] Web Services Resource Framework (WSRF). Web Page. `http://www.globus.org/wsrf`.

[3] M. Govindaraju, S. Krishnan, K. Chiu, A. Slominski, D. Gannon, and R. Bramley. XCAT 2.0 : A Component Based Programming Model for Grid Web Services. In *Grid 2002, 3rd International Workshop on Grid Computing*, 2002. `http://www.extreme.indiana.edu/xcat`.

[4] The Globus Project. Web Page. `http://www.globus.org`.

[5] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001. `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--cog-cpe-final.pdf`.

[6] Gregor von Laszewski and Kaizar Amin. *Grid Middleware*, chapter Middleware for Commnications, pages 109–130. Wiley, 2004. `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--grid-middleware.pdf`.

[7] Kaizar Amin, Mihael Hategan, Gregor von Laszewski, and Nestor J. Zaluzec. Abstracting the Grid. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pages 250–257, La Coruña, Spain, 11-13 February 2004. `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--abstracting.pdf`.

[8] The Legion Project. Web Page. `http://legion.virginia.edu`.

[9] Unicore. Web Page. `http://www.unicore.de/`.

[10] Douglas Thain, Todd Tannenbaum, and Miron Linvy. *Grid Computing: Making the Global Infrastructure a Reality*, chapter Condor and the Grid. Number ISBN:0-470-85319-0. John Wiley, 2003.

[11] Open Grid Services Architecture (OGSA). Web Page. `http://www.globus.org/ogsa`.

[12] Gregor von Laszewski, Mike Seablom, Milo Makivic, Peter Lyster, and Sanya Ranka. Design Issues for the Parallelization of an Optimal

Interpolation Algorithm. In G.-R. Hoffman and N. Kreitz, editors, *Coming of Age, Proceedings of the 4th Workshop on the Use of Parallel Processing in Atmospheric Science*, pages 290–302, Reading, UK, 21-25 November 1994. European Centre for Medium Weather Forecast, World Scientific. `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski94-4dda-design.pdf`.

[13] Gregor von Laszewski. An Interactive Parallel Programming Environment Applied in Atmospheric Science. In G.-R. Hoffman and N. Kreitz, editors, *Making Its Mark, Proceedings of the 6th Workshop on the Use of Parallel Processors in Meteorology*, pages 311–325, Reading, UK, 2-6 December 1996. European Centre for Medium Weather Forecast, World Scientific. `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--ecwmf-interactive.pdf`.

[14] Gregor von Laszewski. A Loosely Coupled Metacomputer: Cooperating Job Submissions Across Multiple Supercomputing Sites. *Concurrency, Experience, and Practice*, 11(5):933–948, December 1999. The initial version of this paper was available in 1996. `http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--CooperatingJobs.ps`.

[15] Rich Wolski, John Brevik, Graziano Obertelli, Neil Sprong, and Alan Su. Writing Programs that Run EveryWare on the Computational Grid. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1066–1080, October 2001.

[16] Hidemoto Nakada, Mitsuhisa Sato, and Satoshi Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems*, 15(5-6):649–658, 1999.

[17] Henri Casanova and Jack Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. *International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, 1997.

[18] G. Allen, W. Benger, T. Goodale, H.-C. Hege, G. Lanfermann, A. Merzky, T. Radke, E. Seidel, and J. Shalf. The Cactus Code: A Problem Solving Environment for the Grid. In *High-Performance Distributed Computing, 2000.*

Proceedings. The Ninth International Symposium on, pages 253 –260, Pitsburg, PA, August 2000. `http://xplore2.ieee.org/iel5/6975/18801/00868657.pdf?isNumber=18801`.

[19] J. Novotny. The Grid Portal Development Kit, 2001. `http://dast.nlanr.net/Projects/GridPortal/`.

[20] Albee Jhoney, Manu Kuchhal, and Venkatakrishnan. Grid Application Framework for Java (GAF4J). Technical report, IBM Software Labs, India, 2003. `https://secure.alphaworks.ibm.com/tech/GAF4J`.

[21] E. Seidel, G. Allen, A. Merzky, and J. Nabrzyski. Gridlab: A grid application toolkit and testbed. *Future Generation Computer Systems*, 18:1143–1153, 2002.

[22] IBM Grid Computing. Web Page. `http://www-1.ibm.com/grid/`.

[23] Gabrielle Allen, Kelly Davis, Konstantinos N. Dolkas, Nikolaos D. Doulamis, Tom Goodale, Thilo Kielmann, Andre Merzky, Jarek Nabrzyski, Juliusz Pukacki, Thomas Radke, Michael Russell, Ed Seidel, John Shalf, and Ian Taylor. Enabling applications on the grid: A gridlab overview. *International Journal of High Performance Computing Applications*, 17(04):449, November 2003.

[24] The Global Grid Forum Web Page. Web Page. `http://www.gridforum.org`.

[25] Gregor von Laszewski. Keynote: Cog kit abstractions. Workshop on Grid Application Programming Interfaces in conjunction with GGF12, Brussels, Belgium, 20 September 2004. (Keynote). `http://www.cs.vu.nl/ggf/apps-rg/meetings/ggf12.html`.

[26] Gregor von Laszewski. Java cog kit workflow abstractions. GGF Workshop Management Working Group, GGF11 - The Eleventh Global Grid Forum, Honolulu, Hawaii USA, 6-10 June 2004. (Presentation).

[27] L. Smarr and C.E. Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[28] D. Bhatia, V. Burzevski, M. Camuseva, G. C. Fox, W. Furmanski, and G. Premchandran. WebFlow - a visual programming paradigm for

Web/Java based coarse grain distributed computing. *Concurrency: Practice and Experience*, 9(6):555–577, 1997.

[29] Gregor von Laszewski, Ian Foster, Jarek Gawor, Warren Smith, and Steve Tuecke. CoG Kits: A Bridge between Commodity Distributed Computing and High-Performance Grids. In *ACM Java Grande 2000 Conference*, pages 97–106, San Francisco, CA, 3-5 June 2000. http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--cog-final.pdf.

[30] Gregor von Laszewski. Using the Globus Metacomputing Toolkit for Seamless Computing. Supercomputing Center at ECMWF, Reading, UK, December 1997. *(Invited Talk)*.

[31] Rashid Al-Ali, Kaizar Amin, Gregor von Laszewski, Omer Rana, and David Walker. An OGSA-based Quality of Service Framework. In *Proceedings of the Second International Workshop on Grid and Cooperative Computing (GCC2003)*, number LNCS 3003 in Lecture Notes on Computer Science, pages 529–540, Shanghai, China, 7-10 December 2003. Revised Papers, Part II. Springer Verlag. ISBN: 3-540-21993-5. http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--qos.pdf.

[32] Rashid Al-Ali, Kaizar Amin, Gregor von Laszewski, Mihael Hategan, Omer Rana, David Walker, and Nester Zaluzec. QoS Support for High-Performance Scientific Applications. In *Proceedings of the IEEE/ACM 4th International Symposium on Cluster Computing and the Grid (CCGrid 2004)*, Chicago IL, USA, 2004. IEEE Computer Society Press. http://www.mcs.anl.gov/~gregor/papers/vonLaszewski--qos-ccgrid04.pdf.

[33] Karl Czajkowski, Ian Foster, Nick Karonis, Cral Kesselman, Stewart Martin, Warren Smith, and Steve Tuecke. Resource Management Architecture for Metacomputing Systems. In *Proc. IPPS/SPDP:Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, March 30 - April 3 1998. http://www.isi.edu/~karlcz/papers/gram97.pdf.

[34] Gregor von Laszewski, Kaizar Amin, Matt Bone, Mike Hategan, Pankaj Sahasrabudhe, Mike Sosonkin nd Robert Winch, Nithya Vijayakumar, and David Angulo. The Next Generation of the Java CoG Kit (Version 4). Supercomputing 2004, Pittsburgh, 6-12 November 2004. (Refereed Poster) Best poster award. http://www.sc-conference.org/sc2004.