The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid

Gabrielle Allen[*†], Kelly Davis[*], Tom Goodale[*†], Andrei Hutanu[‡], Hartmut Kaiser[*], Thilo Kielmann[§], André Merzky[§], Rob van Nieuwpoort[§], Alexander Reinefeld[‡], Florian Schintke[‡], Thorsten Schütt[‡], Ed Seidel[*†], Brygg Ullmer[‡],
[*]Albert Einstein Institute, Golm, Germany [†]Louisiana State University, Baton Rouge, USA
[‡]Zuse Institute, Berlin, Germany [§]Vrije Universiteit, Amsterdam, The Netherlands

http://www.gridlab.org/

***Abstract*— Core grid technologies are rapidly maturing, but there remains a shortage of real grid applications. One important reason is the lack of a simple and high-level application programming toolkit, bridging the gap between existing grid middleware and application-level needs. The Grid Application Toolkit (GAT), as currently developed by the EC-funded project GridLab [1], provides this missing functionality. As seen from the application, the GAT provides a unified simple programming interface to the grid infrastructure, tailored to the needs of grid application programmers and users. A uniform programming interface will be needed for application developers to create a new generation of 'grid-aware' applications. The GAT implementation handles both the complexity and the variety of existing grid middleware services via so-called adaptors. Complementing existing grid middleware, GridLab also provides high-level services to implement the GAT functionality.**

We present the GridLab software architecture, consisting of the GAT, environment-specific adaptors, and GridLab services. We elaborate the concepts underlying the GAT and outline the corresponding API. We present the functionality of GridLab's high-level services and demonstrate how a dynamic grid application can easily benefit from the GAT. All GridLab software is open source and can be downloaded from the project web site.

***Index Terms*— Grid programming, grid computing, generic services, grid applications**

I. INTRODUCTION

THE advocates of grid computing promise a world where large, shared, scientific research instruments, experimental data, numerical simulations, analysis tools, research and development platforms, as well as people, are closely coordinated and integrated in 'virtual organizations'. Still, relatively few grid-enabled applications exist that exploit the full potential of grid environments. This must be largely attributed to the difficulties faced by program developers in trying to master the complex interplay of the various components like resource reservation, security, accounting, and communication. Moreover, grid middleware like Globus [2], Condor-G [3], and Unicore [4] are still undergoing many changes, with new software releases appearing frequently.

Dealing with complex and changing programming interfaces is one problem. Another is making applications grid aware. Unlike single, homogeneous parallel machines or clusters, grid environments are heterogeneous and dynamically changing. To run efficiently, grid applications need to be scheduled and then executed in such a manner that the performance of the resources which are actually used are properly taken into account. Submitting existing application codes, unmodified, to remote grid resources may lead to less than rewarding results. For example, an application that detects a diminishing communication bandwidth during runtime could trigger a load-redistribution tool or search for better-suited resources. Traditionally, application programmers would have to implement themselves such mechanisms into their code. This is not only tedious, but usually restricts the code to *one* specific grid middleware package. This runs contrary to the very nature of grids, which imply a heterogeneous environment in which applications must run. In order to be effective, a grid application must be able to run in any environment in which it finds itself. Ideally grid applications would discover required grid services at run time, and use them as needed, independent of the particular interfaces used by the application programmer.

Even with the right grid resources and middleware in place, a further step is needed for applications. Research must be concentrated on the development of high-level, application-oriented toolkits that free programmers from the burden of adjusting their software to different and changing grid resources, and middleware packages with their release history. Our *Grid Application Toolkit (GAT)* provides the missing link between the application level and the various grid middleware packages. The core idea of GAT is similar to that of the well established MPI [5] message passing standard, but at a much higher (grid) level. GAT has the following properties:

- ease of use
- support for different application programming languages
- support for different grid middleware, even concurrently
- mechanisms for the same application (source) code to run on a variety of systems ranging from laptops to HPC resources
- orientation towards dynamic and adaptive grid-aware applications

If grids are to become successful outside academia, it is important to lower the coding effort for grid application programmers. Currently, the coding effort for a simple program that only copies a file from A to B ranges from approximately 130 lines of code in the case of OGSA [6] to about 100 lines when using native Globus-GASS [2] interfaces, both shown in the Appendix. This large amount of code is needed for the many parameters that can be tuned for the specific setting in use. In many cases, however, the best parameter settings can be determined automatically so that the application does not need to bother. With the GAT library, the above file transfer example can be performed with just two calls and some lines of error checking and handling. The resulting code in Fig. 1 is rather self-explanatory. It maintains at least the same functionality as the OGSA and Globus approaches shown in the Appendix. More than that, the code can be used in many environments without change.

Of course, the grid middleware-dependent code (Globus, OGSA, Unicore, etc.) still exists in our solution, but it is hidden from the application programmer inside the GAT library. Adjustable parameters are automatically optimized, based on available information on the current environment. Overall, GAT provides an easy and high-level application programming interface for programming grid-aware applications.

In Section II we discuss typical, dynamic grid application scenarios and their required, grid-related functionality. Our Grid Application Toolkit (GAT) has been designed to provide such functionality. Its architecture is outlined in Section III, its application programmer interface (API) in Section IV. We discuss APIs of existing grid middleware in Section V. Section VI concludes.

Fig. 1. Code Example: *GAT API example using the* `C++` *language bindings*

```
#include <string>
#include <iostream>
#include <GAT++.hpp>

GAT::Result
RemoteFile::GetFile (GAT::Context context,
                     std::string source_url,
                     std::string target_url)
{
  try
  {
    GAT::File  file  (context, source_url);
     file.Copy       (target_url);
  }
  catch (GAT::Exception const &e)
  {
    std::cerr << "Some␣error:␣" << e.what()
      << std::endl;
    return e.Result();
  }
  return GAT_SUCCESS;
}
```

II. Dynamic Grid Application Scenarios

Design and implementation of the GAT have been driven by a number of important application use cases from the intended user community. Common to all these use cases are simple-looking scenarios which have rather complex implementations in terms of interactions between the user application and the grid services and resources. Such scenarios are:

- The user starts and application.
- The application notifies the user about status changes.
- The user controls the application.
- The application spawns a subtask.
- The user migrates a running application.
- The user requests visualization of result data.

In most of these scenarios the application itself becomes an active entity similar to web portals, interacting with grid services and resources. In the following, we will describe the first scenario in detail (*A*), providing insight into the complexity of the required operations. Two other scenarios (*B*, *C*) will be described only briefly to further outline the scope of the GAT. (*D*, *E*, *F*) describe important recuring elements of various scenarios in more detail, to emphasize the complexity of the underlying service activities.

A. Starting a user application

Usually, the application is initially submitted to some grid compute resource by specifying a job description and triggering its computation. This potentially involves grid services responsible for resource discovery, resource management, authentication, authorization, process management, and data management. Data output files created by the application may get names in a global name space, in addition to the locally valid physical file names. The individual steps taken are illustrated in Fig. 2, which is based on the GridLab software suite.

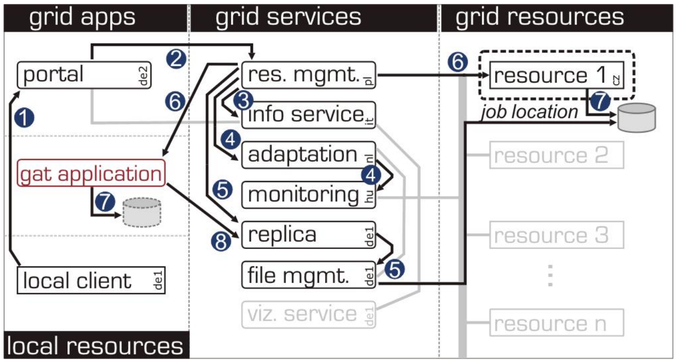


Fig. 2. Starting a user application.

1) A user contacts and logs into a web portal, as his interface to the grid environment.
2) The user requests an application startup via the portal, and specifies the name and location of executable, parameters, environment, resource requirements and input data files to a resource manager (e.g., the GridLab Resource Management System (GRMS)).
3) The resource manager uses some resource discovery mechanism or service (e.g. iGrid in GridLab) to discover suitable resources for execution of the application.
4) The resource manager selects the 'best' resource, with the help of an adaptation grid component (e.g., Delphoi in GridLab).

5) The resource manager uses the replica service to transfer the executable and input data files to the target resource.
6) The resource manager prepares the application environment and starts the executable with the specified set of parameters.
7) The application, now running, starts producing data, and creates physical data files on the host system.
8) The application creates entries in the global file name space of the replica system for its data files.

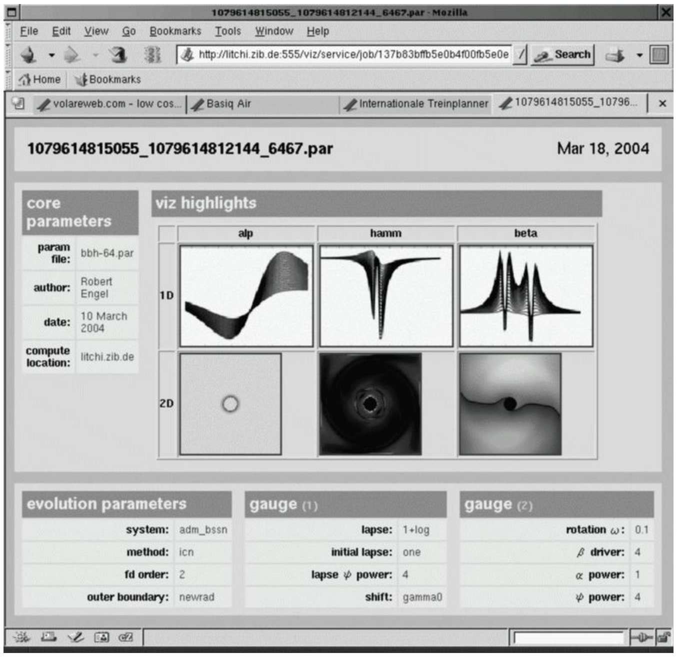
B. Migrating a running application

In this scenario, a running application should be migrated from one resource to another, and continue its computation without any loss of information. The migration of applications requires the re-creation of the application status and data on the remote side. Given the heterogeneity of grid environments, hardware architectures, operating systems, and middleware frameworks, the most feasible way of migration in a grid is application-level checkpointing [7]. Here, the application itself contains some extra code to write the complete set of status information to files, and to restart from these files upon resumption of execution.

At a certain point in time, some entity (e.g., a user via the portal) decides to trigger the migration process. In turn, an application checkpoint is requested. The application writes its checkpoint files and terminates. Information about the created checkpoint files needs to be known afterwards for resuming the computation. Besides the writing of checkpoint files, migrating an application is similar to starting a new user application, while the resource manager has to find *another* computing resource.

C. Visualization of result data

Grid applications can produce large amounts of result data files on the resources on which they are running. Visualizing these results can be performed as follows, using GridLab's visualization service: A running application registers its output directory with the replica catalog service, providing a mapping from logical file names to physical output files. Once the simulation finishes, the web portal (monitoring the application status) invokes the visualization service. The visualization service queries the replica catalog, and retrieves the physical location of the output directory from the user's logical home directory. It then generates images and an HTML page for data visualization. The result (e.g., Fig. 3) is published via the portal's job output page.


1079614815055_1079614812144_6467.par - Mozilla
File
Edit
View
Go
Bookmarks
Tools
Window
Help
http://litchi.zib.de:555/viz/service/job/137b83bffb5e0b4f00fb5e0e
Search
Home
Bookmarks
volareweb.com - low cos...
Basiq Air
Internationale Treinplanner
1079614815055_10796...
viz highlights
alp
hamm
beta
1D
2D

1079614815055_1079614812144_6467.par Mar 18, 2004

core parameters

param file:	bbh-64.par
author:	Robert Engel
date:	10 March 2004
compute location:	litchi.zib.de

evolution parameters		gauge (1)		gauge (2)	
system:	adm_bssn	lapse:	1+log	rotation ω:	0.1
method:	icn	initial lapse:	one	β driver:	4
fd order:	2	lapse ψ power:	4	α power:	1
outer boundary:	newrad	shift:	gamma0	ψ power:	4

Fig. 3. The GridLab visualization service presents result data via a portal web page.

D. Selecting the 'best' resource

An important, recurring issue in grid application scenarios is the selection of the most suitable resource to execute a submitted job. The resource manager (in GridLab, GRMS) bases its decisions on given metrics that may become rather complex, depending on multiple parameters. The GRMS needs to collect the values for these parameters from various sources. The GRMS may consult additional services to gather the parameter values and to perform the actual decision. Among these external services may be

- a grid information system (e.g., GridLab's iGrid) for a list of available resources,
- a grid monitoring system (e.g., GridLab's Mercury) for resource status information like system load, disk space, performance, queue wait time, etc,
- a decision making service (e.g., GridLab's Delphoi) to select the 'best' resource according to some metric. The service may base its decision on the delivered resource parameters, on additionally gathered data, on historic data, on knowledge base data, or on artificial intelligence algorithms.
- a grid authorization service (e.g., GridLab's GAS) to determine whether users are authorized to use the selected resource.

Using external services, the resource management system stays independent of application- and environment-specific configurations and metrics.

E. Security

Another important, recurring issue is security, including both authentication and authorization. Currently, the Grid Security Infrastructure (GSI) [8] is the most commonly used mechanism for ensuring authentication. All communication channels are usually GSI-secured (e.g., by using gSOAP [9], [10]), and GSI credentials with flexible lifetime and validity allow services to act on the users behalf.

Unfortunately, authorization is not handled by a general grid framework thus far. Usually, resources or services authorize users following their own, locally implemented policies, mostly relying on a grid-mapfile, listing all GSI credential subjects which are allowed to use the resource or service in question.

In the long run, Grid Authorization Services (such as GridLab's GAS) are supposed to handle the management of security policies. The GAS is then contacted during each service or resource invocation, and authorizes or denies the operation depending on the installed policies and the user's security credential. Within a grid API like the GAT, the provision of user credentials to services has to be taken into account, too.

F. Resource virtualization

One of the most powerful paradigms of Grids is *virtualization*, abstracting resources like computational entities and data files from the physical entities. The virtualization of resources in grids hides the actual physical resource behind a well defined interface. A resource management system may virtualize physical compute resources, and a file management system may abstract physical file resources. Virtualization of physical file locations is accomplished through replica management systems, which are widely used in grids [11]. A replica system provides a global name space for files, and the ability to map entries in that name space to a physical location, possibly one of many. This frees the application and service programmer from the need to track remote physical locations of files across a grid. Grid APIs like the GAT should thus support virtualized interfaces to grid resources.

III. The GAT Architecture

The diversity of deployed technology is simultaneously one of the major strengths and challenges of the grid. The range of available grid services is wide and constantly growing. Although the Global Grid Forum (GGF) aims at a global standardization for these services, these efforts will take time; will not cover all grid aspects; will not necessarily simplify the use of grid middleware (at the application level); and will not cover all grid middleware systems such as research projects, proprietary systems etc.

Also, grid environments are dynamically changing environments – that is, resources and services may dynamically join or leave the grid. Various versions of services may co-exist in a single grid, and various services providing similar capabilities may be available.

The GAT is designed to handle this diversity of grid middleware. In its current design, the GAT is split in two parts: the GAT engine and the GAT adaptors. The API exposed to the application is, as far as possible, independent of the grid middleware service used. All GAT applications link against the GAT engine, which provides proxy calls for all GAT API calls. The GAT API is designed to be simple and stable, and to provide the application with calls for essential Grid operations.

The GAT adaptors are lightweight, modular software elements which *provide access* to these specific capabilities. Adaptors are used to bind the GAT engine to the actual middleware service providing the capabilities. The interfaces between the GAT engine and the GAT adaptors mirror the GAT API. When called via the GAT API, the GAT engine dynamically selects from the currently available adaptors implementing the specific capabilities, and forwards the API request.

The GAT software architecture is layered, allowing a loose coupling of various software components. In particular, every application written using the GAT consists of four software layers (see Fig. 4):

- *The application layer*
  This layer contains all of the application-specific code that uses functions from the GAT API.
- *The GAT layer*
  This layer is represented by the GAT engine. It provides the GAT API for the application layer, and translates all API calls into calls to the adaptor bound to the corresponding API function. The adaptors glue the GAT API calls to the actual functionality provided by the grid middleware service.
- *The service layer*
  This layer represents the capabilities provided by the grid environment upon which the application actually runs, such as implemented by the GridLab project, or provided by middleware like Globus.
- *The core layer*
  This layer represents the resources available in the grid, such as operating system services or infrastructural components, compute resources, and data sources.

Both the application layer and the GAT layer execute in the *user space* of the application. The GAT adaptors form the interface from the user space to the *capability space*, consisting of the service layer and the core layer. The service and core layers expose the capabilities provided by the available resources.

A. GAT Engine

The GAT engine is a runtime library exposing the GAT API to the application. It represents a unified interface to the grid, and abstracts the application from the ever-changing grid infrastructure. The GAT engine consists of three logical parts.

- *The objects providing the GAT API functions:*
  This part provides the glue that maps the API function calls executed by an application to the corresponding adaptor-provided functionality. It consists of a very thin abstraction layer for each of the GAT API functions which selects the right adaptor, and dispatches the function call to this adaptor.
- *The adaptor management subsystem:*
  This part is responsible for loading available adaptors, managing their lifetime, and maintaining a capability registry that allows the GAT API subsystem to select the right adaptor. Each loaded adaptor registers its capabilities (groups of related functionality) with the GAT engine. The capability registry stores mappings between the adaptors and their provided capabilities. Every capability has a set of meta data attached (so-called "preferences"), allowing further control within the adaptor selection.
- *The utility objects and functions:*
  This part primarily consists of a set of utility functions for data handling (e.g., for lists and tables), as well as

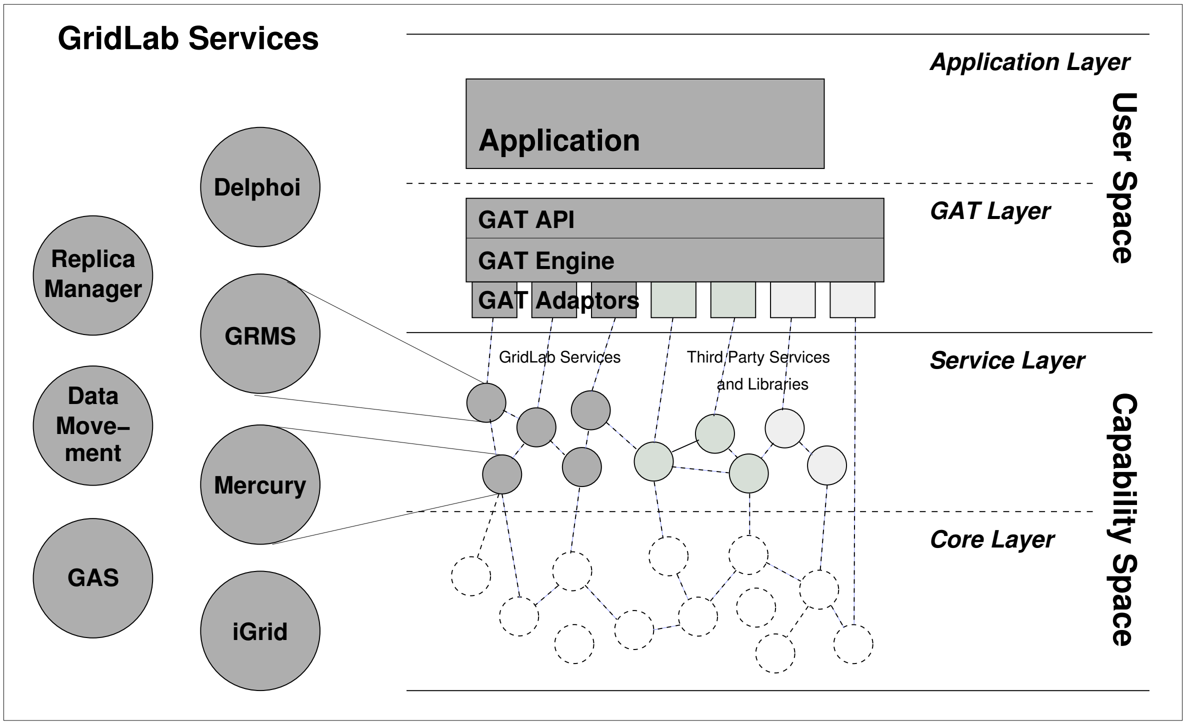


Fig. 4. The GAT framework software architecture.

error handling and reporting. These are not necessary in all implementation languages.

The GAT engine exposes two different sets of APIs: the API exposed to the application, and the API exposed for connection to the adaptors. This approach supports decoupling the application from the underlying grid middleware, and connecting it with any middleware service supporting the appropriate functionality. The GAT engine is supposed to be an extremely thin layer, providing an efficient way to switch between different implementations of the capabilities provided by the GAT API. All capability logic and all grid service interactions are implemented in the GAT adaptors, which is described in the following section.

B. Adaptors

As described above, GAT adaptors are lightweight modular software elements which provide access to specific capabilities. The capabilities are defined by the GAT API definition. The engine chooses adaptors on demand in order to satisfy the capabilities required by the application.

The interface between the GAT engine and adaptors is called the *Capability Provider Interface* (CPI). This interface mirrors the GAT API itself. Upon calls to the GAT API, the GAT engine dynamically selects from the currently available adaptors which provide access to the corresponding capabilities, and forwards the API request to the chosen adaptor.

An adaptor is compiled against the GAT engine and linked as a shared library (on platforms which support this). On loading, the adaptor's initialization function is called, in which the adaptor registers the capabilities it provides access to with GAT engine. This includes specification of the implemented interface (the GAT object); the functions that implement each method in this interface; and a set of properties for that adaptor, such as supported security models or resource identifier schemes such as URLs. These properties are used by the GAT engine to choose between adaptors providing the same capabilities.

The actual implementation of the individual methods depends on the kind of infrastructure to which the adaptor must provide access. For example, the methods in the `GATFile` reference adaptor are implemented using SOAP calls to two GridLab data management web services: *File Movement* and the *File Browsing*. The File Movement service provides synchronous and asynchronous third-party file transfer using the GASS libraries of the Globus Toolkit. The File Browsing service provides general file information as needed by the `GATFile` class. Example information includes file modification time, file size, and readable/writable flags.

As an adaptor usually binds to *one* capability provider (like a grid service or library) at a time, the implementation of adaptors is rather simple and straightforward. In fact, this implementation does not significantly differ from the code an application programmer would have used if coding directly against the grid environment. However, given that adaptors may often be reused across multiple applications, there is additional motivation for programmers to carefully implement fail safety, clean error messaging, good parameter defaults, data caching, performance tuning, etc. Hence, an adaptor may abstract rather complex grid interactions, for the benefit of the end user.

C. Integrating Existing Middleware Services

The purpose of the GAT is to decouple the application from the available grid middleware and its services. The GAT engine is supposed to work in various environments, ranging from an offline laptop to an international grid. For each framework, a set of adaptors is needed to provide the respective functionality. For offline machines (like developer laptops), resources from the *core layer* can be accessed directly. Adaptors to *core* capabilities can thus provide access to the machine itself on which the application is running. Such adaptors provide useful fallback solutions for local execution of a GAT application.

For granting access to resources within a grid environment, adaptors need to bind to *services*, like the ones provided by standard grid middleware such as Globus or Unicore.

For dynamic application scenarios like the one presented in Section II, the GridLab project has developed a set of high-level services and corresponding adaptors, providing advanced functionality for the GAT. Using these high-level services demonstrates the potential of writing GAT applications. However, it should be noted that the GAT is also fully functional without GridLab's services, as long as *any* complete set of adaptors is available. The GridLab services, as used in this manuscript, are the following.

GRMS
: The *GridLab Resource Management System* [12], [13] is a resource broker, implementing multicriteria job scheduling mechanisms adhering to user needs, VO policies, and resource-local policies. The GAT can use GRMS to start or migrate applications on a grid.

GAS
: The *Grid Authorization Service* [14] manages authorization policies in a grid. The GAT can use GAS indirectly via GRMS to determine which grid resources are accessible to a user who wants to run an application.

iGrid
: The *GridLab Information Services* (iGrid) [15], [16] extend the MDS information system by adding information about services, available software, users, firewalls, and recognized certificate authorities. The GAT can use iGrid for resource discovery purposes.

Replica Manager
: The *Replica Manager Service* [17] maintains a replica catalogue, mapping logical files to physical copies. The GAT can use the replica manager to access, locate, select and move replicated files.

Data Movement/Access
: The *Data Movement Service* [18] implements file transport using protocols such as *GridFTP* or *scp*. The GAT can use the *Data Access Service* [19], [20] to support remote file access.

Delphoi
: The *Delphoi* service [21] provides monitoring data and performance predictions for grid resources like network bandwidth, CPU utilization, or job waiting time in queueing systems. The GAT uses Delphoi indirectly, for guiding scheduling decisions in GRMS or for optimizing replica selection.

Mercury
: The *Mercury* monitoring service [22] monitors the status of compute resources and running jobs. The GAT can use Mercury either indirectly via Delphoi or directly for signaling events like triggering checkpointing.

IV. The GAT Application Programming Interface

The GAT application programming interface is divided into several subsystems which handle different aspects of Grids. These aspects include data management and access, resource management, monitoring and event handling, and information management. These subsystems are accompanied by a utility subsystem, providing convenience functionality; and the base subsystem, providing an interface to the GAT engine itself.

The specification of the GAT API is object oriented [23]. Hence, we describe the API in terms of objects and methods to these objects. However, the current implementations is in `C`, and proves that bindings to non-OO languages are possible (as has been our intention from the beginning). A `C++` wrapper to the `C` implementation is largely finished; wrappers to `Fortran`, `Perl`, and `Python` are planned for the near future. An independent native implementation within Java is within early stages of development.

The API description includes a number of small code examples. These are for brevity in C++, and have been chosen to illustrate the usage of certain objects or some subsystem, and are not necessarily reflecting real scenarios.

A. GAT Base Subsystem

The base subsystem of the GAT API defines general objects and methods which support interaction with the GAT engine. This subsystem also provides the base object (`GATObject`) which is inherited by all other objects of the API.

1) The `GATObject` class: The `GATObject` class is the ancestor of all (non-utility) classes in the GAT API. It is used to provide common functionality needed by all objects of the API, and helps to achieve a common look-and-feel throughout the API. Methods of the `GATObject` class are limited to administrative functions such as comparison, cloning, destruction and retrieving of any GAT object instance.

In particular, the `GATObject` class provides the following methods:

- `Equals`
  test two `GATObject` instances for equality.
- `Destroy`
  destroy a `GATObject` instance, and frees memory and associated resources allocated by this instance.
- `GetType`
  return the type of the `GATObject` instance.
- `Clone`
  create a copy of the given `GATObject` instance.

2) The `GATSelf` class: The `GATSelf` class represents the current process, and hence the GAT application itself. This class is singleton, meaning that there exists only one instance of this class per application instance. This instance is obtained by the `GATSelf.GetInstance` method.

`GATSelf` supports the handling of an application as a `GATJob` (see the Resource Management subsystem, sec. IV-B), and can be used to change various properties of this process. Examples include whether it is checkpointable or not, and what metrics or events it can report (see Event Management subsystem, see sec. IV-D.1). The `GATJob` instance which is made available through `GATSelf` can also be advertised (see Information Management subsystem), as well as used to access the corresponding job properties.

Operations:

- `GetInstance`
  a class level operation, which allows to access the single existing `GATSelf` object instance.
- `GetJob`
  gets the `GATJob` instance which is associated with this process. The returned `GATJob` can be advertised to other jobs.
- `AddRequestListener`
  adds a listener for a specific `GATRequest`. See the Event Management Subsystem description below for more details.

Fig. 5. Code Example: *Usage of `GATSelf`*

```
GAT::Self self = GAT::Self_GetInstance ();
GAT::Job  me   = self.getJob ();

me.stop ();
```

3) The `GATContext` class: The `GATContext` class represents the state and security context of an application. It is used to encapsulate a number of GAT API method calls into a common scope, including adaptor loading preferences, security settings and status code management.

Operations:

- `AddPreferences, GetPreferences, RemovePreferences`
  are used to manage the default adaptor loading preferences for `GATObjects` created within this `GATContext`. These preferences are used by the engine, whenever no explicit preferences are specified.
- `AddSecurityContext, RemoveSecurityContext, GetSecurityContexts, GetSecurityContextsByType`
  are used to manage the security settings used for operations executed within this `GATContext`.
- `GetCurrentStatus, SetCurrentStatus`
  are used to access the current status of the `GATContext`. This status is updated in all GAT API calls executed inside this `GATContext`.

4) The `GATStatus` class: The `GATStatus` class represents an error, trace, audit, or information message from a GAT operation or an underlying adaptor. This class is used to provide audit trails, allowing the user to trace (and the developer to debug) the sequence of events which happened in any particular GAT operation. Since the GAT engine and adaptors may perform several independent operations, each of which may have associated errors or status messages, a `GATStatus` instance forms a node in a tree of `GATStatus` instances, reflecting the complete execution path of any API call.

Operations:

The essential `GATStatus` operations are for retrieving messages and status codes, and for navigation in the call tree. Additionally, several additional methods are used by the GAT engine to construct and initialize new `GATStatus` object instances.

- `GetMessages`
  returns a list of messages (strings) associated with this `GATStatus` instance.
- `GetStatusCode`
  gets the status code of this `GATStatus` instance.
- `GetChildren`
  gets the child `GATStatus` instances of this instance.
- `GetParent`
  returns the parent `GATStatus` of this instance.

The typical use cases for this object is as follows:

- The application invokes `GATContext.GetStatus` to get the `GATStatus` object, created during the last GAT operation.
- The application gets the messages associated with this `GATStatus` object instance (these are strings) by invoking `GATSTatus.GetMessages`, and makes them available to the user.
- The application receives the numerical status code associated with this `GATStatus` object instance by invoking `GATStatus.GetStatusCode`, and similarly makes this information available.
- The application invokes `GATStatus.GetChildren` and, for each child, repeats the above procedure.

B. Resource Management Subsystem

The resource management subsystem deals with resources provided by the grid. It provides functions for both resource discovery and job management. The complete scheme for job descriptions within GAT is designed to support mapping into the job description languages typically used on the grid; e.g., those defined by Globus (RSL) or Condor (ClassAds), or those currently being defined by the JSDL [24] group within GGF.

From the GAT point of view, job submission is a simple four step process:

1) create a description of the software to be submitted;
2) create a description of the hardware requirements;

3) create a description of the software requirements; and
4) submit the job to a resource broker.

In the first step, basic details about the impending job are provided in a `GATSoftwareDescription`. Examples include the location of input and output files; the handling of stdin, stdout and stderr; command line arguments; and environment variables.

In the second step, the `GATHardwareResourceDescription` contains a set of requirements which must be met by the hardware on which the job will run. Typical parameters are the number of CPUs and the required amount of memory or disk space.

There are two common methods for dealing with executables in an heterogenous grid environment. The first involves compiling the application before submission and moving the executable to the remote site. The second involves submitting a shell script to the remote machine, which downloads the source code and builds the application. In both cases, a set of libraries must be installed on the machine. For the latter case, a compiler must also be installed to build the application. Such constraints are described in the third step: the `GATSoftwareResourceDescription`.

The final step is to forward all of these data and constraints to the resource broker, which selects a matching site and submits the job. The result of this step is a `GATJob` object, which allows for tracing and/or changing the job's status, shown in Fig. 6.

Normally, jobs are submitted via portals or other tools. While job submission functions are well known to users of other batch computing environments, programmers of these environments may not be familiar with functions for resource discovery. However, as described in Section II, there are several scenarios where these functionalities are essential.

Since the API for the Resource Management subsystem is somewhat more extensive than for the other subsystems, we will not list it here completely. Instead, the core objects with the most relevant methods are listed here. The remainder of the subsystem similarly reflects the general GAT design goals of simplicity and ease of use.

1) The `GATJob` class: The `GATJob` represents a grid application, and allows to check its status, and to interact with that application. For example, a spawned subtask can be requested to checkpoint and terminate.

The `GATJob` class provides the following methods:

- `GetJobDescription`
  returns the description of the job, similar to the description used to submit the job.
- `GetState`
  returns the current status of the job, such as `RUNNING`, `SCHEDULED` etc.
- `GetJobID`
  returns a globally unique identifier for the job, which can be used to refer to the job in another GAT application.
- `Checkpoint`
  requests the job to perform a checkpoint.
- `Clone`
  requests a complete clone of the job - after success, the job is running twice.
- `Migrate`
  requests the migration of the job, shown in Fig. 7.
- `Stop`
  requests termination of the job.

Fig. 6. Code Example: *Spawn a subtask*

```
GAT::Table sdt; sdt.add (”location”, ”/bin/date”);
GAT::Table hdt; hdt.add (”machine.type”, ”i686”);

GAT::SoftwareDescription         sd  (sdt);
GAT::HardwareResourceDescription hrd (hdt);

GAT::JobDescription jd (context, sd, hrd);
GAT::ResourceBroker rb (context, prefs);

GAT::Job j = rb.SubmitJob (jd);
```

2) The `GATResourceBroker` class: The `GATResourceBroker` is the most central object in this subsystem, and is responsible for all interactions with resources. It allows to search for resources, to reserve them for job submission, and to submit jobs to it.

The `GATResourceBroker` class provides following methods:

- `FindResource`
  returns a list of `GATResources` matching the given criteria.
- `ReserveResource`
  reserves a resource matching given criteria for later job submission, if possible.
- `SubmitJob`
  submits a job to a specific resource, or to some resource matching the given criteria, and returns a `GATJob` object.

Fig. 7. Code Example: *Migrate the spawned subtask `j` from Fig. 6*

```
hdt.set (”machine.name”, ”fs0.das2.cs.vu.nl”);

list<GAT::Resource> resources
                       = rb.findResources (hrd);

j.migrate (resources[0]);
```

3) The `GATResource` class: The `GATResource` class represents a grid resource, as for example returned by a resource broker after a search for free resources. The class can be used to get reservations on that resource, and can be given to a `GATResourceBroker` to submit a job to it.

The `GATResource` class provides following methods:

- `GetResourceDescription`
  returns the description of the resource, e.g. number of CPUs, available memory and disk space, installed software etc.
- `GetReservation`
  performs a reservation for later job submission, if possible.

4) The `GATReservation` class: This class is a simple placeholder for a reservation made on some specific resource. The `GATReservation` class provides following methods:

- `GetResource`
  returns the `GATResource` object representing the reserved resource.

C. Data Management Subsystem

The data management subsystem is not as extensive as the resource management subsystems, but covers a wide range of capabilities. Hence, we will describe the capabilities and their usage in some detail, but abstain from a detailed description of the API calls. The data management subsystem covers three areas: interprocess communication, remote file access, and file management.

1) Interprocess Communication: The `GATPipe` is the basic abstraction for interprocess communication. It represents a bidirectional communication channel between two processes.

In the TCP/IP world, the `GATPipe` corresponds to a connection. Like a TCP connection, the `GATPipe` connects two endpoints (`GATEndpoint`s), and is always bidirectional. Endpoints support two functions: `connect` and `listen`. The former is used to connect to a remote process, while the latter is used to wait for incoming connections. Both operations return a `GATPipe`, which can then be used for communication.

Like most objects in GAT, `GATEndpoint`s may be "advertised" (see Section IV-E). Thus, we can publish them and search for suitable endpoints in the advertisement databases. This property significantly reduces the complexity of connection establishment.

Our distinction between endpoints and connections differs from that of BSD sockets. For BSD sockets, the same underlying abstraction (sockets) is used for both connection establishment (connect and accept) and communication (read and write) operations.

To further simplify usage on the application side, it is planned to eventually extend `GATPipe` with support for typed read and write operations, as also known from MPI. However, the IPC capabilities of the GAT are primarily intended to provide connection-oriented communication between application processes. In grids, this already is a challenging problem in the presence of firewalls and not directly addressable nodes [25].

2) Remote File Access: There are three basic types of remote file access:

1) Moving the file to local storage and performing local file access;
2) Communicating with a remote site using a file format independent protocol; and
3) Communicating with a remote site using a file format dependent protocol.

The first method is the simplest approach, and sufficient for a wide range of applications. Its efficiency depends on the ratio between file size and the amount of data which is actually read from the file. For some applications, it might be sufficient to read just a few kilobytes of a multi-gigabyte file. In this case, the overhead is large and it would be much more efficient to only transfer the required parts of the file. This is the motivation for the latter methods.

The second method is in widespread use as well, although it is sometimes hidden from the user (e.g., in the case of NFS). GridFTP supports an extension implementing read, write and seek operations for remote files. The data is transferred via the GridFTP protocol. This approach works well for low-latency networks, but in wide area networks, the protocol overhead might dominate the execution time. It is best suited for a few medium-sized or large read requests.

As part of the GridLab project, we have developed a remote file access component [26] which can extract regular subsets from remote files very efficiently.

File-dependent protocols are relatively infrequently used, as the requirement for installing a different service or plugin for each file format can generate a large administrative overhead. Nonetheless, such systems can be very efficient, as they can be optimized for special file access patterns [19], [26].

The GAT supports the first two methods for remote file access because they apply to almost all scenarios and are supported on most machines. GAT provides two classes dealing with files. The first, `GATFile`, provides functions which work on files as a whole, such as query file size or moving the file in the grid. The second, `GATFileStream`, provides access to the contents of the file using read, write, and seek. An example is shown in Fig. 8.

The `GATFile` class includes the following methods (among others), which are self-explanatory :

- `Copy`
- `Move`
- `Delete`
- `IsReadable`
- `IsWritable`
- `GetLength`

The `GATFileStream` class includes the following methods (among others):

- `Read`
- `Write`
- `Seek`
- `Open`
- `Flush`
- `Close`

Fig. 8. Code Example: *Read a remote physical file*

```
char data[25];

GAT::File       file   (context, source_url);
GAT::FileStream stream (file);

stream.open  (RD_ONLY);
stream.seek  (100, SEEK_SET);
stream.read  (data, sizeof(data));
stream.close ();
```

3) Replica Management: A frequent grid application scenario deals with input files. After running several grid jobs reading such a file, multiple copies of the file are distributed across various locations of the grid. Even for a single user, it can be a difficult task to track the locations of these multiple copies.

Replica management systems provide mechanisms for managing distributed files and their copies. Many systems distinguish between physical and logical file names, as described earlier in Section II. In replica systems, logical file names provide a handle for a group of replicas of one file.

For the above example, one might create a logical file name identifying this set of results, e.g. `experiment-data-001`. Several physical file names would be associated with this logical file name, describing the locations where copies of the data is stored.

While the advantages for single users is clear, there is special benefit for the use of such systems by large scientific communities. A user can now browse or search a common catalog where copies of required data are stored, and pick the copy which provides the best access performance.

The Storage Resource Broker (SRB) [27] is an example of such a system. It provides interfaces for browsing the catalogs, as well as moving replicas between the sites.

The GAT provides access to such systems with the `GATLogicalFile` object. It provides methods for browsing catalogs, creating new replicas at user-specified sites, and accessing the individual replicas, shown in Fig. 9.

The `GATLogicalFile` class provides the following methods (among others):

- `AddFile`
- `RemoveFile`
- `Replicate`
- `GetFiles`

Fig. 9. Code Example: *Read a logical file*

```
char data[25];

GAT::LogicalFile  logical_file (context, name);
list<GAT::File>   files = logical_file.getFiles ();

GAT::FileStream stream (files[0]);

stream.open   (RD_ONLY);
stream.seek   (100, SEEK_SET);
stream.read   (data, sizeof(data));
stream.close  ();
```

D. Event and Monitoring Subsystem

The GAT event and monitoring subsystem allows the application to send and receive events, such as events generated by (e.g.) a grid monitoring service (e.g., Mercury). The programming model for this subsystem is based on subscriptions and callbacks. The application can subscribe to events of a certain type (events with a certain 'metric'), and register callbacks to handle incoming events of this type. For example, an application can assert its ability to respond to a checkpoint event, and register a callback for performing the checkpoint upon receiving an incoming requests. The application can also create its own events with self defined metrics, and insert them into the grid monitoring system. This allows external entities to monitor the application, and to obtain (e.g.) performance and simulation progress information.

The following paragraphs describe two example scenarios in GAT terms. The first example illustrates subscription to an event with a specific metric, and the retrieval of a corresponding event. The second example considers the creation of a custom event.

1) Event Retrieval: To subscribe to specific events, the application creates a `GATRequestListener` object. This object is then passed to `GATSelf.AddRequestListener`, parameterized as a *command* request listener with the name *checkpoint*. This information defines the event metric.

At appropriate points in the flow of control, the application invokes `GATContext.ServiceActions` (in a single-threaded implementation). This method iterates over all requests which have been received in the meantime, and invokes the `ProcessRequest` operation on the `GATRequestListener`, thus passing information about the received event. The application can later asynchronously `respond` to the event; e.g., to signal success or failure in processing the event. An example is shown in Fig. 10.

Fig. 10. Code Example: *React to a checkpoint event*

```
//MyClass : public GAT::RequestListener;

  MyClass::someRoutine ()
  {
    GAT::Self self = GAT::Self_GetInstance ();
    self.addRequestListener (GAT_COMMAND,
                             ”checkpoint”)

    while ( TRUE )
    {
      work ();
      context.serviceActions ();
    }
  }

  MyClass::processRequest
             (GAT::Request          request,
              GAT::RequestNotifier notifier)
  {
     bool result = FALSE;

     if ( request.GetRequestName == ”checkpoint” )
     {
        result = checkpoint ();
     }

     GAT::Table  tab;

     tab.add (”name”,     ”checkpoint_result”);
     tab.add (”datatype”, ”bool”);
     tab.add (”value”,    result);

     GAT::Metric metric (tab);
     notifier.respond   (metric);
  };
```

2) Event Creation: The procedure for creating events is very similar to the sequence described above. The application creates a `GATMetric` object describing the event to be issued, and fires that event by calling `GATSelf::FireEvent (metric)`. The engine triggers the matching adaptor to the monitoring services, which injects that event into the system.

E. Information Management Subsystem

The GAT's connection to a generic Grid Information System (GIS) is the GAT Advertisement Subsystem which is represented primarily by the `GATAdvertService` object. Such a service is a persistent external repository for any information which may be useful outside the application itself. This information may include (e.g.):

- files published by the application,
- port and protocol information for contacting the application online, and
- information about jobs spawned by the application.

To publish application-specific information to a GIS, the `GATAdvertService` provides a means for annotating the appropriate `GATObjects` with arbitrary metadata, and for storing this information in a hierarchical namespace within the GIS. This namespace is also called the Advertisement Directory (AD). The nodes in the AD are tuples containing the absolute path in that namespace, the associated `GATObject`, and the attached set of metadata. The metadata is considered to be a list of key-value pairs, where keys and values are strings.

For inclusion in the `GATAdvertService`, `GATObjects` must implement the `GATAdvertisable` interface. The `GATAdvertisable` class requires the implementation of the methods `Serialize` and `DeSerialize`. This allows to attach the serialization of `GATObjects` to a node in the AD, and to re-instantiate this object on retrieval from the AD.

The namespace of the advert directory resembles a standard file system namespace. This includes support for the notions of absolute and relative paths, root directory, current working directory, and home directory.

The AD supports the search for advertised objects in the current directory (recursively or non-recursively) by lookups in the metadata. In order to initiate a search operation, a list of key-value pairs (query meta data) needs to be created. This list will be matched against the stored metadata of the objects in the `GATAdvertService`. The value elements in the query meta data support regular expressions to be interpreted during query execution, such as `http://.+\.org:\d+/data/.+` to specify existing, empty, or matching metadata values.

Metadata keys starting with `GAT_` are reserved for internal use, such as storing the path to the node `GAT_PATH`, the type of the published object `GAT_TYPE`, or the name of the node `GAT_NAME`. These special metadata attributes are generated by the GAT engine and are always available to the user, and may also be searched.

The `GATAdvertService` provides the following methods:

- `Add`
  publishes a new node in the AD.
- `Delete`
  deletes the specified node from the AD
- `GetMetaData`
  gets the complete set of metadata from the specified node.
- `GetAdvertisable`
  retrieves the `GATObject` associated with the specified node.
- `Find`
  searches (recursively or non-recursively) for nodes matching the given specification.
- `SetPWD` and `GetPWD`
  handle the users current working directory in the AD.

Fig. 11 shows example code to advertise a task; Fig. 12 shows the code to kill this task by another application.

Fig. 11. Code Example: *Advertise a spawned subtask*

```
bool advertJob (GAT::Job j)
{
  GAT::AdvertService as (context);
  GAT::Table         tab;

  tab.set (”name”,   ”subtask-123”);

  as.add  (”/gridlab/ullmer/adverts/subtask”,
           j, tab);
}
```

Fig. 12. Code Example: *Kill an advertised subtask*

```
bool killJob (void)
{
  GAT::AdvertService as (context);
  GAT::Table         tab;

  tab.set    (”GAT_TYPE”, ”GAT::Job”);
  tab.set    (”name”,     ”subtask-\d+”); // regex

  as.setPWD (”/gridlab/ullmer/adverts/”);
  list<String> paths = as.find (tab, GAT_RECURSIVE);

  GAT::Job j = as.getAdvertiseable (paths[0]);
  // "/gridlab/ullmer/adverts/subtask"

  task.stop ();
}
```

F. Utility subsystem

The GAT API contains a number of classes providing general, convenience, and/or utility oriented methods. This section gives a short overview of these classes.

The implementation and presence of these classes is highly language dependent. For example, in the case of Java, several of these classes will not be implemented, as Java provides a native equivalent.

1) The `GATTime` and `GATTimePeriod` classes: The `GATTime` and `GATTimePeriod` classes handle a concrete time and a period of time, respectively. They allow the integration of timing values into the serialization framework provided by the GAT engine, and also support the specification of resource reservation requests.

2) The `GATLocation` class: The `GATLocation` class was introduced to encapsulate an arbitrary URL. It supports verifying the correctness of any given URL by applying the rules defined in the RFC 1738 [28], and also extracting or modifying any part of a URL. For example, this class is used inside the GAT API wherever a physical file location needs to be be specified.

3) The `GATTable` class: The `GATTable` class is a general ordering container. It holds arbitrary key-value pairs, where the keys may be of any type which provides an ordering function. The `GATTable` may be used to store a wide spectrum of data types as strings, integers, `GATObjects`, or structures. The GAT API uses this class to store data collections, preferences, meta data, configuration data, etc.

4) The `GATSecurityContext` class: The `GATSecurityContext` class stores security information necessary for authentication and authorization purposes. For example, it may hold user names and passwords, certificates, or the location of the remote credential server to use. The `GATSecurityContext` is used by the `GATContext`.

V. Related Work

The application programming interfaces (APIs) of existing grid middleware are both heterogeneous and rapidly evolving. For example, the Globus toolkit [2] currently provides the defacto standard grid middleware. Its API has been designed as fully featured, enabling access to all details of a grid, its resources, and services. A drawback of this approach is that even simple application requirements result in elaborate code, as shown in the Appendix. In contrast, the GAT strives to simplify the API, transferring some control from the user to the adaptor for a given functionality.

OGSA, the Open Grid Service Architecture [6], was introduced to unify access to grid resources in 2002. Although a big step in the right direction, OGSA still does not prescribe the actual API of a middleware; it merely facilitates the protocols by which remote resources can be reached. Likewise, WSRF, the Web Service Resource Framework [29] does not intrinsically contribute to simple and stable APIs. In fact, a layer on top of these middleware frameworks is needed; this layer is the target of GAT.

Unicore [4] provides an alternative to Globus. However, its functionality is restricted to job submission and user authentication. In lacking further functionality like file access or job steering, Unicore does not provide a real API. Unicore adaptors are currently being developed for the GAT, allowing submission of jobs from a running application via Unicore services. In combination with other GridLab services, this will broaden the capabilities of Unicore applications.

Condor [30] was designed for job submission to idle workstations within a local cluster. Condor-G [3] allows job submission to grid machines using Globus. An important reason for the success of Condor is the approach of *transparent* remote execution, using special runtime libraries to relay system calls to the local machine of the job submitter. Condor does not need a grid-related API at all, which is a strong plus for application developers. A combination of Condor with GAT functionality could overcome some of its limitations, such as access to grid resources beyond the job submitter's machine, while still keeping the grid API as simple as possible.

Another important area of functionality is communication between multiple processes belonging to a grid application. The most widely used API is MPI, the Message Passing Interface [5]. MPI has been designed for classical parallel computing, where high performance is more important than flexibility and dynamic reconfiguration. MPI's restrictions in these respects limit its usefulness for grid environments. Also, the MPI interface is known to be very complex. The GAT provides a simple pipe mechanism for inter-process communication. Unlike MPI, this mechanism is not tuned for tightly-coupled applications and provides only very simple point-to-point communication.

Ibis [31] is a Java-based grid programming environment, focusing on both flexibility and efficiency of inter-process communication. Its communication mechanisms are built using Java's method invocation paradigm. Once completed, the Java version of the GAT will complement the Ibis functionality, forming an expressive, yet simple, grid API.

The Java Commodity Grid (CoG) Kit [32] provides access to grid services from Java, aiming at simplified user interfaces. Unlike the GAT, the Java CoG Kit offers an interface to Globus-specific, low-level grid services, such as GridFTP, GSI and the MDS. The GAT offers higher level abstractions like file movement that are independent of the underlying middleware infrastructure.

GAF4J, the Grid Application Framework for Java [33] is an application framework for multi-threaded Java applications. GAF4J replaces thread objects by so-called *task* objects that can be executed on remote machines in a grid. The GAT, however, has a broader applicability and provides high-level interfaces to resource management, grid monitoring, information services, remote file access, etc.

The Global Grid Forum (GGF [34]) has many groups working on the development of grid APIs for various aspects like resource management (DRMAA [35]), remote data access (GridFTP [36]), application checkpointing (GridCPR [7]), or remote procedure calls (GridRPC [37]). Together with other researchers, the authors recently have established a new group, called SAGA [38], addressing a *Simple API for Grid Applications*. The primary goal of this group is to develop a standardized, simple grid API, blending GAT concepts with input from other application users and developers within GGF.

VI. Conclusions

Grid computing offers promise for a world with many new opportunities. Grid applications will have the potential to integrate geographically dispersed compute resources, data repositories, scientific instruments, and human users. However, only a few grid applications have actually been deployed so far. We attribute this largely to the complexity and rapid changes of the programming interfaces for existing grid middleware. For grids to achieve widespread deployment, the availability of an application programming interface (API) that is both easy-to-use and platform-independent is of vital importance.

To address these issues, we have presented the *Grid Application Toolkit* (GAT) which has been designed and built by the European GridLab project. The GAT defines a simple, platform-independent API to grid resources and services. The functionality of the GAT API has been defined to meet the needs of dynamic grid application scenarios. The GAT focuses on resource management (job submission and migration), data management (access to files and pipes), event management (application monitoring and control), and information management (application-specific meta data). However, the GAT does *not* attempt to cover all possible use cases, thus favoring simplicity over exhaustive functionality.

The GAT is implemented as a runtime library against which applications can be linked. The *GAT engine* implements the API as a thin wrapper layer which dispatches incoming API calls to the services which happen to be available and appropriate within the evolving grid environment. Services are made available to the GAT engine via *adaptors*. An adaptor provides the interface between a given GAT functionality and the services which offer corresponding capabilities. Such services might range from basic operating system calls on a local laptop, to a computing service provided on an HPC machine. At runtime, adaptors are dynamically linked into the GAT engine (where the architecture allows), allowing dynamic selection of appropriate services. The GAT can utilize multiple grid middleware environments, possibly in multiple versions, at the same time.

Widespread usage of the GAT also requires embodiments which are compatible with existing and potential grid applications. Therefore, the current implementation of GAT engine and adaptors is written in the C language, with wrappers for C++ close to completion, and with Fortran, Perl, and Python soon to follow. An independent, native implementation in Java is currently under development. With this set of supported languages, we expect to cover a wide spectrum of potential application codes. The current set of adaptors uses both local resources as well as Globus (v2) and GridLab-specific services, as introduced above. Additional adaptors for Unicore are currently under development. To foster openness and widespread deployment, all GridLab software has been made open source, with availability from *www.gridlab.org*.

By combining a simple, flexible API with support for key programming languages and grid middleware packages, we are confident the GAT has strong potential to provide the general-purpose, high-level API that application developers have been seeking in recent years. Given its open design, the GAT will be able to adapt to upcoming middleware layers (e.g., the WSRF framework) without forcing end users to explicitly confront and address the specific details of these systems.

ACKNOWLEDGEMENTS

The Grid Application Toolkit is funded as part of the GridLab project by the European Commission, grant IST-2001-32133. The authors acknowledge the many colleagues from GridLab who have contributed to the concept and development of the GAT. Special thanks go to Peggy Lindner at HLRS, Stuttgart for contributing Unicore adaptors for the GAT.

REFERENCES

[1] G. Allen, K. Davis, K. N. Dolkas, N. D. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, J. Shalf, and I. Taylor, “Enabling Applications on the Grid – A GridLab Overview,” *International Journal on High Performance Computing Applications*, vol. 17, no. 4, pp. 449–466, 2003.

[2] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” *Int. Journal of Supercomputer Applications*, vol. 11, no. 2, pp. 115–128, 1997.

[3] D. Thain, T. Tannenbaum, and M. Livny, “Condor and the grid,” in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and T. Hey, Eds. John Wiley & Sons Inc., December 2002.

[4] D. Erwin (Ed.), *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003. [Online]. Available: http://www.unicore.org/documents/UNICOREPlus-Final-Report.pdf

[5] Message Passing Interface Forum, “MPI: A Message Passing Interface Standard,” *International Journal of Supercomputing Applications*, vol. 8, no. 3/4, 1994.

[6] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, “Grid Services for Distributed System Integration,” *Computer*, vol. 35, no. 6, pp. 37–46, 2002.

[7] GGF. (2003) Grid Checkpoint Recovery Working Group. [Online]. Available: http://forge.gridforum.org/projects/gridcpr-wg/

[8] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, “A security architecture for computational Grids,” in *ACM Conference on Computer and Communications Security*, 1998, pp. 83–92.

[9] R. A. Van Engelen and K. A. Gallivan, “The gsoap toolkit for web services and peer-to-peer computing networks,” in *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, 2002, p. 128.

[10] G. Aloisio, M. Cafaro, D. Lezzi, and R. V. Engelen, “Secure Web Services with Globus GSI and gSOAP,” in *Proceedings of Euro-Par 2003*, no. 2790. Klagenfurt, Austria: Lecture Notes in Computer Science, Springer-Verlag, 2003, pp. 421–426.

[11] R. W. Moore and A. Merzky, “Persistent archive concepts,” *Global Grid Forum, GGF Informational Document - GFD.26*, November 2003. [Online]. Available: http://forge.gridforum.org/projects/ggf-editor/document/GFD.26/en/2

[12] GridLab Project. (2003) The GridLab Resource Management System. [Online]. Available: http://www.gridlab.org/grms

[13] GridLab Project. (2004) GRMS v1.9.0 Admin Guide. [Online]. Available: http://www.gridlab.org/WorkPackages/wp-9/res/stuff/grms1.9-admin-guide.pdf

[14] GridLab Project. (2002) The Grid Authorization Service. [Online]. Available: http://www.gridlab.org/security

[15] G. Aloisio, M. Cafaro, I. Epicoco, D. Lezzi, M. Mirto, and S. Mocavero, “The Design and Implementation of the GridLab Information Service,” in *Second International Workshop on Grid and Cooperative Computing (GCC 2003)*. Lecture Notes in Computer Science, Springer-Verlag, 2003.

[16] G. Aloisio, M. Cafaro, S. Fiore, and M. Mirto, “The GRelC Library: A Basic Pillar in the Grid Relational Catalog Architecture,” in *Proceedings of Information Technology Coding and Computing (ITCC)*, vol. I, 2004, pp. 372–376.

[17] GridLab Project. (2002) The Replica Management Service. [Online]. Available: http://www.gridlab.org/data

[18] G. Project. (2002) The File Movement Service. [Online]. Available: http://www.gridlab.org/data

[19] H.-C. Hege, A. Hutanu, R. Kähler, A. Merzky, T. Radke, E. Seidel, and B. Ullmer, “Progressive Retrieval and Hierarchical Visualization of Large Remote Data,” in *Workshop on Adaptive Grid Middleware*, St. Louis, USA, September 2003.

[20] T. Schütt, A. Merzky, A. Hutanu, and F. Schintke, “Remote Partial File Access Using Compact Pattern Descriptions,” in *Proceedings of the 4th International Symposium on Cluster Computing and the Grid (CCgrid)*, 2004.

[21] GridLab Project. (2003) The Delphoi Service. [Online]. Available: http://www.gridlab.org/delphoi

[22] P. Kacsuk, G. Dózsa, J. Kovács, R. Lovas, N. Podhorszki, Z. Balaton, and G. Gombás, “P-GRADE: A Grid Programming Environment,” *Journal of Grid Computing*, vol. 2, pp. 171–197, 2003.

[23] K. Davis, T. Goodale, and A. Merzky, “GAT API Specification: Object Based,” *EU Project GridLab, Deliverable D1.5*, June 2003. [Online]. Available: http://www.gridlab.org/WorkPackages/wp-1/Documents/Gridlab-1-GAS-0004.ObjectBasedAPISpecification.pdf

[24] GGF. (2004) The JSDL Specification - Draft. [Online]. Available: http://forge.gridforum.org/projects/jsdl-wg/document/draft-ggf-jsdl-spec/en/8

[25] A. Denis, O. Aumage, R. Hofman, K. Verstoep, T. Kielmann, and H. E. Bal, “Wide-Area Communication for Grids: An Integrated Solution to Connectivity, Performance and Security Problems,” in *HPDC-13*. IEEE, 2004, pp. 97–106.

[26] F. Isaila and W. Tichy, "Mapping functions and data redistribution for parallel files," *Proceedings of IPDPS 2002 Workshop on Parallel and Distributed Scientific and Engineering Computing with Applications*, April 2002.

[27] C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," *Proceedings of CASCON'98, Toronto, Canada*, 1998.

[28] T. Berners-Lee, L. Masinter, and M. McCahill, "RFC 1738: Uniform Resource Locators (URL)," http://www.ietf.org/rfc/rfc1738.txt, 1994. [Online]. Available: http://www.ietf.org/rfc/rfc1738.txt

[29] D. F. Snelling, "Web Services Resource Framework: Impact on OGSA and the Grid Computing Roadmap," *GridConnections – News and Information for the Global Grid Forum Community*, vol. 2, 2004.

[30] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[31] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal, "Ibis: a Flexible and Efficient Java-based Grid Programming Environment," *Concurrency and Computation: Practice and Experience*, 2004.

[32] G. von Laszewski, I. Foster, J. Gawor, and P. Lane, "A Java Commodity Grid Kit," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 8–9, pp. 643–662, 2001, http://www.cogkits.org.

[33] IBM. (2003) Grid Application Framework for Java. [Online]. Available: http://www.alphaworks.ibm.com/tech/GAF4J

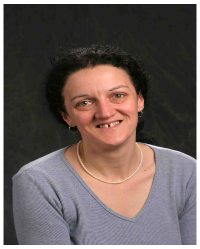
[34] GGF. (2001) Global Grid Forum. [Online]. Available: http://www.ggf.org/

[35] GGF. (2003) Distributed Resource Management API Working Group. [Online]. Available: http://forge.gridforum.org/projects/drmaa-wg/

[36] GGF. (2001) GridFTP Working Group. [Online]. Available: http://forge.gridforum.org/projects/gridftp-wg/

[37] GGF. (2002) Grid Remote Procedure Call Working Group. [Online]. Available: http://forge.gridforum.org/projects/gridrpc-wg/

[38] GGF. (2004) Simple API for Grid Applications Research Group. [Online]. Available: http://forge.gridforum.org/projects/saga-rg/



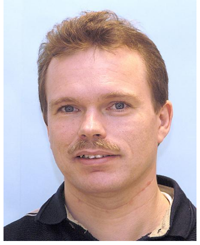
Gabrielle Allen is associate professor of computer science at Louisiana State University, and Assistant Director for Computing Applications at the Center for Computation & Technology. She obtained her PhD in Astrophysics at Cardiff University in 1993 after obtaining the Certificate of Advanced Study in Applied Mathematics and Theoretical Physics from Cambridge University in 1989, and a B.S. in Mathematics from Nottingham University in 1988. Her interests include grid and high performance computing, computational science and numerical relativity.

Kelly Davis received a degree in Mathematics from M.I.T. in Boston, U.S.A. and studied towards a Ph.D. in high energy theoretical particle physics, in particular superstring theory, at Rutgers in New Brunswick, U.S.A. He then went on to various industry positions, programming in various languages, in which he implemented solutions varying from Java web applications to Prolog knowledge based speech applications. His research interests vary from grid computing to the holographic conjecture.

Tom Goodale is a research associate at the Center for Computation & Technology at Louisiana State University. He received his BA in Theoretical Physics from Cambridge University in 1992, and, after obtaining the Certificate of Advanced Study in Applied Mathematics and Theoretical Physics from Cambridge in 1993, worked as a software engineer in the Oil and Gas industry before returning to academia in 1996. He received a masters in Computational Fluid Dynamics from Cranfield University in 1998. His interests include high performance computing, frameworks, grid computing, CFD and numerical relativity.

Andrei Hutanu received his diploma in Computer Science from Politehnica University of Bucharest in 2002. Currently he is a researcher in the visualization department of the Zuse Institute Berlin, with research interests in distributed visualization, data management and high level grid services.

Hartmut Kaiser received his diploma in Computer Science at the Leningrad Electrotechnical University, Petersburg, Russia, in 1985, a Ph.D. and the habilitation in Computer Engineering, in 1988, both from the Technical University of Chemnitz, Germany. Currently he is a research programmer at the Albert Einstein Institute for Gravitational Physics in Golm, Germany. His research interests include application programming interfaces to the grid, parsing problems and the C++ programming language in general.



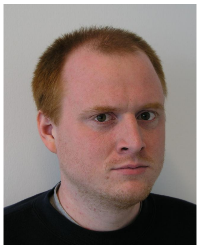
Thilo Kielmann is Associate Professor at the Department of Computer Science of Vrije Universiteit. He received his diploma in Computer Science from Darmstadt University of Technology, Germany, in 1992, a Ph.D. in Computer Engineering in 1997, and the habilitation in Computer Science in 2001, the latter both from the University of Siegen, Germany. His research interests include computational grids, performance analysis of networked applications, and parallel and distributed programming.

André Merzky received his diploma in Particle Physics in 1998 at the Humboldt University in Berlin. He has worked since on Grid-related topics concerning data management and visualization, and is active in the Data Area of the Global Grid Forum (GGF). He is leading the Technical Board of the GridLab project. Currently, he is working at the Computer Science Department at the Vrije Universiteit.

Rob van Nieuwpoort received his M.Sc. and Ph.D. in Computer Science at Vrije Universiteit in 1998 and 2003, respectively. Currently, he is a postdoctoral researcher at the Computer Science department of Vrije Universiteit, working on the GridLab project. His research interests include Java-centric grid computing, divide-and-conquer parallelism and design and implementation of parallel languages.

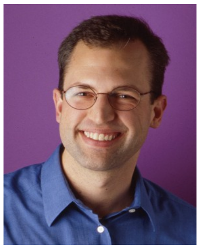
Alexander Reinefeld is director of Computer Science at the Zuse Institute Berlin (ZIB) and professor for parallel and distributed systems at the Humboldt-Universität zu Berlin. In his previous position as managing director of the Paderborn Center for Parallel Computing (PC^2), he played an important role in establishing the PC^2 as a super-regional competence center for massively parallel computing in the years 1992 to 1998. Before, he was an assistant professor (1989-1992) and a research assistant (1983-1987) at the Universität Hamburg. In 1987, he was awarded a Sir Izaak Walton Killam Memorial Post Doctoral Fellowship by the University of Alberta and in 1984, he was awarded a PhD scholarship by the German Academic Exchange Service. He graduated with a computer science diploma in 1982 and a PhD in 1987 at the University of Hamburg.

Florian Schintke is a PhD candidate with the Zuse Institute Berlin (ZIB). He graduated in 2000 with distinction from the Technical University, Berlin. Since then he works as a research staff member in the Computer Science Research Department at ZIB and participates in the EU projects DataGrid, GridLab, and FlowGrid. He is a member of the German Computer Science Society and the IEEE Task Force on Cluster Computing. His research interests are in the areas of distributed data management, scalable grid systems and autonomic computing.



Thorsten Schütt is a PhD candidate with the Zuse Institute Berlin (ZIB). He got his Diploma with distinction in 2002 from the Technical University Berlin. Since then he works as a research staff member in the Computer Science Research Department at ZIB. His research interests include distributed data management, scalable grid systems and p2p computing.

Ed Seidel is the director of the Center for Computation & Technology (CCT) at Louisiana State University in Baton Rouge, Louisiana and a senior scientist at the Max Planck Institute for Gravitational Physics in Potsdam, Germany. He received his Ph.D. in Relativistic Astrophysics in 1988 from Yale University, a M.S. in Physics from The University of Pennsylvania, and a B.S. in Mathematics and high honors in Physics from The College of William and Mary. His research interests include general relativity, relativistic astrophysics, grid computing, computational science, and high performance computing. He is a co-chair of the Applications Research Group of the Global Grid Forum.



Brygg Ullmer is a postdoctoral researcher in the visualization department of the Zuse Institute Berlin (ZIB). He received his M.S. and Ph.D. at the MIT Media Lab in 1997 and 2002. He also holds a B.S. in computer engineering from the University of Illinois, Urbana-Champaign (1994). His research interests include tangible and graphical user interfaces for interaction with online media, complex infrastructure (including the grid), and biological systems, as well as rapid physical prototyping.

APPENDIX

This appendix contains the full source code for the file copy example, using OGSA/Java and Globus/C. The first listing contains a file copy implementation in Java using OGSA. The developer has to deal with several aspects in this code. In the first half he has to set several parameters concerning the file transfer (e.g. block size, TCP buffer size and number of streams) whereas in the second half he has to deal with the OGSA protocol.

```
package org.globus.ogsa.gui;

import java.io.*;
import java.net.URL;
import java.util.*;
import javax.xml.rpc.Stub;
import org.apache.axis.message.MessageElement;
import org.apache.axis.utils.XMLUtils;
import org.globus.axis.gsi.GSIConstants;
import org.globus.ogsa.*;
import org.gridforum.ogsi.*;
import org.globus.gsi.proxy.IgnoreProxyPolicyHandler;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class RFTClient {
  public static void copy (String source_url,
                           String target_url) {
    try {
      File requestFile      = new File (source_url);
      BufferedReader reader = null;

      try {
        reader = new BufferedReader (
          new FileReader (requestFile));
      }
      catch (java.io.FileNotFoundException fnfe) { }

      Vector requestData = new Vector ();
      requestData.add (target_url);

      TransferType[] transfers1      =
        new TransferType[transferCount];
      RFTOptionsType multirftOptions =
        new RFTOptionsType ();

      multirftOptions.setBinary(Boolean.valueOf(
        (String)requestData.elementAt (0))
          .booleanValue());
      multirftOptions.setBlockSize(Integer.valueOf(
        (String)requestData.elementAt (1))
          .intValue());
      multirftOptions.setTcpBufferSize(Integer.valueOf(
        (String)requestData.elementAt (2))
          .intValue());
      multirftOptions.setNotpt(Boolean.valueOf(
        (String)requestData.elementAt (3))
          .booleanValue());
      multirftOptions.setParallelStreams(Integer
        .valueOf(
          (String)requestData.elementAt (4))
            .intValue());
      multirftOptions.setDcau(Boolean.valueOf(
        (String)requestData.elementAt (5))
          .booleanValue());

      int i = 7;
      for (int j = 0; j < transfers1.length; j++) {
        transfers1[j] = new TransferType ();
        transfers1[j].setTransferId(j);
        transfers1[j].setSourceUrl(
          (String)requestData.elementAt (i++));
        transfers1[j].setDestinationUrl(
          (String)requestData.elementAt (i++));
        transfers1[j].setRftOptions(multirftOptions);
      }

      TransferRequestType transferRequest =
        new TransferRequestType ();
      transferRequest.setTransferArray (transfers1);

      int concurrency = Integer.valueOf(
        (String)requestData.elementAt(6))
          .intValue();

      if (concurrency > transfers1.length) {
        System.out.println(
          "Concurrency␣should␣be␣less␣than␣the␣number"
          "of␣transfers␣in␣the␣request");
        System.exit (0);
      }

      transferRequest.setConcurrency (concurrency);

      TransferRequestElement requestElement = new TransferRequestElement ();
      requestElement.setTransferRequest (transferRequest);

      ExtensibilityType extension = new ExtensibilityType ();
      extension = AnyHelper.getExtensibility (requestElement);

      OGSIServiceGridLocator factoryService = new OGSIServiceGridLocator ();
      Factory factory = factoryService.getFactoryPort (new URL (source_url));
      GridServiceFactory gridFactory = new GridServiceFactory (factory);
      LocatorType locator = gridFactory.createService (extension);
      System.out.println ("Created␣an␣instance␣of␣Multi-RFT");

      MultiFileRFTDefinitionServiceGridLocator loc
                          = new MultiFileRFTDefinitionServiceGridLocator();
      RFTPortType rftPort = loc.getMultiFileRFTDefinitionPort (locator);

      ((Stub)rftPort)._setProperty (Constants.AUTHORIZATION,
                                    NoAuthorization.getInstance());
      ((Stub)rftPort)._setProperty (GSIConstants.GSI_MODE,
                                    GSIConstants.GSI_MODE_FULL_DELEG);
      ((Stub)rftPort)._setProperty (Constants.GSI_SEC_CONV,
                                    Constants.SIGNATURE);
      ((Stub)rftPort)._setProperty (Constants.GRIM_POLICY_HANDLER,
                                    new IgnoreProxyPolicyHandler ());

      int requestid = rftPort.start ();
      System.out.println ("Request␣id:␣" + requestid);
    }
    catch (Exception e) {
      System.err.println (MessageUtils.toString (e));
    }
  }
}
```

In contrast to the OGSA example, the following Globus code does not have to deal with a special protocol for calling the copy function. It just contains file-copy related calls. Still, there is quite a lot of code setting up various data structures.

```
int RemoteFile::GetFile (char const* source,
                         char const* target) {
  globus_url_t                       source_url;
  globus_io_handle_t                 dest_io_handle;
  globus_ftp_client_operationattr_t  source_ftp_attr;
  globus_result_t                    result;
  globus_gass_transfer_requestattr_t source_gass_attr;
  globus_gass_copy_attr_t            source_gass_copy_attr;
  globus_gass_copy_handle_t          gass_copy_handle;
  globus_gass_copy_handleattr_t      gass_copy_handleattr;
  int                                output_file = -1;
  globus_ftp_client_handleattr_t     ftp_handleattr;
  globus_io_attr_t                   io_attr;

  if ( globus_url_parse (source_URL, &source_url) != GLOBUS_SUCCESS ) {
    printf ("can␣not␣parse␣source_URL␣\"%s\"\n", source_URL);
    return (-1);
  }

  if ( source_url.scheme_type != GLOBUS_URL_SCHEME_GSIFTP &&
       source_url.scheme_type != GLOBUS_URL_SCHEME_FTP    &&
       source_url.scheme_type != GLOBUS_URL_SCHEME_HTTP   &&
       source_url.scheme_type != GLOBUS_URL_SCHEME_HTTPS  ) {
    printf ("can␣not␣copy␣from␣%s␣-␣unsupported␣protocol\n", source_URL);
    return (-1);
  }

  globus_gass_copy_handleattr_init  (&gass_copy_handleattr);
  globus_gass_copy_attr_init        (&source_gass_copy_attr);

  globus_ftp_client_handleattr_init (&ftp_handleattr);
  globus_io_fileattr_init           (&io_attr);

  globus_gass_copy_attr_set_io      (&source_gass_copy_attr, &io_attr);
  globus_gass_copy_handleattr_set_ftp_attr
                                    (&gass_copy_handleattr, &ftp_handleattr);
  globus_gass_copy_handle_init      (&gass_copy_handle, &gass_copy_handleattr);

  if (source_url.scheme_type == GLOBUS_URL_SCHEME_GSIFTP ||
      source_url.scheme_type == GLOBUS_URL_SCHEME_FTP    ) {
    globus_ftp_client_operationattr_init  (&source_ftp_attr);
    globus_gass_copy_attr_set_ftp         (&source_gass_copy_attr,
                                           &source_ftp_attr);
  }
  else {
    globus_gass_transfer_requestattr_init (&source_gass_attr, source_url.scheme);
    globus_gass_copy_attr_set_gass(&source_gass_copy_attr,    &source_gass_attr);
  }

  output_file = globus_libc_open ((char*) target,
                                  O_WRONLY | O_TRUNC | O_CREAT,
                                  S_IRUSR  | S_IWUSR | S_IRGRP | S_IWGRP);

  if ( output_file == -1 ) {
    printf ("could␣not␣open␣the␣destination␣file␣\"%s\"\n", target);
    return (-1);
  }

  /* convert stdout to be a globus_io_handle */
  if ( globus_io_file_posix_convert (output_file, GLOBUS_NULL, &dest_io_handle)
       != GLOBUS_SUCCESS) {
    printf ("Error␣converting␣the␣file␣handle\n");
    return (-1);
  }

  result = globus_gass_copy_register_url_to_handle (
```

```
                                                   &gass_copy_handle,
                                                   (char*)source_URL,
                                                   &source_gass_copy_attr,
                                                   &dest_io_handle,
                                                   my_callback,
                                                   NULL);

  if ( result != GLOBUS_SUCCESS ) {
    printf ("error:␣%s\n", globus_object_printable_to_string
                              (globus_error_get (result)));
    return (-1);
  }

  globus_url_destroy  (&source_url);

  return (0);
}
```