A Day in the Life of a Grid-Enabled Application:
Counting on the Grid

Gabrielle Allen[1,2], Tom Goodale[1,2], Hartmut Kaiser[1], Thilo Kielmann[3],
Archit Kulshrestha[2], André Merzky[3], Rob V. van Nieuwpoort[3]

[1] Albert Einstein Institute, Golm, Germany
[2] Louisiana State University, Baton Rouge, USA
[3] Vrije Universiteit, Amsterdam, The Netherlands
http://www.gridlab.org

Abstract

While Grid technologies are maturing rapidly, there still remains a shortage of real Grid applications. One important reason is the lack of simple and high-level application programming interfaces to the Grid, 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 a unified, simple programming interface to the Grid infrastructure, tailored to the needs of Grid application programmers and users. In this paper, we outline a motivating use case, present the GAT API functionality, and sketch existing bindings to programming languages and their implementations.

1 Introduction

In Grid platforms, or Virtual Organizations (VO's) [4], applications access a variety of more or less heterogeneous resources. For most applications, the important resources are computers and data files. For computers, heterogeneity stems from different hardware architectures and operating systems. Besides, the administrative autonomy of the member sites in a VO leads to a diversity of access policies and installed middleware packages and package versions. Unfortunately, even minor differences between versions of the same middleware package often lead to prohibitive incompatibilities. Access to data files suffers from similar problems.

Hiding this heterogeneity and even the individual entities in favour of providing a unified distributed computing platform is important for writing Grid-enabled applications. For this purpose, the GridLab project [1] has been developing the Grid Application Toolkit (GAT) [2]. The GAT's main objective is to provide a single, easy-to-use Grid API, while hiding the complexity and diversity of the actual Grid resources and their middleware layers.

The GAT provides abstractions for computers, data, and application instances (jobs running on a VO).

In this paper, we outline a motivating use case of a simple Grid-enabled application and its required operations on Grid resources, along with GAT-based code examples (Section 2). In Section 3, we briefly summarize the whole GAT API. Section 4 outlines the GAT architecture and its current implementations. We conclude in Section 5.

2 A Day in the Life of a Grid-enabled Application

In this section we present a simple, but typical, Grid-enabled application. On startup, it reads some input data from a file, then does some computation, and writes the output back to a file. In addition, our application migrates to another machine afterwards. The input data file is stored somewhere in the Grid, so it has to be found by the application before it can be read. Likewise, our application has to make sure that it can find its output file again, after having migrated to another machine. For simplicity of presentation, our application is a toy counter: the data set consists of a single number and the "computation" merely increments this value. Figure 1 outlines the application structure. In the following, we will walk through the code and show how such an application can easily be written with the GAT API.
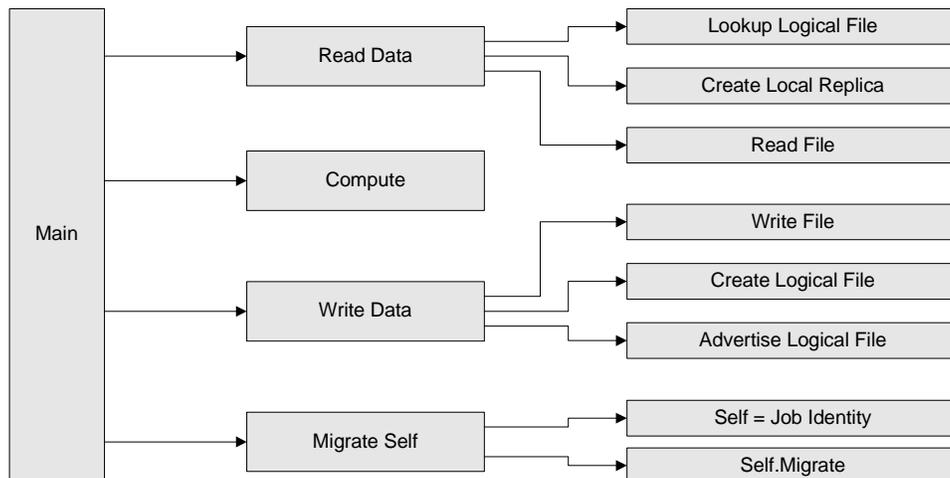

Figure 1: Structure of the Grid-enabled counter application.

2.1 The Main Function

The *main* function implements the operation sequence from Figure 1, extended by a termination condition. For simplicity of presentation, all error handling is done here as well. The

class *MigrateApp* (not shown) is a regular C++ class. Besides the application-specific code, it contains the *counter* value and the *GAT::Context*, used for GAT operations.

```cpp
int main(int argc, char *argv[]) {
   try {
      // initialize the GAT
      GAT::Self::SetExecutionEnvironment(argc, argv);

      // initialize application
      bool finished = false;
      MigrateApp app;

      app.ReadCounter();                // read the input data
      finished = app.Compute();         // increment the counter
      app.WriteCounter();               // write the output data
      app.MigrateSelf();                // migrate this app
      app.Terminate(finished);          // clean up
   }
   catch(GAT::Exception const &e) {
      // a GAT specific error was catched
      std::cerr << e.what() << std::endl; // print out error message
      return e.GetResult();
   }
   catch(std::exception const &e) {
      // a general error was catched
      std::cerr << e.what() << std::endl;
      return -1;
   }
   return 0;
}
```

2.2   Read Data

Reading the input data (the current counter value) involves the discovery of the input file. For this purpose, the GAT provides the concept of *logical files* which are collections of replicas. The logical files themselves are located via a registry called the *AdvertService*. Our application thus first does a lookup for the logical file. It then creates a local (physical) replica which finally gets read. Creating a local replica is beneficial in the case of large input files. However, directly reading a remote replica would also be possible.

Within the *AdvertService* directory, the information can be found using a given key, represented via the constant *ADVERT_PATH*. The actual, local file name is represented by *DATAFILE_IN*. If the logical file could not be discovered, the application starts from scratch.

```cpp
#define ADVERT_PATH     "/ggf12-paper/migrate_app"
#define DATAFILE_IN     "gat_in.dat"

void MigrateApp::ReadCounter() {
   // Lookup Logical File
   GAT::LogicalFile logical_file;
```

```
    if (LookUpLogicalFile(ADVERT_PATH, logical_file)) {
        // create local replica, read current counter value
        GAT::FileStream local_replica = CreateLocalReplica(logical_file, DATAFILE_IN);
        counter = ReadCounterFromFile(local_replica);
    }
    else {
        // No input file exists, restart from scratch
        counter = 0;
    }
}
```

2.2.1   Lookup Logical File

The *AdvertService* gets a table of key/value pairs for lookup and returns a list of matching entries. In our example, we are looking for entries that have a *JOBID* key, and any arbitrary value. The *AdvertService* does regular-expression matching, so we specify the string ".+" matching all entries. We could also be more selective and use the application's own job id. This id could be retrieved from the GAT *context* and is kept constant across job migrations. As our application expects exactly one result, it takes the first list entry. After successful lookup, we clean up the advert service database and return the created logical file.

```
bool MigrateApp::LookUpLogicalFile(
                char const *advert_path, GAT::LogicalFile &logical_file) {
    // initialise the metadata patterns in the query parameter table
    GAT::Table query;
    query.Add("JOBID", ".+");    // looking for arbitrary JOBID's

    // set the pwd as the advert service search root and search for
    // existing entries
    GAT::AdvertService advertservice(context);
    advertservice.SetPWD(advert_path);

    std::list<GAT::String> paths;
    GAT_IGNORE_EXCEPTION(advertservice.Find(query, paths));

    // the list should contain exactly one path
    if (paths.size() > 0) {
        // if we got a logical file, return it to the caller
        advertservice.GetAdvertisable(paths.front(), logical_file);
        advertservice.Delete(paths.front()); // clean up advert service
        return true;
    }
    return false;         // no input data found
}
```

2.2.2   Create Local Replica

Creating a local replica is simply done by calling the respective function of the logical file object, and by constructing a local file stream to access the newly created local file.

```
GAT::FileStream MigrateApp::CreateLocalReplica(
                GAT::LogicalFile logical_file, char const *local_name) {
   GAT::Location local_replica_location(local_name);
   logical_file.Replicate(local_replica_location);
   return GAT::FileStream(context, local_replica_location);
}
```

2.2.3 Read File

Reading the actual counter value is done using standard input operations.

```
int ReadCounterFromFile(GAT::FileStream data_file) {
   char buffer[10];
   data_file.Read(buffer, sizeof(buffer));
   return atoi(buffer);  // return the counter value
}
```

2.3 Compute

After the counter has been incremented, the return value is set to indicate the termination condition.

```
bool MigrateApp::Compute() {
   ++counter;    // do the useful work ;-)
   sleep(1);

   // return true, if the application should terminate
   return counter > MAX_ITERATIONS;
}
```

2.4 Write Data

Writing the data consists of creating a file stream, writing to it, creating a logical file with the file stream as the only physical replica, and of advertising the new logical file. We delete the input file, because it isn't required anymore

```
#define DATAFILE_OUT    "gat_out.dat"
#define REPLICA_NAME    "/ggf12-paper/datafile"

void MigrateApp::WriteCounter() {
   // write counter to output file
   GAT::FileStream output_file(context, GAT::Location(DATAFILE_OUT),
                              GATFileStreamMode_Write);
   WriteCounterToFile(output_file, counter);
   output_file.Close();

   // Add the written file to a new replica and put this into the advert service
   GAT::LogicalFile logical_file = CreateLogicalFile(REPLICA_NAME, DATAFILE_OUT);
   AdvertiseLogicalFile(logical_file, ADVERT_PATH);
}
```

2.4.1 Write File

Writing the actual counter value is done using standard output operations.

```
void WriteCounterToFile(GAT::FileStream data_file, int counter) {
   char buffer[10];
   snprintf(buffer, sizeof(buffer), "%d\n", counter);
   data_file.Write(buffer, strlen(buffer));
}
```

2.4.2 Create Logical File

We create a new logical file using the same name for all iterations. By overwriting the old contents (using *GATLogicalFileMode_Truncate*), older entries of the logical file will be removed. (The physical files remain intact.) The new output file then gets associated with (added to) the logical file.

```
GAT::LogicalFile MigrateApp::CreateLogicalFile(
                char const *replica_name, char const *output_file_name) {
   // create a new logical file, overwrite existing
   GAT::LogicalFile logical_file(context, GAT::Location(replica_name),
                                 GATLogicalFileMode_Truncate);

   // associate physical file with logical file
   GAT::File output_file(context, GAT::Location(output_file_name));
   logical_file.AddFile(output_file);

   return logical_file;
}
```

2.4.3 Advertise Logical File

For advertising the logical file, we first have to construct a table with the meta data under which the logical file can later be found. This is the key *JOBID* with the value determined by the job id of the current application, as it can be retrieved via the GAT *context* object and *GAT::Self*. The *advert_path* is used to select the directory inside the *AdvertService*.

```
void MigrateApp::AdvertiseLogicalFile(
                GAT::LogicalFile logical_file, char const *advert_path) {
   // use the job id as additional metadata for the advertised data
   GAT::Job self_job(GAT::Self::GetJob(context));
   GAT::Table metadata;
   metadata.Add("JOBID", self_job.GetJobID().GetBuffer());

   // advertise the logical file
   GAT::AdvertService advertservice(context);
   advertservice.Add(logical_file, metadata, advert_path);
}
```

2.5 Migrate Self

For migration we first build a table, containing key/value pairs describing suitable machines to migrate to. Then, the table is used for building a *HardwareResourceDescription* object, using which a *ResourceBroker* can find suitable machines in our virtual organization. The resource broker returns a list of matching machines from which we simply select the first one. We extract the own job environment from *GAT::Self* and migrate the running job to the chosen machine.

```
void MigrateApp::MigrateSelf() {
   // build a table with requirements for the migration target machine
   GAT::Table hardware_requirements;
   hardware_requirements.Add("operating_system", ...);

   GAT::HardwareResourceDescription hr(hardware_requirements);
   GAT::ResourceBroker rb(context, "GGF12 demo VO");
   std::list<HardwareResource> resources = rb.FindResources(hr);

   // get own job object and migrate it to a machine found by the resource broker
   GAT::Job self(GAT::Self::GetJob(context));
   self.Migrate(resources.front());
}
```

2.6 Terminate

The purpose of the *Terminate* method is to clean up the used Grid resources, namely the logical file and the physical input file. We leave the output file on purpose. Please note that the advert service entry has already been deleted in the *LookUpLogicalFile* method.

```
void MigrateApp::Terminate(bool finished) {
   GAT::LogicalFile logical_file(context, GAT::Location(REPLICA_NAME));
   GAT::File input_file(context, GAT::Location(DATAFILE_IN));

   // remove file from replica catalog
   logical_file.RemoveFile(input_file);

   // delete physical input file
   input_file.Delete();

   if (finished) {
      // clean up replica set, when finished
      logical_file.RemoveFile(GAT::File(context, GAT::Location(DATAFILE_OUT)));
      logical_file.Remove();
   }
}
```

7

3   Summary of the GAT API

The sample application has demonstrated the use of the GAT API. However, it does not touch all its aspects and functionalities. In this section, we will summarize the whole GAT API which is organized in so-called *subsystems*. More detailed API descriptions can be found in [2, 3].

GAT Base Subsystem
The GAT base subsystem defines general objects and methods, supporting interaction with the GAT engine.

GAT::Object	Is inherited by all other objects of the API; provides common functionality.
GAT::Self	Represents the running job.
GAT::Context	Represents the current execution environment.
GAT::Status	Used for error handling.
GAT::Exception	

Data Management Subsystem
The data management subsystem covers three areas: interprocess communication, remote file access, and file management.

GAT::Endpoint, *GAT::Pipe*	Used to establish a connection between two applications and to transfer information over this connection.
GAT::File, *GAT::FileStream*	Used to access and operate on remote files using a file format-independent protocol.
GAT::LogicalFile	Management of and access to replica sets on the Grid.

Resource Management Subsystem
The resource management subsystem deals with Grid (compute) resources. It provides functions for both resource discovery and job management. The complete scheme for job descriptions within GAT is designed to support mappings to the job description languages typically used on Grids.

GAT::Job	Represents an instance of a GAT job, used to control job status and query job properties.
GAT::ResourceBroker	Used to control job submission process on the Grid; main object for resource discovery.
GAT::Resource	Any hardware or software resource on the Grid.
GAT::Reservation	Represents a resource reservation for job submission.

Event and Monitoring Subsystem
The event and monitoring subsystem allows applications to send and receive events, such as events generated by a Grid monitoring service. The programming model for this subsystem is based on subscriptions and callbacks. The application can subscribe to events of a certain type or *"metric"*, and register callbacks to handle incoming events of this type. It can also

create its own events with self-defined metrics, which makes those events available for other applications on the Grid.

GAT::Metric	Description of the event to be handled.
GAT::Request	Description of a request for information or a request for operation.
GAT::MetricEvent	Represents the event which is passed to an application.

Information Management Subsystem

The GAT's connection to a generic Grid information system is the information management subsystem which is represented primarily by the *GAT::AdvertService* object. Such a service is a persistent, external repository for any information which may be useful outside the application itself. It helps to transfer information across job life times and process boundaries, from one application to another.

GAT::AdvertService	Represents a hierarchical, remote information namespace, where *GAT::Object*'s may be stored. Provides a means for annotating the stored *GAT::Object*'s with arbitrary meta data, which may be used to discover those objects later.

Utility Subsystem

The GAT API contains a number of classes providing general, convenience, and/or utility-oriented methods. The implementation of this subsystem is highly language-dependent and uses language-specific data structures wherever possible. One example:

- The C API provides *GATList*, *GATTable* and several string-related utility functions.

- The C++ API uses the *std::list<>*, *std::map<>* and *std::string* templates.

- The Python API exposes these objects as lists ([...]) and dictionaries ({...}).

- The Java API uses *java.util.List* and *java.util.Map* from the standard collection framework.

4   GAT Architecture and Implementations

This section briefly outlines the GAT architecture and its current implementations. A more detailed presentation can be found in [2]. Figure 2 shows the GAT architecture. It mainly distinguishes between user space and capability space. In user space runs the application code that has been programmed using the GAT API. The GAT *engine* is a lightweight layer that dispatches GAT API calls to service invocations via GAT *adaptors*. Adaptors are specific to given services and hide all service-specific details from the GAT. A GAT engine typically loads adaptors dynamically at runtime, whenever a certain service is needed. The GAT functionality is grouped into various, so-called *subsystems* (see Section 3). An engine may load multiple adaptors implementing the same subsystem at the same time. For example,

one adaptor can give access to a local file system while another one gives access to files via GridFTP [5], and a third one is using a replica catalog [7].

While application and GAT together run in user space, the services are executed in the capability space, which can be distributed across the machines of a VO, including the one running the application. The capability space consists of the resources themselves, and the middleware providing services to access them.
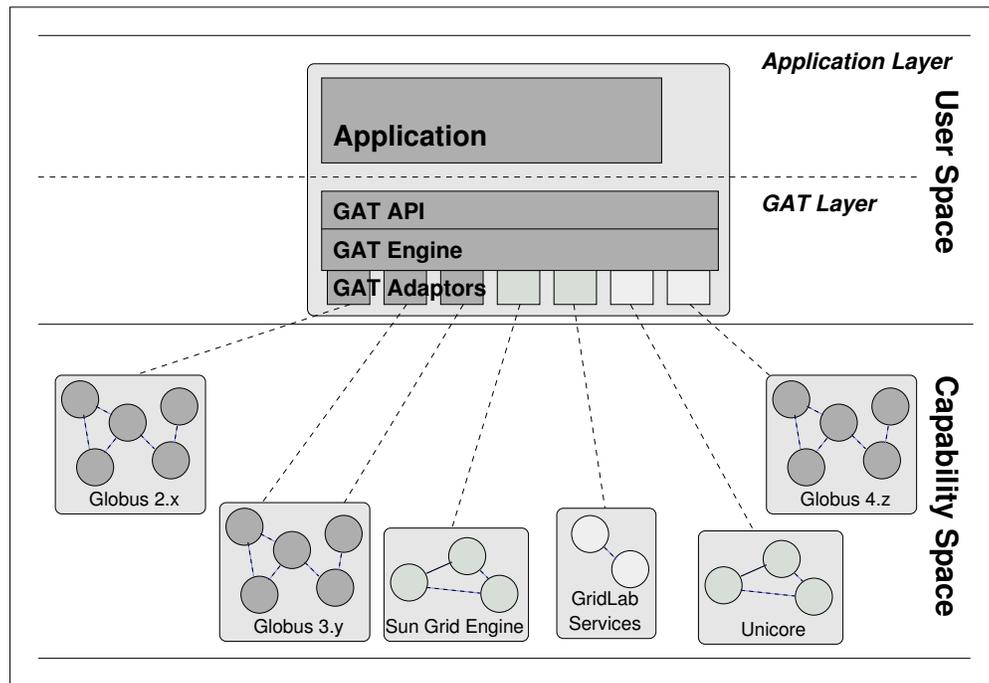

Figure 2: The GAT framework software architecture.

The GAT API has been designed using an object-based approach, trying to avoid dependencies on any specific language binding [3]. However, for each supported application programming language, a complete GAT layer has to be provided, consisting of API, engine, and adaptors. In this section, we briefly describe the current GAT implementations. Besides, we are planning to add further languages, like Fortran 90, Perl, command line shells, etc.

4.1 C Implementation

The currently most advanced version of the GAT is implemented in the C programming language, mainly for portability reasons. The default installation of the C GAT engine includes a full set of adaptors to machine-local resources. Additionally, many adaptors to Globus V2.4, Globus V3.2 and to different Gridlab services are available. Not only the GAT engine, which exposes the C-interface to the GAT API (see Section 3), but also the corresponding adaptors are written in C.

The C GAT engine consists of three logical parts. *The GAT API functions* map calls made by the application to the corresponding, adaptor-provided functionality. *The adaptor management subsystem* is responsible for loading available adaptors, managing their lifetime, and maintaining a capability registry that allows a GAT API subsystem to select the right adaptor. Each loaded adaptor registers its capabilities (groups of related functionality) with the capability registry. Every capability has a set of meta data attached (so-called "preferences"), allowing further control within the adaptor selection. *The utility objects and functions* provide utility functions for data structures (e.g., for lists and tables), as well as error handling and reporting.

The GAT engine exposes two different interfaces, the GAT API exposed to the applicaton, and the CPI (the *Capability Provider Interface*) to the adaptors. The CPI mirrors most of the GAT API functionality. It is the key for enabling dynamic adaptor binding. Every adaptor is a runtime library, and gets dynamically loaded at application startup time, based on given configuration information. Since the GAT engine is able to load several adaptors providing the same CPI, it has to negotiate with the existing adaptors to select the right one to serve a given API call.

4.2   C++ Implementation

Based on the existing C implementation, providing GAT functionality for C++ applications is easy. In fact, our C++ GAT merely defines a truly object-oriented API. Its classes are merely a wrapper to the C GAT engine. Therefore, all the existing C adaptors may be reused. The C++ binding allows to write code in a very clean and concise way.

4.3   Python Implementation

For providing access to the GAT API from within a scripting language, we have implemented an object-oriented GAT interface in Python. This is useful for rapid prototyping, as in other scripting environments, where a dynamic language binding is a useful alternative. Like with C++, the Python GAT is implemented as a wrapper around the C implementation.

4.4   Java Implementation

We are currently implementing a Java GAT. Java has several properties making it attractive for Grid computing, notably it's "write-once, run anywhere" portability. Java code can run without recompilation on any Grid site that has a JVM installed. Like with C++, the Java GAT is truly object-oriented; Java's exception handling mechanism considerably simplifies error-handling code.

The structure of the Java GAT closely resembles that of the C GAT. There is the Java GAT API for applications, mirrored by a CPI for adaptors. The Java GAT engine implements the dispatching of API calls to the adaptors. Although it would be possible to re-use adaptors written for the C GAT via the Java Native Interface (JNI), we chose to pursue a pure Java implementation to retain Java's portability. This way, applications only need the GAT

11

library (a jar file) and the adaptors (also jar files). No recompilation or configuration is necessary. We currently have adaptors for local operations and for the GridLab services.

An interesting property of the Java GAT is that it uses late binding: the actual adaptor to execute a GAT operation is selected only when the operation is invoked, and not when the corresponding GAT object is created. Late binding is slightly more robust and flexible than static binding. For example, if a file is to be copied from site A to the sites B and C, different transfer mechanisms can be used for the transfer from A to B and for the transfer A to C. For example, the first transfer could use the GridLab Data-Movement service, while the second uses GridFTP. Late binding also simplifies adaptor writing. For example, a file adaptor might only support a highly optimized implementation of file.copy, but no file.delete operation. The Java GAT engine will automatically fall back to another adaptor that can do the delete. This feature is convenient, as many Grid services do not provide the complete functionality that is offered by the GAT API. The Java GAT will then automatically use multiple services to implement the functionality of a single GAT object.

5   Conclusion

While a variety of Grid technologies are currently maturing, applications are still lacking simple and high-level programming interfaces. In this paper, we have demonstrated a simple, Grid-enabled application and its required, Grid-related operations. We have shown how the GAT API can provide a suitable platform for such applications. The GAT API has been designed to be simple and independent of both programming languages and Grid middleware systems. We expect the API to evolve along with the progress being made with GGF's SAGA research group [6].

Currently, the GAT has been implemented for C, C++, Python, and Java. Further languages bindings are under way. Currently available adaptors use local resouces, Globus 2.4 or 3.2, Unicore, as well as GridLab's services for data management, resource brokering and information management. The GAT API and its implementations are open-source software. They can be downloaded from *www.gridlab.org*.

References

[1] 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 on High Performance Computing Applications*, 17(4):449–466, 2003.

[2] 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, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 2004. To appear.

[3] Kelly Davis, Tom Goodale, and Andre Merzky. GAT API Specification: Object Based. *EU Project GridLab, Deliverable D1.5*, June 2003. http://www.gridlab.org/WorkPackages/wp-1/Documents/Gridlab-1-GAS-0004.ObjectBasedAPISpecification.pdf.

[4] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[5] GGF. GridFTP Working Group, 2001. http://forge.gridforum.org/projects/gridftp-wg/.

[6] GGF. Simple API for Grid Applications Research Group, 2004. http://forge.gridforum.org/projects/saga-rg/.

[7] GridLab Project. The Replica Management Service, 2002. http://www.gridlab.org/data.