User-Friendly and Reliable Grid Computing Based on Imperfect Middleware

Rob V. van Nieuwpoort, Thilo Kielmann and Henri E. Bal
Vrije Universiteit Amsterdam
De Boelelaan 1081, 1081 HV
Amsterdam, The Netherlands
rob, kielmann, bal @cs.vu.nl
www.cs.vu.nl/ibis

ABSTRACT

Writing grid applications is hard. First, interfaces to existing grid middleware often are too low-level for application programmers who are domain experts rather than computer scientists. Second, grid APIs tend to evolve too quickly for applications to follow. Third, failures and configuration incompatibilities require applications to use different solutions to the same problem, depending on the actual sites in use.

This paper describes the Java Grid Application Toolkit (JavaGAT) that provides a *high-level*, *middleware-independent* and *site-independent* interface to the grid. The JavaGAT uses *nested exceptions* and *intelligent dispatching* of method invocations to handle errors and to automatically select suitable grid middleware implementations for requested operations. The JavaGAT's *adaptor writing framework* simplifies the implementation of interfaces to new middleware releases by combining nested exceptions and intelligent dispatching with rich default functionality. The many applications and middleware adaptors that have been provided by third-party developers indicate the viability of our approach.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.2.6 [**Software Engineering**]: Programming Environments

Keywords

grid computing, intelligent dispatching, nested exceptions

1. INTRODUCTION

Grid computing aims to integrate collections of heterogeneous resources across administrative boundaries into a single virtual system (a grid). In this paper, we identify and solve three problems that interfere with the widespread adoption of grid technology for production use today.

First, a large number of wildly varying grid middleware systems is currently being developed. The standardization landscape has not settled yet, and grid technology and application programming interfaces (API's) are still a topic of research. As of today, grid middleware and API's *change frequently* and are often *unstable* or *incomplete* [18]. Furthermore, different middlewares often offer *different functionality*.

Second, as grid computing still is a research area with many open questions, middleware implementations tend to focus on technical issues, and provide *low-level programming interfaces* as a result. For instance, the Globus toolkit [15], a widely used middleware platform, significantly changed its API over the last three mayor versions. However, all versions expose the underlying technology (respectively proprietary protocols, web-services and the Web Services Resource Framework (WSRF)). It is clear that this is interesting and necessary research, but for today's grid *users* these technical details are irrelevant and exposing them is counter productive. Much like home appliance users are not interested in the details of electrical power generation, grid application programmers are typically not interested in the technical issues behind the grid, and want to use the grid for production systems today, using a high-level interface.

Third, *heterogeneity* in processors, the operating system, constantly changing and evolving middleware, and the fact that different grid sites use different grid middleware make it extremely difficult to develop and deploy *portable* grid applications.

This paper deals with the Java Grid Application Toolkit (JavaGAT), which solves the three aforementioned problems. The low-level problem is addressed by implementing the GAT [10] specification, which is a *high-level* API that aims to facilitate development of complex grid applications. The GAT specification is language independent, and implementations for C, C++, Python and Java exist. The GAT API is object oriented and offers high-level primitives for access to the grid, *independent of the grid middleware* that implements this functionality. The JavaGAT is the Java reference implementation of the GAT API. The problem of heterogeneous processors and operating systems is solved because we use virtualization techniques. In this case, we exploit the fact that Java uses a virtual machine based approach.

The solutions for the imperfect and evolving middleware and for the heterogeneity problems are discussed below. JavaGAT integrates multiple grid middleware systems with different and incomplete functionality into a single, consistent system, using a technique called *intelligent dispatching*. This technique dynamically forwards (dispatches) application calls on the JavaGAT API to one or more grid middlewares that implement the requested functionality. The selection process is done at runtime, and uses policies and heuristics to automatically select the best available middleware, enhancing portability. If a grid operation fails, the intelligent dispatching feature will automatically select and dispatch the API call to an alternative grid middleware. This process continues until

SC07 November 10-16, 2007, Reno, Nevada, USA

a grid middleware successfully performs the requested operation, achieving transparent fault tolerance. The JavaGAT defines special *nested exceptions* that contain the reason for failure for each individual middleware. When no grid middleware can perform the requested operation, the JavaGAT throws the nested exception to facilitate reasoning about errors and debugging, improving application reliability. Equally important, JavaGAT defines a framework for developing grid middleware bindings. The framework contains a large collection of generic code, significantly simplifying this process.

It is important to recognize that the middleware developers are as important as the end users: without robust bindings to middleware, a grid API is useless. Because of the sheer number of different middleware systems and their constantly changing APIs, we cannot provide access to all middleware systems ourselves. Furthermore, because writing middleware bindings for JavaGAT is straightforward, middleware developers achieve the freedom to experiment with different architectural designs and new techniques. This can now be done without interfering with the application programmers, as the top-level API (the JavaGAT API) remains fixed.

The contributions of this paper are as follows:

- We introduce a novel technique called *intelligent dispatching* that allows to integrate the heterogeneous and incomplete functionality offered by current grid middlewares into a simple and consistent API. Intelligent dispatching solves portability problems, both of the underlying computing platform (OS, library versions, architecture) and the different grid middleware systems and versions. Intelligent dispatching can even provide transparent fault tolerance if operations are not present in a middleware system, or if they fail.

- We define special *nested exceptions*, which can contain multiple inner exceptions describing the reason why grid operations failed. Nested exceptions allow applications to deal with and reason about errors, even when intelligent dispatching tries multiple grid middlewares subsequently.

- We define a powerful interface and framework that can be used by grid middleware developers to quickly and efficiently implement GAT bindings to their middleware system, without unnecessary duplication of code.

- By offering solutions for the heterogeneity problems, while providing high-level programming interfaces, we achieve functionality that grid users urgently need today, despite the (flaws of) evolving middleware, bringing production use of the grid a step closer.

The remainder of this paper is structured as follows. In the Section 2, we describe the API and global structure of the JavaGAT. We use examples to demonstrate the GAT API and to explain why intelligent dispatching is useful. Section 3 describes the nested exceptions. The intelligent dispatching technique is described in detail in Section 4. Section 5 describes the adaptor writing framework that facilitates the implementation of new middleware bindings. We evaluate our approach and show some experimental results in Section 6. Finally, we discuss related work in Section 7, and conclude in Section 8.

2. THE JAVAGAT

The JavaGAT currently is the most advanced implementation of the GAT API [10], which was defined in the EU-funded GridLab project [4]. Even though GridLab was finished in 2005, the user base of JavaGAT is still increasing, and we actively continued developing JavaGAT in the context of the Ibis project [5], and the Virtual Labs for E-Science (Vl-e) [9] project, and will continue to do so in the years to come. Both projects involve grid application programmers who need to deploy and use their applications on the grid today. It is important to note that grid application programmers usually are *not* computer scientists. Our users include physicists, chemists, astrophysicists, bioinformatics, medical researchers, etc. They are typically not interested in the grid middleware and the technology that is used underneath (web-services, WSRF, etc). Furthermore, users generally have the requirement to use several different externally-managed compute clusters simultaneously for production work (e.g., since they have bought cycles there). These clusters have different architectures and use different grid middleware systems.

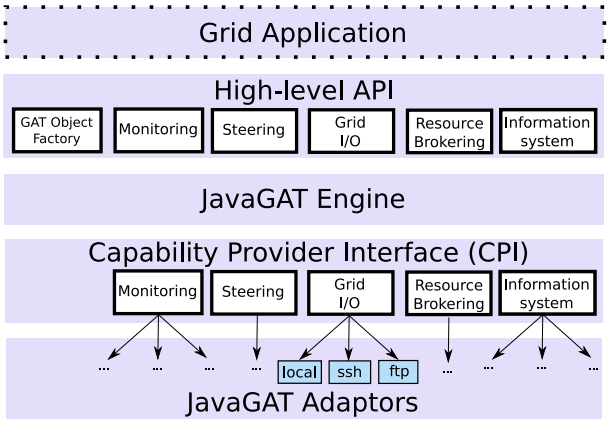

Figure 1: The structure of the JavaGAT implementation.

Since the initial JavaGAT implementation we have taken advantage of the user feedback, and identified and resolved many issues that were present in the original GAT specification. As a result, the JavaGAT implementation drifted away from the specification. We found that users deem integration with existing Java packages and tools extremely important. Sometimes this required API changes that deviate from the specification. For instance, the remote file access interface of the JavaGAT differs from the GAT specification [10], and is compatible to Java's standard file API, because that is what programmers are familiar with. In other areas, we extended JavaGAT beyond the scope of the GAT specification. For example, the GAT specification only specifies access to files, and not to directories. In JavaGAT, users can use directories in all places where they can use files (e.g., to stage in a directory with all its subdirectories before a job is started). JavaGAT also adds an application steering API, while the original specification only deals with monitoring.

2.1 The Global Structure of the JavaGAT

The global structure of the JavaGAT is shown in Figure 1. The system consists of four layers. The top layer is the high-level user API. The second layer is the JavaGAT engine, wich is responsible for delegating the API calls to the correct middleware. Because JavaGAT has to support multiple grid middlewares, we use a "plug-in" architecture. The third layer of the JavaGAT is the interface that is used by these plug-ins. We call this the Capability Provider Interface (CPI). The bottom layer consists of the plug-ins, called *adaptors* in this context. The adaptors contain code that binds to a specific middleware platform.

The GAT API is object oriented: the grid functionality is exposed through GAT objects, such as `File`, `Job` and `ResourceBroker`. Grid applications can create GAT objects with the GAT Object factory (the leftmost box in the API layer of Figure 1). The API packages provide support for monitoring (both for the grid itself and for applications), steering of applications, grid I/O (e.g., remote file access), resource brokering and job submission, and an information system to store application-specific data. Each GAT API uses a set of Java interfaces to define its functionality. The only exception is the GAT I/O API, which does contain a set of interfaces, but also provides an additional set of classes that extend the classes in the standard *java.io* package. This way, users of the JavaGAT can use remote files with the same classes they are familiar with for local files. Our users indicate that this is a tremendous advantage; it makes it trivial to grid-enable the I/O part of an existing Java application.

One of the central objects in the JavaGAT API is the GATContext. It contains handles to security information. The user can specify security information (credentials, passwords, etc) by using SecurityContexts, which are in turn stored in the GATContext. An application can use more than one GATContext, and can restrict access that adaptors have to security information. As we will explain in more detail in Section 4.2, the GATContext also functions as a container for engine and adaptor state.

Another central idea in GAT is the concept of *preferences*. The Preferences class contains key-value pairs (both Strings) that express information that is passed on to the adaptors. For example, the preference (`"ftp.connection.passive", "true"`) instructs the FTP adaptor to use passive connections. Preferences are typically opaque to the engine, they are just forwarded to the adaptors. There is, however, a small set of preferences that is interpreted by the engine itself. An example of this is a preference to enforce the use of a particular adaptor for a GAT object. Preferences can be global, or local for a specific GAT Object. In the latter case, the application provides the preferences when the object is created.

Throughout the JavaGAT, Uniform Resource Identifiers (URIs) are widely used, especially for file access. URIs are a superset of URLs. The engine has some built-in knowledge of URIs, and can use them to select adaptors. The engine knows that a URI with an "ssh" scheme can only be used by adaptors that support the SSH protocol, for instance, and will thus not instantiate other adaptors. An important special scheme that GAT defines is the "any" scheme, which means that the engine is allowed to select any adaptor. For a file, for instance, this means that it can be transfered with any transfer protocol that works. As we will explain later, the JavaGAT tries to use an intelligent mechanism to select the best adaptor in these cases. The “any” scheme is the most widely used URI scheme in applications.

The capability provider interface (the third layer in Figure 1) contains a set of abstract Java classes that implement the GAT API interfaces. The CPI classes thus can contain generic code that is shared between the adaptors. JavaGAT makes extensive use of this feature, to facilitate the adaptor writing process. Because JavaGAT is human-oriented, we consider the adaptor interface as important as the API that is exported to the application. Because grid middleware functionality and APIs change frequently, it is important that it is as easy as possible to develop GAT adaptors for new middleware, or to modify the adaptor if the middleware changes. Due to the great number of middleware platforms, and their frequency of changing, we cannot write and maintain all adaptors ourselves. JavaGAT thus exports the GAT API to higher layers (typically a grid application), and provides a plug-in interface (the CPI) to the lower layers (i.e., the adaptors). The engine is responsible for routing calls between the layers.

```
1  import org.gridlab.gat.*;
2  import org.gridlab.gat.io.File;
3
4  public class Copy {
5    public static void main(String[] args)
6        throws Exception {
7      GATContext context = new GATContext();
8      URI source = new URI(args[0]);
9      URI dest = new URI(args[1]);
10
11     // Create a GAT File object
12     File file = GAT.createFile(context, source);
13
14     file.copy(dest); // The actual file copy.
15
16     GAT.end(); // Shutdown the JavaGAT.
17   }
18 }
```

Figure 2: Actual code to copy (remote) files and directories with the JavaGAT.

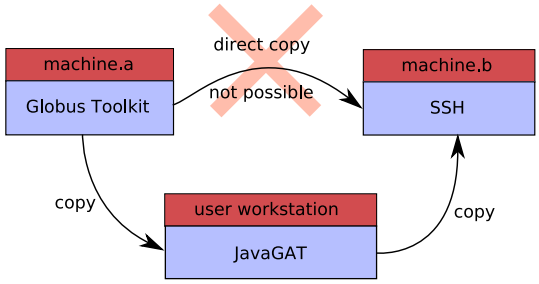

Figure 3: A third party copy using different middlewares.

2.2 An Example: Copying Files

We illustrate the concepts explained above with a JavaGAT program that can copy files and directories. The code shown in Figure 2 is *not* pseudo code, but actual code that implements this functionality. The example copies a file (or directory) that is passed as the first command-line argument to the destination that is the second argument. The examples below run this program on a user’s workstation (runGATApp is a small wrapper script that sets up the Java environment).

- A local copy:
  `runGATApp Copy /bin/echo foo`
- A remote to local copy with SSH:
  `runGATApp Copy ssh://machine.a//bin/echo foo`
- A remote to local copy with GridFTP:
  `runGATApp Copy gsiftp://machine.a//bin/echo foo`
- A remote to local copy, letting JavaGAT choose the “best” protocol, because the URI scheme is “any”:
  `runGATApp Copy any://machine.a//bin/echo foo`
- A third party copy: from a remote location to another remote location, while letting JavaGAT choose the best transfer protocol:
  ```
  runGATApp Copy any://machine.a//bin/echo
                 any://machine.b/foo
  ```

The example shows the flexibility and expressiveness of the GAT API. It also illustrates what we mean with the human-oriented approach: an application programmer wants to think in terms of file objects, not in terms of (web) services, WSRF, etc. Although desirable as a flexible technical infrastructure for middleware, such models are not suitable as an application programmer interface. JavaGAT provides a high-level API, and can even hide the grid middleware and transfer protocols altogether, as is shown by the use of the "any" scheme. In this case, the engine will automatically select the “best” transfer mechanism, and will even retry with other protocols if the best mechanism fails, until one is found that works. What we mean with the “best mechanism”, and how the engine selects it will be described in Section 4. The JavaGAT can automatically perform copies between two remote sites that use different grid middlewares, even if a direct copy is not possible. An example of this is shown in Figure 3.

2.3 Designing for Portability, Fault-Tolerance, and Middleware Evolution

The examples above demonstrate that it is important to deal with portability, changing and incomplete middleware and fault tolerance when designing a high-level grid API. We discuss each of these areas in turn below.

Portability

An important feature to obtain a better user experience is portability. There are two aspects to portability. The first aspect is that it is important that development and deployment of the application is as easy as possible. With traditional languages, applications have to be recompiled for each platform. Different operating systems and different (versions of) libraries make this extremely error prone and time consuming. We solve this problem by using a Java-based approach. Java has several properties making it attractive for Grid computing, notably its "write-once, run anywhere" portability. Java code can run without recompilation on any Grid site that has a Java virtual machine (JVM) installed.

The second portability aspect is that of heterogeneity of the grid middleware. If the sites where a grid application is deployed use different (versions of) middleware, a user-friendly grid API automatically selects a working middleware (e.g., see Figure 3). A special mechanism is needed that forwards API calls to a specific middleware at run time, as the correct middleware used is not known at compile time. The forwarding of methods is generally called *dispatching* [14].

We call the approach we take with the JavaGAT *intelligent dispatching* for two reasons. First, the JavaGAT uses several techniques to automatically select the “best” middleware at runtime (See Section 4). Second, the actual middleware that has to execute a specific operation is selected only whenever an operation is invoked, and not when the corresponding API object is created. Thus, JavaGAT uses function-level binding instead of object-level binding. A single API object can thus use multiple middlewares. Intelligent dispatching is more robust and flexible, and thus more user-friendly than static dispatching. For example, if a file is to be copied from a workstation to machine A and to machine B, the transfer from the workstation to site A can use a different transfer mechanism than the transfer from the workstation to site B. For example, the first transfer may use SSH, while the second uses GridFTP. When static dispatching is used, the programmer needs to know this fact in advance.

With static dispatching, an application has to create *two different* source file objects, one explicitly with the SSH protocol, and one explicitly with the GridFTP protocol. As a result, hiding the details of the underlying middleware is effectively impossible, because the application has to explicitly specify the adaptors that have to be used. With the techniques introduced in this paper, users can be oblivious to these technical details. The intelligent dispatching feature seperates the JavaGAT from other similar projects. No other framework can do this, including GAT implementations for other languages. We will discuss intelligent dispatching in more detail in Section 4.

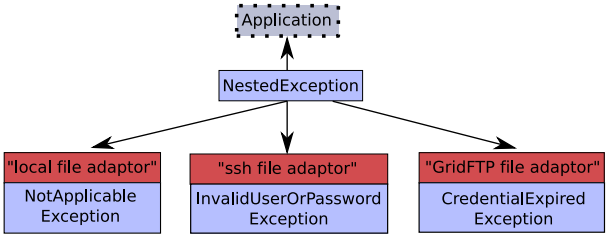

Figure 4: A NestedException.

Fault Tolerance

Grid APIs should also deal with faults. We demonstrate that a well-designed dispatching mechanism can provide this by subsequently executing operations with different grid middlewares, until the operations succeeds (see Section 4). Obviously, this can only work if multiple middlewares are applicable and implement the requested operation. If no middleware can successfully perform the requested operation, failure has to be reported to the grid application.

In Section 3 we show that this can be done in the context of intelligent dispatching by using special nested exceptions. If all available middlewares failed, the JavaGAT throws a nested exception, containing a list of middlewares that were tried, and the reason for each failure. The application can use the nested exception to investigate and deal with the error.

The JavaGAT can tolerate both the *unavailability* and the *failure* of an implementation. However, it cannot deal with transient failures (failures that are not detected by the middleware itself) and operations that do not terminate.

Dealing with Changing and Incomplete Middleware

Intelligent dispatching significantly simplifies adaptor writing. For example, a file adaptor might only support a highly optimized implementation of *file.copy*, but no *file.delete* operation. In that case, the JavaGAT engine will automatically fall back to another adaptor that does implement the delete operation (if such an adaptor is available). This feature is of key importance, because many grid services do not provide the *complete* functionality that the GAT API offers. The JavaGAT exploits intelligent dispatching to automatically use multiple services to implement the functionality of a single GAT object.

3. NESTED EXCEPTIONS

Discussions with our users led to the insight that reliable grid applications need to reason about errors. For instance, a reliable system may retry certain operations if a remote service is down or unreachable, as it may be restarted, or the cause could be a temporary network failure. However, if the problem is caused by an invalid user credential, retrying is useless as this situation will not rectify itself. Instead, the user must be informed. Therefore, an application must be able to differentiate between different errors, and well-defined exception types are important.

Nested exceptions are thrown by the JavaGAT engine when *all* adaptors failed. Standard Java exceptions can contain another exception that caused it. JavaGAT generalizes this idea: NestedExceptions can have more than one cause. In this case, there is one cause for each adaptor that failed. NestedExceptions also contain the name of the adaptor that threw each inner exception. Further, NestedExceptions have methods to iterate over the inner exceptions, methods to print meaningful strack traces, etc. An example of a NestedException is shown in Figure 4. The nested exceptions and the adaptors work together to produce user-intelligible error messages. If a nested exception is printed, it produces one line per adaptor that failed. The adaptors are responsible for producing meaningful error messages. The nested exception in Figure 4 would produce the following output:

```
--- START OF NESTED EXCEPTION ---
LocalFileAdaptor failed: Cannot copy to remote destination
SshFileAdaptor failed: Invalid user name or password
GridFTPFileAdaptor failed: Credential expired
--- END OF NESTED EXCEPTION ---
```

A stack trace of a nested exception is handled similarly: the stack traces of all adaptors are subsequently printed on the screen.

The original GAT specification does not define any exceptions, let alone nested exceptions, as the specification is language-neutral, and some languages do not have exceptions. However, we found that this feature makes it substantially easier for application programmers to debug their applications.

A key observation concerning intelligent dispatching is that the JavaGAT engine always catches *all* exceptions and errors when it invokes methods on adaptors. The rationale behind this is the following. JavaGAT is a complex piece of software, but it also uses and depends on many external libraries which are equally complex themselves. Especially the adaptors which bind to grid middleware usually have many dependencies. All complex software contains errors, and we assume that the libraries do as well. In practice, we found that this is a realistic assumption, in particular for rapidly changing and evolving grid middleware. We regularly experience errors and crashes in grid middleware libraries, especially in corner cases that are not frequently tested. For example, we found that a certain library crashes when we request a listing of a directory that contains more that a certain number of files. JavaGAT works around the instability problems by assuming that *things can and will go wrong* in complex distributed systems, and that libraries contain bugs. Typical examples of errors that can occur are null-pointer dereferences and arrays that are indexed outside their bounds. As long as a library does not hang indefinitely, JavaGAT's intelligent dispatching mechanism will effectively provide fault tolerance, and will select another adaptor in case of errors. This process continues until an adaptor successfully performs the requested operation, or until all adaptors have failed. We assume that adaptors do not fail silently without throwing an exception.

4. INTELLIGENT DISPATCHING

In this section we will explain intelligent dispatching in detail. The mechanism consists of two steps. First, when a GAT object is created, an initial filtering is done, and the adaptors that can implement the object are instantiated. Second, when a method is invoked on the object created in the previous step, the method must be dispatched to one or more adaptors.

When the JavaGAT engine is initialized, it must load the adaptors that implement the actual grid functionality. To locate the adaptors, JavaGAT uses an approach that is similar to Java's classpath mechanism, because it is easy to use and familiar to Java programmers.

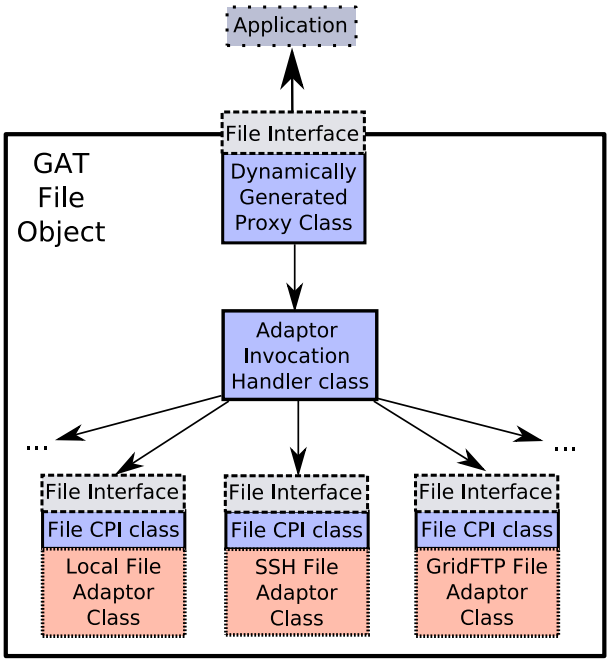

Figure 5: The implementation of a GAT object.

JavaGAT adaptors are distributed in Java *jar* files, which contain compiled Java code. The manifest in each jar file describes which classes implement which JavaGAT API. Jar files can be signed with a digital signature, and signed jar files could have more privileges than normal jar files. During startup, the engine dynamically loads the code in the jar file into the running application.

4.1 GAT Object Creation and Adaptor Instantiation

When an application creates a GAT object using the GAT object factory, the engine must determine which of the available adaptors are applicable for the requested object, and it has to instantiate them. In most cases, multiple adaptors will apply. Therefore, JavaGAT creates a proxy class that implements the interface of the requested object, but forwards the application's calls to one or more adaptors. We describe the initial adaptor selection process and the creation of the GAT object in this section. The next section explains how the proxy class dynamically dispatches the invocations on the GAT object to the adaptors. Figure 5 shows the internal structure of a GAT object, in this case a *File*. Although trivial for the user, internally GAT object creation is a complex process that consists of several separate steps, which are described below, using Figure 5 as a reference.

1. Before creating any adaptors, the JavaGAT engine creates a new nested exception (See section 3). If an adaptor is not selected for some reason, an exception describing the reason is added to the nested exception. The nested exception can in turn be investigated by the application for reliability (See Section 3), testing and debugging purposes.

2. When a GAT object is created, the application can specify additional local preferences. These are merged into a single set with the global preferences that are attached to the GATContext (typically, applications use only one GATContext). Local preferences override global preferences. The resulting

set is matched against the preferences that are specified in the manifest of the adaptor jar file. The merged list of application preferences must contain all preferences required by the adaptor. If not all required preferences are matched, the engine will not select the adaptors in the jar file for the requested GAT object. In this case, the engine will create a special *AdaptorNotSelectedException*, which is defined in the JavaGAT API, so the application can determine this fact. Currently, most adaptors have an empty list of required preferences, so they can always be selected.

The JavaGAT engine also defines special preferences that allow an application to enforce the use of a specific adaptor. Such a preference could for instance be "File.adaptor.name", with the value set to "ssh". Effectively, specifying preferences like this disable JavaGAT's intelligent dispatching feature: only one adaptor is allowed as an implementation of a certain object. This mechanism can be useful for testing and debugging of applications and adaptors alike.

3. Instantiate all remaining adaptors that were not filtered out in the previous step. The JavaGAT engine uses Java's reflection mechanism to lookup and invoke the constructor for the adaptors that remain after the filtering process.

4. Filter out adaptors where the constructor threw an exception, and add this exception to the nested exception. There can be many reasons for an adaptor to throw an exception, and the JavaGAT API defines several exception types for some common reasons. A frequently occurring case, for instance, is that the adaptor is not applicable. An example of this is when the user creates a file with a URI that contains a hostname, which refers to a remote machine. The local file adaptor is obviously not applicable in this case, as it can only access files on the local disk. Another example is the creation of a file object where the URI contains a specific scheme component, like "sftp". Only adaptors that understand this scheme are applicable. For the others, JavaGAT throws the *AdaptorNotApplicableException*, which is defined in the JavaGAT API. There can be multiple adaptors that understand a particular scheme. For instance, the default JavaGAT distribution currently contains three different SFTP adaptors that use different libraries.

   Another common reason why an adaptor cannot be initialized is that the required middleware is not installed, or that a required service is not running. Finally, there can be a problem with the security context that is provided, a user name or password can be invalid, or a credential could be expired. JavaGAT also defines exception types for these cases.

5. If no adaptors remain after the previous steps, there is no adaptor that can implement the requested object. In this case, the JavaGAT engine throws the nested exception that contains the reason for the adaptors to fail. The application or user can now investigate the reasons for failure, and take appropriate action. We found that, in practice, the cause is often a misconfiguration or a middleware failure. For instance, in some cases the user specifies a certain scheme in a URI (SSH, FTP) or selects a particular adaptor, as he knows which middleware is installed on a particular resource. This effectively disables JavaGAT's intelligent dispatching feature. If the selected adaptor is not applicable because the service is down or the user credential is not initialized or expired (the most common error), JavaGAT will notify the user of this fact by throwing the nested exception.

6. When, after the selection process, multiple applicable adaptors remain, JavaGAT will sort them. This is mostly done for performance reasons. For instance, file transfers are typically faster with GridFTP than with SSH, as GridFTP can use parallel data streams. Another reason for a particular order could be security: the most secure transfer protocol could be tried first (much like SSH does). To specify the order in which adaptors are tried, the JavaGAT user or application can specify an *adaptor ordering policy*. The JavaGAT API defines the AdaptorOrderingPolicy class, and an easy way of creating user-defined policies. Finally, JavaGAT provides a default policy that aims to use a reasonable ordering of all adaptors that are distributed with the standard JavaGAT distribution. Which policy should be used can be specified by the application code, and can be overridden by the user (by using a command-line option that sets a Java system property). The idea of adaptor ordering policies and the mechanism that JavaGAT uses to implement them are not specified in the GAT specification, but is a novel feature in the JavaGAT. We found that it is a useful feature for advanced users. Typically, the default policy supplied by the JavaGAT is sufficient, so less advanced users are not bothered with this. The instantiated adaptors are shown at the bottom of Figure 5.

7. Create a proxy that will forward calls to the adaptors. Because the application's method invocation on a GAT object must be forwarded to one or more adaptors, the engine has to create an object that implements the API of the requested object, and that intercepts and forwards the calls to the adaptors. We use the standard *java.lang.reflect.Proxy* mechanism to do this. This mechanism dynamically generates a proxy class that implements a set of requested interfaces at runtime. In our case, the interface that must be implemented is the JavaGAT interface for the requested GAT object. The generated proxy class implements the requested interface, and will later be returned to the application as the GAT object. The proxy object is the top-level object in Figure 5. The proxy will forward the application's invocations on it's interface to an *invocation hander*.

   The engine creates an invocation handler object, called *AdaptorInvocationHandler*, that has a reference to the ordered list of selected adaptors. When an application invokes a method on a GAT object, the method is actually invoked on the generated proxy. The name of the method and the parameters of the invocation are provided to the invocation handler as parameters. The invocation handler can now use Java's reflection mechanism to invoke the requested method on the adaptors. We will explain this process in more detail in the next section. In Figure 5, the invocation handler is represented by the box in the center.

The mechanism described above is a novel feature of the JavaGAT. All other GAT implementations and other high-level grid middlewares use static dispatching instead of intelligent dispatching. With static dispatching, only one user-specified adaptor is instantiated when a grid API object is created. JavaGAT, in contrast, selects adaptors dynamically, and new adaptors that were not available at the time the application was developed are a part of this process. Although the adaptor selection process is quite complex, it is transparent for the application. Nevertheless, advanced users or applications can still influence the process using ordering policies, preferences, and URI schemes.

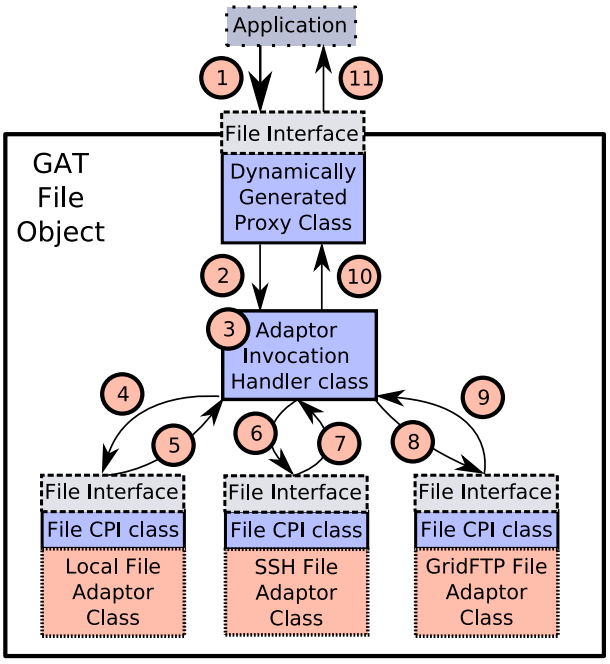

Figure 6: Invoking a method on a GAT object.

4.2 Intelligent Dispatching of GAT Object Methods

We will explain the inner workings of the GAT engine when a method is invoked on a GAT object by using an example. Figure 6 shows the same GAT File object we used in the previous section, but now annotated with the steps that are taken when an application invokes a method on the object. We will now describe each of the steps in more detail. The numbers inside the circles in Figure 6 correspond with the steps below.

1. The application invokes a method on the GAT object. For sake of the example, let us assume that the method invoked is *copy*, with the destination parameter `any://machineA.domain.com/testFile`.

2. The dynamically generated Proxy class (see step 7 in Section 4.1) is the actual Java object that implements the GAT object API, in this case the File interface. The Proxy forwards a description of the invocation to the AdaptorInvocationHandler class.

3. The adaptor invocation handler first creates a NestedException object, which acts as a container for possible exceptions the adaptors might throw. Next, the handler employs Java's reflection mechanism, and uses the description of the method invocation that is provided by the proxy to forward the call to the adaptors, until one succeeds (i.e., does not throw an exception). In the example in Figure 6, there are three adaptors available: a local file adaptor, an adaptor for SSH, and one for GridFTP.

4. The invocation handler invokes the requested method, *copy* in our case, on the first adaptor. The handler can invoke the copy method, because the adaptor extends the FileCPI class, which in turn implements the File API. The FileCPI class contains implementations *for all methods* in the File interface, although most are "dummy" implementations that just throw a *NotImplementedException*. The file adaptors will override some, but not necessarily all methods with their own versions containing an actual implementation. Nevertheless, the adaptor invocation handler can always invoke the requested method, as it is always implemented, either in the CPI class or in the adaptor itself. The CPI is described in more detail in Section 5.

5. The local adaptor throws an *AdaptorNotApplicableException*, as it determines that the destination URI contains a remote hostname. The local adaptor cannot copy files to remote hosts, and is therefore not applicable.

6. The adaptorInvocationHandler catches the exception, and adds it to the nested exception it created earlier, in step 3. As explained in Section 3, the adaptor invocation handler always catches *all* exceptions and errors. Because the first adaptor failed, the handler will now invoke the method on the second adaptor (i.e., the SSH adaptor).

7. The SSH adaptor throws an exception, because the SSH daemon on the remote site is not running.

8. This exception is caught by the invocation handler, and added to the nested exception. Next, the *copy* method of the third adaptor is invoked.

9. The GridFTP adaptor copies the file and returns normally.

10. The adaptor invocation handler returns the result value (if any) to the Proxy class.

11. The Proxy class forwards the result to the application.

The adaptor invocation handler invokes the adaptors in the order that is specified in the adaptor ordering policy, as described in the previous section. However, for performance reasons, we implemented an additional optimization. If an adaptor successfully implemented an operation, the adaptor is moved to the front of the ordered adaptor list. This is done during step 10 in the example above. If the applications performs subsequent invocations on the same object, the adaptor that succeeded last time will be tried first. This optimization is effective when adaptors fail after performing expensive operations, such as web service invocations. In some cases, adaptors fail after a timeout. Using this optimization, this overhead only occurs once, during the first invocation on a GAT object.

5. THE ADAPTOR WRITING FRAMEWORK

In this section, we describe the framework that the JavaGAT provides to facilitate adaptor writing. The framework consists of two parts. The first part is the abstract Capability Providers Interface (CPI), which is the interface used by the adaptors. The second part is a set of generic high-level adaptors that provide additional functionality on top of the real adaptors that bind to the grid middleware. The adaptors are an important part of the JavaGAT system. They implement the actual functionality that is defined by the JavaGAT API. Adaptors typically consist of "glue code" that translates GAT API requests into one or more grid middleware operations.

Because we consider JavaGAT adaptor writers equally important as the grid application programmers, it is imperative that adaptor writing is as easy as possible. Many different middleware systems exist, and JavaGAT has to support different versions of those systems *simultaneously*. This becomes even more important because

the grid middleware changes so frequently. Adaptors therefore also have to be modified frequently, and new adaptors have to be written for new middleware. To facilitate the adaptor implementation, JavaGAT provides as as much generic code in the CPI classes as possible.

The CPI classes implement the interfaces specified in the JavaGAT API, and provide code for *all methods* that are defined in the API. Some methods contain code that actually implements the required functionality. For instance, simple getter and setter methods are typically implemented at the CPI level. All other method implementations just trow *NotImplementedExceptions*. Methods can be overridden by the adaptors that extend the CPI classes as required. If methods that do not contain an actual implementation are not overridden, JavaGAT's intelligent dispatching mechanism will catch the *NotImplementedExceptions*, and will automatically select another adaptor.

JavaGAT Adaptors have to extend the CPI classes. Since the CPI classes implement the corresponding JavaGAT API interfaces, the adaptors automatically inherit this, and thus implement the GAT API. All methods are already implemented in the CPI classes, so adaptors do not have to implement the complete API. In fact, most adaptors only implement a small fraction of the full API. JavaGAT's generic code in the CPI classes adds additional functionality, and the intelligent dispatching mechanism fills in the remaining blanks, by using other adaptors to implement the missing functionality.

Adaptors can have internal state, which can be saved between invocations. The GridFTP adaptor, for instance, caches connections to the GridFTP servers between file operations. In some cases, adaptors can return adaptor-specific handles (e.g., a ResourceBroker can return a Job handle). In such cases, adaptors are responsible for recognizing the handles that they returned, because the intelligent dispatching mechanism could try different adaptors with the handle. If an adaptor is called with an adaptor-specific handle that does not belong to it, an error is thrown, allowing the intelligent dispatching mechanism to select another adaptor. Eventually, the correct adaptor will be executed.

The JavaGAT implementation comes with a full set of local adaptors that implement the complete API on a local system. This is useful as an example for adaptor writers. Furthermore, it gives application developers the possibility to test their application on their local workstation or laptop, before it is deployed on the grid. Since the debugging of distributed applications is notoriously difficult, this is an important feature. Thanks to JavaGAT's intelligent dispatching feature and the fact that the engine dynamically loads the adaptors into the running program, deployment on the grid is almost trivial. No recompilation or reconfiguration is needed.

5.1 Grid I/O

One of the most important and widely used APIs of the GAT is the Grid I/O API. We currently have many adaptors that implement it. However, many middlewares only provide a subset of the functionality that the JavaGAT API defines (e.g., many protocols do not support third party copies, or copies of directories). Therefore, we invested a lot of effort to provide us much functionality as possible in a generic way at the CPI level. Below, we list some of the functionality the JavaGAT grid I/O CPI implements. All generically defined code can be overridden by an adaptor if a better or more efficient implementation is available.

- All code that deals with the handling of URIs is generic, and is implemented in the CPI.
- A file move operation can be implemented by a copy followed by a delete of the original file.
- The file API provides several ways of listing files in a directory, called *list* and *listFiles*. The first returns an array of Strings that contain the file names, while the latter returns an array of *File* objects. The *File* CPI provides generic implementations of each of the two methods on top of the other method. Therefore, an adaptor only has to implement either *list* or *listFiles*, but not both. Most adaptors simply implement only the *list* method.
- Java's *File* class, and the compatible JavaGAT *File* provide methods to filter directory listings, using *FileFilters* and *FileNameFilters*. The code to do the filtering is implemented in the *File* CPI class.
- Operations that deal with directories can be implemented in a generic way. The *File* CPI, for instance, provides a mechanism to recursively delete a directory. JavaGAT implements this by creating GAT *File* objects for each entry in a directory, using the *listFiles* method described above. The created *File* objects are normal GAT objects, and thus support intelligent dispatching. The CPI can now simple invoke a normal delete operation on each of the files. Any adaptor can be used to perform the actual delete. Similarly, the JavaGAT CPI provides implementations for copying and moving directories, and for the *mkdirs* operation that creates a sequence of directories in one call.

5.2 Resource Management

With the resource management API, applications can, amongst others, submit jobs to the grid. Before the job is submitted, a set of input and application binary files can be pre-staged (copied to the site where the job will run), while afterwards the files that are produced by the application run can be post-staged (copied back to the submission site). Many grid middlewares do provide this functionality in some way, but in many cases the support is not flexible enough for our needs. Sometimes only files can be staged in, and not directories. In other cases, the middleware does not create a special sandbox directory, so when multiple jobs are submitted to the same machine, they may overwrite each others files. Also, many systems assume that the files to be staged in reside at the submission site, while this is not always the case. Finally, middlewares almost always assume that the compute elements share a filesystem (e.g., using NFS) with a frontend system where the grid scheduler is running, and staging files to a local disk of a compute element is not possible. Nevertheless, this can be imperative for good performance. JavaGAT implements all aforementioned features in a generic way in the resource management CPI.

JavaGAT provides a mechanism that creates a sandbox on remote machines. All application input files and directories are copied to the sandbox directory before the job is started. The copy operations are done using JavaGAT's file API, and thus automatically use intelligent dispatching, and will exploit the functionality of all available file adaptors. Because third party copies are supported by the File API, source files can be located anywhere in the grid. They could also be replicated. When the Job is finished, the output files are copied back using the same mechanism.

5.3 Security

The security CPI provides some generic functionality to cache security information such as passwords and credentials. Furthermore, it provides support to retrieve credentials from a MyProxy credential management service [6]. Finally, the JavaGAT security

CPI provides a mechanism to restrict the availability of security information to certain adaptors or remote machines. This is useful, because some adaptors might not be trusted with certain security information. Likewise, passwords should be restricted to a set of trusted hosts in the grid. This mechanism is not present in the GAT specification. However, some of our users (e.g., those who deal with medical data) require this feature.

5.4 Information Services

With the information service API, GAT objects can be persistently stored. Objects that support persistency are for instance files, jobs and communication endpoints. To ensure that different GAT implementations and different language bindings can use each others serialized objects, Advertisable objects are marshaled to XML. The JavaGAT engine and the CPI implement the marshaling in a generic way, using the Castor library [3]. For GAT objects that do not have adaptor-specific state, the engine implements the marshaling in a manner that is completely transparently for the adaptors. This is often the case for files, where the only state typically is the URI that contains the file's address. Adaptors are responsible of marshaling their own internal state, if they have any. The JavaGAT engine offers convenient hooks to implement this.

5.5 Filling in the Blanks

We also provide some simple adaptors that add backup implementations for some functionality if no adaptor implements it. Adaptors like this are typically added as the last entry in the adaptor ordering policies, so they are only selected if all else fails. For example, we provide a File adaptor that only implements an inefficient third-party copy, and nothing else. This adaptor uses GAT files (and thus benefits from the intelligent dispatching) to implement the copy. First, the file or directory is copied from the source host to the local host. Next, another copy operation copies from the local host to the destination machine in a second step. Although this is not very efficient, we found that many JavaGAT users prefer a slow copy over no copy at all. This is especially true for batch jobs and production systems. A similar approach is taken for file streams. If no adaptor can do remote streaming, JavaGAT features an adaptor that copies a file to the local disk and streams it from there. This method of providing additional functionality using generic adaptors that in turn use other adaptors as building blocks is possible only because of the intelligent dispatching of grid operations.

5.6 Optimizing Special Cases

We can also exploit JavaGAT's intelligent dispatching features to increase application performance in special cases. This can be done by writing a specialized adaptor that implements some operation in a more efficient way. For example, we have an SSH File adaptor for JavaGAT that uses the Jsch library [1]. However, we found that the encryption of the channel is inefficient in this library. Although this is not a problem for most file operations, for big file copies this results in a performance problem. Therefore, we wrote a very simple command-line SSH adaptor that can copy files if a native command-line SSH client is installed. On most Unix systems this is the case, and native SSH clients are available for Windows as well. Next, we can use the adaptor ordering policy mechanism to make sure the JavaGAT engine tries the command-line SSH copy adaptor *before* the Jsch-based SSH adaptor. If no SSH client is installed, the command-line adaptor will fail, and the copy method of the Jscj-based SSH adaptor will be invoked. The command-line adaptor only overrides and implements the CPI's copy operation. For all other file methods, the original CPI methods that it throw *NotImplementedExceptions* are automatically inherited. If they are invoked, JavaGAT's intelligent dispatching mechanism will automatically select the Jsch-based SSH adaptor. In general, using this mechanism, we can add specialized adaptors in that increase performance in special cases, without breaking existing functionality.

Submission method	time (seconds)
native globus-job-run	11.1
CoG kit	11.8
JavaGAT	12.6

Table 1: Submission time of a trivial job to a local cluster.

6. EVALUATION

In this section we evaluate the overhead that is introduced by intelligent dispatching and the other techniques described in this paper. Furthermore, we discuss the successfulness of our approach, in terms of application programmer and middleware developer adoption of the JavaGAT.

6.1 Performance Evaluation

We argue that the overhead introduced by the techniques we describe here is insignificant, due to the typically high cost of grid operations. We now provide some experimental results that support this claim. We ran our experiments on a dual-CPU / dual-core 2.4 GHz AMD Opteron DP 280, with 8 GB of memory, and 10 Gigabit/s Ethernet.

First, we ran a simple job submission test, that submits the executable "/bin/hostname" to the local cluster, using the Globus toolkit 3.2 pre web services, and using SUN Grid Engine (SGE) as the local cluster scheduler. The "hostname" command prints the name of the compute element and only takes 12 milliseconds. No files are pre-staged or post-staged. This therefore is a trivial job, and the lower bound of job submission time. The measurements presented in Table 1 show that even the submission of a trivial job already takes 11 seconds to a local machine. If the destination is remote, or if the newer Globus Toolkit version 4 that is based on WSRF is used, the times become even higher (not shown). The overhead of the JavaGAT is only 800 milliseconds, and this includes the time for initializing the JVM, the JavaGAT engine, and loading 38 adaptors. For realistic jobs, this overhead is insignificant.

We also performed micro benchmarks of the intelligent dispatching mechanism. Our test creates one million GAT File objects, and measures how long this takes on average. Nine File adaptors are installed, of which seven are actually applicable in this case. We found that the creation of a new GAT object takes 1.3 milliseconds on average. This time includes the instantiation of seven GAT adaptors, and checking the provided security information for each adaptor (e.g, checking for a valid Globus proxy).

Subsequent invocations on the created GAT File object use JavaGAT's intelligent dispatching mechanism to forward the calls to the adaptors. We measured that the invocation of an operation on a GAT Object takes only 8 microseconds on our hardware. It is clear that the overhead of intelligent dispatching, which is in the microsecond range is insignificant for grid operations, which typically are in the second range.

6.2 Successfulness of our Approach

The successfulness of our approach is demonstrated by the large number of adaptors that was successfully implemented for the JavaGAT. For instance, for file access, JavaGAT has adaptors for local files, GridFTP, RFT, FTP, SSH, SFTP, HTTP, HTTPS and SMB/-

CIFS. For resource management, the JavaGAT supports local forking, GRMS, Globus GRAM, SSH, prun, PBS, Sun Grid Engine (SGE), ProActive, Integrade and Zorilla. Several of these adaptors were written by external groups, who confirm that the adaptor writing framework that the JavaGAT provides is extremely useful. This is also shown by the size of the adaptors. The SGE adaptor, for instance contains only 422 lines of Java code (including comments). The complete Globus GRAM adaptor needs only 1614 lines of code.

Furthermore, many applications were developed using the JavaGAT. For instance, medical people at the VUMC hospital in Amsterdam were able to write applications that analyze Magneto Encephalography (MEG) scans on several different clusters simultaneously [13]. Physicists at the AMOLF institute are using the JavaGAT to perform parallel Fourier transforms on mass spectrometry data [22], transparently streaming files with different protocols such as SFTP, SSH and GridFTP. A computational chemistry group has used the JavaGAT to implement file browsing and experiment data management on the grid. Computer scientists have used the JavaGAT as a back-end for work-flow submissions and portals [12, 2], and linguists are using the JavaGAT to implement analysis of specialist texts [8].

7. RELATED WORK

The GAT [10] is a language independent object-oriented specification. This paper describes the Java reference implementation of the GAT. However, other language bindings also exists, e.g., for C, C++ and Python. The most important difference between the JavaGAT and the other implementations of the specification is that the JavaGAT uses intelligent dispatching, while the other implementations use static dispatching. Furthermore, the JavaGAT has an additional steering interface, and an more user-friendly Java compatible file API.

The Java CoG Kit [11] uses an abstraction model to provide a grid execution framework. The API provided by the CoG kit only supports remote file access and job submission. An interesting feature of the CoG Kit is that it supports a form of *late binding*. The actual implementation of an API object is selected at run time, and not at compile time. However, the application must explicitly specify which middleware is used, whereas JavaGAT can do this selection process automatically. Moreover, a single adaptor (called provider in the Java CoG) is selected for an *entire object*. We have shown that the JavaGAT approach of using intelligent dispatching is more flexible (see Section 4). Intelligent dispatching has several key advantages, such as the ability to use different middlewares to implement a single object, better portability, and the support for fault tolerance.

An interesting new development is the standardization of the Simple API for Grid Applications (SAGA) [16] by the Open Grid Forum (OGF) [7]. Like GAT, SAGA is a high-level middleware-independent API for grid applications. The API for SAGA is largely based on the GAT and on the CoG kit abstraction model. A reference implementation for C++ already exists [17]. JavaGAT's intelligent dispatching mechanism heavily influenced the design and implementation of the C++ reference implementation, which currently already supports a simpler form of dynamic dispatching. Moreover, our group has received funding to implement the Java reference implementation for SAGA. We intend to reuse large parts of the JavaGAT engine, including the intelligent dispatching feature described in this paper.

DRMAA [20] is an API specification for job submission and control to distributed resource management systems, developed in the context of OGF [7]. DRMAA is object-oriented and middleware independent, like the GAT. However, DRMAA only deals with resource management, while the GAT also deals with other important issues, such as grid I/O, monitoring and information services. Finally, in contrast to JavaGAT, DRMAA uses static binding to a particular middleware. Nevertheless, DRMAA provides a useful and important abstraction, and JavaGAT has adaptors that use DRMAA to interface to cluster schedulers.

The GridRPC specification [19] defines a model and API for a remote procedure call mechanism for grid environments. GridRPC is specifically targeted for end-user applications, not middleware. Two reference implementations for GridRPC exist, Ninf-G [21] and NetSolve/GridSolve [23]. GridRPC uses static dispatching: when an object is bound to a server, all calls to that object will be performed at that particular server. We argue that intelligent dispatching is more flexible. An interesting framework for implementing dispatching mechanisms, called PolyD, is presented in [14]. JavaGAT uses a similar mechanism, but adds intelligence to it, letting the dispatching mechanism *automatically* select suitable adaptors.

8. CONCLUSIONS AND FUTURE WORK

We have introduced the JavaGAT: an environment providing an object-oriented, high-level, and middleware-independent API to the grid. We have described a novel technique, called *intelligent dispatching*, used by the JavaGAT to integrate the heterogeneous and incomplete functionality offered by current grid middlewares into a simple and consistent API. With intelligent dispatching, the JavaGAT can automatically select the best middleware for each individual grid operation. When a particular middleware fails, the JavaGAT engine will automatically select alternative implementations of the requested operation until one succeeds, providing transparent fault tolerance, and solving heterogeneity problems. In case all available middleware implementations fail to provide a requested operation, JavaGAT provides the application with a special, *nested exception*, enabling detailed error analysis when needed. The overhead of using intelligent dispatching is insignificant compared to the cost of the grid operations.

We also have shown how JavaGAT provides a powerful framework that allows grid middleware developers to quickly and efficiently implement GAT bindings to their middleware system, without unnecessary duplication of code. This way, grid researchers can experiment with new middleware ideas without interfering with the grid application programmers. The many middleware adaptors that have been provided by third-party developers indicate the viability of our approach.

We have demonstrated that the JavaGAT implementation as a whole achieves functionality that grid application programmers require today. Several groups have implemented grid applications using the JavaGAT, using the grid for production systems, despite existing shortcomings of the underlying middleware. In the future, we will also apply intelligent dispatching, nested exceptions, and our adaptor development framework to the SAGA reference implementation for Java. As SAGA has been inspired by the GAT, it will combine JavaGAT's benefits described in this paper with an API that will have been standardized within the Open Grid Forum.

9. ACKNOWLEDGMENTS

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl). This project is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W) and is part of the ICT innovation program of the Ministry of Economic Affairs (EZ). This work has been supported

by the Netherlands Organization for Scientific Research (NWO) grant 612.060.214 (Ibis: a Java-based grid programming environment). We kindly thank Niels Drost, Jason Maassen and Frank Seinstra for all their help. The useful feedback of Alexander Beck-Ratzka, Keith Cover and many other users of the JavaGAT implementation is greatly appreciated. We also like to thank the anonymous reviewers for their insightful and constructive comments.

10. REFERENCES

[1] JSch – Java Secure Channel Library. http://www.jcraft.com/jsch.

[2] The AstroGrid Project. http://www.astrogrid.org.

[3] The Castor Project. http://www.castor.org.

[4] The GridLab project. http://www.gridlab.org.

[5] The Ibis Project. http://www.cs.vu.nl/ibis.

[6] The MyProxy credential management service. http://grid.ncsa.uiuc.edu/myproxy.

[7] The Open Grid Forum (OGF). http://www.ogf.org.

[8] The TextGrid Project. http://www.textgrid.de.

[9] The Virtual Labs for E-Science Project (Vl-e). http://www.vl-e.nl.

[10] G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, March 2005.

[11] K. Amin, G. von Laszewski, M. Hategan, R. Al-Ali, O. Rana, and D. Walker. An abstraction model for a grid execution framework. *Journal of Systems Architecture: the EUROMICRO Journal*, 52(2):73–87, feb 2006. ISSN:1383-7621.

[12] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming Scientific and Distributed Workflow with Triana Services. *Concurrency & Computation: Practice & Experience*, 18(10):1021–1037, 2006.

[13] K. Cover, J. Verbrunt, J. de Munck, and B. van Dijk. Fitting a single equivalent-current-dipole model to MEG data with exhaustive search optimization is a simple, practical and very robust method given the speed of modern computers. In *New Frontiers in Biomagnetism, Proceedings of the 15th International Conference on Biomagnetism*, International Congress Series No 1300, Vancouver, B.C., Canada, August 2006. Elevier B.V. ISBN-13:978-0-444-52885-8.

[14] A. Cunei and J. Vitek. PolyD: a flexible dispatching framework. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'2005)*, pages 487–503, San Diego, CA, USA, 2005. ACM Press New York, NY, USA. ISSN:0362-1340.

[15] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13. Springer-Verlag LNCS 3779, 2006.

[16] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-level application programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[17] H. Kaiser, A. Merzky, S. Hirmer, and G. Allen. The SAGA C++ Reference Implementation. In *Library-Centric Software Design LCSD'06 workshop*, Portland, Oregon, October 2006.

[18] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauve. Faults in grids: why are they so bad and what can be done about it? In *Fourth International Workshop on Grid Computing (Grid2003)*, pages 18–24, Phoenix, Arizona, nov 2003.

[19] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. The GridRPC API standardization at the Grid Remote Procedure Call Working Group of the Global Grid Forum, 2005. GGF proposed recommendation (GFD-R.P 52).

[20] H. Rajic, R. Brobst, W. Chan, F. Ferstl, J. Gardiner, A. Haas, B. Nitzberg, and J. Tollefsrud. Distributed Resource Management Application API Specification 1.0 (DRMAA). GFD.22, http://www.drmaa.org.

[21] Y. Tanaka, H. Takemiya, H. Nakada, and S. Sekiguchi. Design, implementation and performance evaluation of GridRPC programming middleware for a large-scale computational Grid. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (Grid2004)*, 2004.

[22] Y. van der Burgt, I. Taban, M. Konijnenburg, M. Biskup, M. Duursma, R. Heeren, A. Rompp, R. van Nieuwpoort, and H. Bal. Parallel Processing of Large Datasets from NanoLC-FTICR-MS Measurements. *Journal of the American Society of Mass Spectrometry*, 18(1):152–161, oct 2006. PubMed ID: 17055738.

[23] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra. Recent Developments in GridSolve. *International Journal of High Performance Computing Applications (Special Issue: Scheduling for Large-Scale Heterogeneous Platforms), Robert, Y eds.*, 20(1), 2006. Sage Science Press.