Facilitating the Portability of User Applications in Grid Environments

Paul Z. Kolano

NASA Advanced Supercomputing Division
NASA Ames Research Center
M/S 258-6, Moffett Field, CA 94035, USA
kolano@nas.nasa.gov

Abstract. Grid computing promises the ability to connect geographically and organizationally distributed resources to increase effective computational power, resource utilization, and resource accessibility. For grid computing to be successful, users must be able to easily execute the same application on different resources. Different resources, however, may be administered by different organizations with different software installed, different file system structures, and different default environment settings. Even within the same organization, the set of software installed on a given resource is in constant flux with additions, upgrades, and removals. Users cannot be expected to understand all of the idiosyncrasies of each resource they may wish to execute jobs on, thus must be provided with automated assistance. This paper describes a new OGSI-compliant grid service (the *Naturalization Service*) that has been implemented as part of NASA's Information Power Grid (IPG) project to automatically establish the execution environment for user applications.

1 Introduction

Grid computing [6] promises the ability to connect geographically and organizationally distributed resources to increase effective computational power, resource utilization, and resource accessibility. Real world experiences with grids [1,11], however, have had mixed results. While gains in computational power were eventually achieved, they were only made a reality after significant efforts to get user applications running on each suitable resource. Differences across resources in installed software, file system structures, and default environment settings required manually transferring dependent software and setting environment variables. This problem is common even in non-grid environments. Users frequently encounter missing or incompatible shared libraries (.so files) on Unix systems or missing dynamic link libraries (.DLL files) on Windows systems when attempting to execute binaries that have been transferred from a similar system. Even in a language that is designed for portability, such as Java, this same problem exists. That is, a Java application can only be executed on a system that has all of the classes installed on which it depends. If all dependent software is present, an application still may not be able to execute if the environment variables are

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 73–85, 2003.
© IFIP International Federation for Information Processing 2003

not set such that it can find that software. In grid environments, this problem is only amplified. For grid computing to be successful, users must be able to easily execute the same application on different resources without being expected to understand or compensate for all of the idiosyncrasies of each resource.

In general, the ability of an application to migrate from one system to another depends on a number of issues. A given application may have dependencies over which an ordinary user has no control such as processor architecture, operating system type, operating system version and features, system architecture, and system configuration. These dependencies limit the set of resources on which the application can execute. An application may also have other dependencies such as software availability, software locations, software versions and features, and environment variable settings. Although the set of resources can also be limited based on these dependencies, this is undesirable as it may eliminate the best resources from consideration even though the user can satisfy these dependencies by copying files and setting path variables appropriately.

The typical approach used for dealing with software dependencies is to rely on statically linked executables or custom packages containing all required software. The drawbacks of statically linked executables are well known including overly large executables, inefficient use of memory, and hard-coding library bugs into code. Building custom software packages correctly and writing an associated setup script may require expert knowledge of dependency analysis techniques, differences in operating systems, and environments required by different software types. Every minute spent by a user constructing a software package or transferring an unnecessarily large file is another minute of the user's time or resource allocations that could be better spent on their real work.

Naturalization is defined[1] as the process of "adapting or acclimating (a plant or animal) to a new environment; introducing and establishing as if native". This paper describes a new grid service (the *Naturalization Service*) developed as part of NASA's Information Power Grid (IPG) project to automatically naturalize user applications to grid resources. The functions of this service include (1) automatically identifying the dependencies of user applications with support for executables, shared libraries, Java classes, and Perl and Python programs, (2) establishing a suitable environment by transferring dependent software and setting key environment variables necessary for each application to run, and (3) managing a flexible software catalog, which is used to locate software dependencies based on both centrally managed and user controlled mappings.

Section 2 presents related work. Section 3 gives a brief overview of the NASA IPG. Section 4 describes the steps involved in establishing the execution environment for an application. Section 5 gives the implementation details of the Naturalization Service. Finally, section 6 presents conclusions and future work.

2 Related Work

There are several projects that address issues similar to those addressed by this work. The Globus Executable Management (GEM) [2] system was implemented

[1] American Heritage Dictionary at `http://www.bartleby.com/61/6/N0030600.html`.

to allow different versions of an executable to be staged to a machine based on its processor architecture type and operating system version. Executables are retrieved from a network-based executable repository. This system only supports executables, however, and has no support for shared libraries, Java classes, or Perl or Python programs, nor does it support automated dependency analysis.

The Uniform Interface to Computing Resources (UNICORE) [4] allows jobs to be built from platform-independent abstract job operations, which are translated into concrete operations that can be executed on an actual system. The translation relies on a static configuration file located on each resource describing the software installed there. For example, an abstract job executing "ls" would be mapped using the configuration file to a concrete job executing "/bin/ls". This approach requires extra administration every time software is added to, removed from, or updated on a system, it only supports executables, and it only allows software to run on systems that already have all required software installed.

The Automatic Configuration Service [10] automatically manages the installation and removal of software for component-based applications according to user-specified dependency information. This service has goals similar to those of the Naturalization Service, but is implemented as a CORBA service as opposed to an OGSI-compliant service. A limitation of this service is that the user must fully specify all dependencies manually. There is also no discussion of managing environment variables, which are required for an application to find installed software and which differ according to software type. In addition, this service uses a centralized repository, thus cannot take advantage of software individually deployed by users.

Installers, package managers, and application management systems [3] are typically used to manage the software installed on standalone systems and systems on the same network. While these approaches greatly increase the ability of system administrators to provide a consistent and stable set of software across an organization's resources, they are only of use when the administrator knows what software will be needed. Since grids enable users from different organizations with different software requirements to share resources, these mechanisms do not provide the necessary level of support.

Replica management systems such as Reptor [8] provide high-level mechanisms for managing the replication, selection, consistency, and security of data to provide users with transparent access to geographically distributed data sets. Much of this functionality is also suitable for managing software across grid resources and is, in fact, the basis of part of the Naturalization Service. Replica management systems do not address software specific issues, however, such as automatic dependency analysis and environment variable settings.

3 NASA Information Power Grid

NASA's Information Power Grid (IPG) [9] is a computational and data grid spanning a number of NASA centers that consists of various high performance su-

percomputers, storage systems, and data gathering instruments scattered across the United States. The goal of the IPG is to increase the utilization and accessibility of existing resources, ultimately resulting in an increase in productivity at each NASA center.

Although a grid provides access to additional resources, as the number of resources increases, it becomes more and more difficult to use those resources. A user must know the name of each resource, which resources they have accounts on, how many allocations they have on each resource, which resources are least loaded, what software is installed on each system, etc. This becomes a daunting task even when the number of resources is small. For this reason, the IPG project is also developing a set of grid services to facilitate the use of the grid [13]. Three prototype services have been implemented including a Resource Broker for selecting resources meeting specified constraints, a Job Manager for reliable job execution, and the Naturalization Service, which is the subject of this paper.

In the current job model of the IPG, jobs consist of a sequence of file and execution operations, each of which may have an associated cleanup sequence. File operations consist of operations on files and directories including copying, moving, and removing files and creating and removing directories. Execution operations describe an application to execute on a given resource. For the purposes of this paper, an execution operation will consist of a host to run on, the path of the application on that host, and an environment mapping from variable names to variable values. An actual IPG Job Manager job consists of a number of other fields that are not relevant to the discussion of the Naturalization Service including a queue name, a project name, a working directory, stdin, stdout, and stderr redirection, a number of processors, and memory requirements.

Within the current IPG architecture, a job to execute and a set of resource constraints are submitted to the Resource Broker component from a client application. The resource constraints consist of restrictions on resource characteristics (e.g. number of processors, operating system type, processor type, etc.) that must be satisfied for the application to properly execute. The Resource Broker selects a set of resources satisfying those constraints and incorporates those selections into the job and passes it on to the Naturalization Service. The Naturalization Service then transforms the job as necessary to establish the execution environment for each application, which is passed on to the Job Manager for execution. These services are based on the Globus Toolkit [5], which provides grid security through the Grid Security Infrastructure (GSI), low-level job management through the Globus Resource Allocation Manager (GRAM), data transfer through the Grid File Transfer Protocol (GridFTP), and resource/service information through the Monitoring and Discovery Service (MDS).

4 Establishing Execution Environments

The execution environment for an application on a given resource consists of the software existing on that resource and the settings of the environment variables when the application runs. For each execution operation in a given job, the function of the Naturalization Service is to:

1. Determine the software that the execution operation application requires
2. Provide a location for that software on the execution operation host by:
   (a) Determining if the software exists on the execution operation host
   (b) Finding a source for any missing software
   (c) Copying missing software to the execution operation host
3. Set environment variables based on provided software locations

A list of dependencies is associated with each execution operation. A dependency consists of basic requirements including a type, a name, a version range, and a feature list as well as information gathered during processing including a source host and path, a target path, and an "analyzed flag" to indicate its analysis status. The Naturalization Service currently supports five software types: executables, shared libraries, Java classes, and Perl and Python programs. The dependency name holds the canonical name for the software depending on its type (e.g. ls, libc, java.util.List, File::Basename, xml.sax.xmlreader, etc.). The version range consists of a minimum and/or maximum version required. The feature list contains features that the dependency must support. For example, the application might require the w3m browser compiled with SSL support. Currently, versions are only supported for shared libraries and features are not yet supported as a way to derive these automatically has not yet been determined.

The source host and path, target path, and analyzed flag are used to store information as processing proceeds. Stages are only executed if the information they provide has not already been gathered. Thus, a job for which the execution environment has already been fully established can be sent through the Naturalization Service without effect. This allows the user to have complete control of job processing. A user can execute stages individually, can specify dependencies manually, can turn analysis off, can specify an exact source for software, can specify a location where software already exists, or any combination thereof. The Naturalization Service will fill in any gaps in the environment left by the user or return the job unchanged if no modifications are necessary.

Although the Naturalization Service makes its best attempt to establish the execution environment for a job, it is not possible to guarantee that the resulting environment will always be suitable. There are three scenarios for which such a guarantee cannot be made:

1. Application depends on A, but A cannot be located anywhere
2. Application depends on A, which depends on B, but analysis techniques used on A are inadequate to determine B is a dependency
3. Application does not depend on A, but analysis reports A is a dependency

Since executing a job for which the execution environment has not been fully established leads to wasted CPU cycles, it is desirable to notify the user prior to job execution. For the last two scenarios, nothing can be done besides documenting the limitations of the analysis techniques. The first scenario, however, can be identified by searching the resulting job for empty target paths and false analyzed flags. The Naturalization Service provides convenience methods for finding unresolved dependencies to determine if a job should be executed as is.

A pedagogical example job will be used to illustrate the functionality of the Naturalization Service. This job consists of running a Python program "addall.py", which uses a module "Adder.py" as shown in Figure 1. Figure 3 shows the original job and the actual modifications made to that job as it passes through the five stages of the Naturalization Service discussed in the following sections.

4.1 Dependency Analysis

In order to establish the environment for a particular application, it is first necessary to analyze exactly what software the application needs. In general, this problem is undecidable as applications can dynamically load or execute a file derived from an arbitrarily complex computation at any point during execution (e.g. char *lib = complex_func(); dlopen(lib)). Although this type of analysis is infeasible for the general case, the large majority of cases are much simpler and can be handled by appropriate static analysis techniques.

The Naturalization Service first collects the complete set of host/file pairs that need to be analyzed for the given job, which includes the execution operation applications and any manually-specified dependencies with a source location. File operations in the job are traversed to find the original location of each file. A single job is then executed on each collected host in parallel. This job runs a self-contained analysis shell script on the list of files located on that host. The output of this job is a list of dependencies for each file, which are added as dependencies to the appropriate execution operations.

Analyzing software on the system it originates on is advantageous since that system is likely to be the one that the user has tested it on. Thus, it is likely to have all dependencies present, even if they are in non-standard locations such as the user's home directory. This facilitates complete analysis and provides a source for files to be transferred to the target system. The main concern is execution time, since it is undesirable for these jobs to wait in a heavily loaded FIFO queue for execution. Since analysis only requires access to files, however, it is not necessary for the analysis jobs to run on the main compute node of a resource. Instead, they can simply run on the file server for that node, either as GRAM jobs or GSI-enabled ssh jobs, which will execute almost immediately.

Currently, the Naturalization Service analyzes executables, shared libraries, Java classes (both class and jar files), and Perl and Python programs. Executable and shared library analysis is the most straightforward as the Executable and Linking Format (ELF) [15] standard used by Unix systems requires that an executable contain the names of the shared object dependencies necessary for it to execute. This information can be obtained using the "ldd" or "elfdump" commands. Like the ELF format, the Java class file format [14] also contains dependency information. Namely, it contains the list of classes that the given class requires to execute. The analysis code uses a slightly modified version of the com.sun.jini.tool.ClassDep utility of the Jini Software Kit[2].

[2] Available at `http://www.sun.com/software/jini`.

```
# addall.py (add #'s from stdin)
import sys
import string
import Adder
adder = Adder.Adder()
for line in sys.stdin.readlines():
  n = string.atoi(line)
  adder.add(n)
print adder.sum()

# Adder.py (maintain sum)
class Adder:
  def __init__(self):
    self.value = 0
  def add(self, n):
    self.value += n
  def sum(self):
    return self.value
```

Fig. 1. Example job.

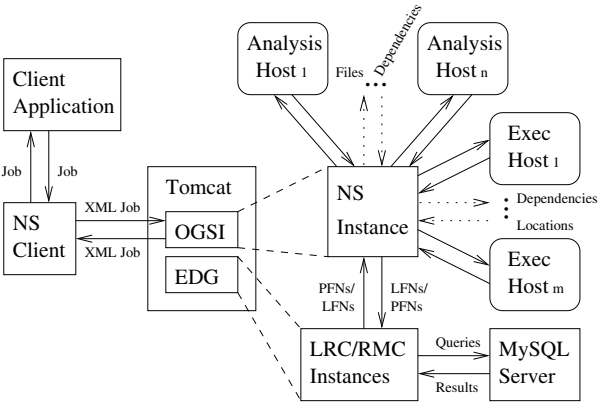

Fig. 2. Naturalization Service implementation.

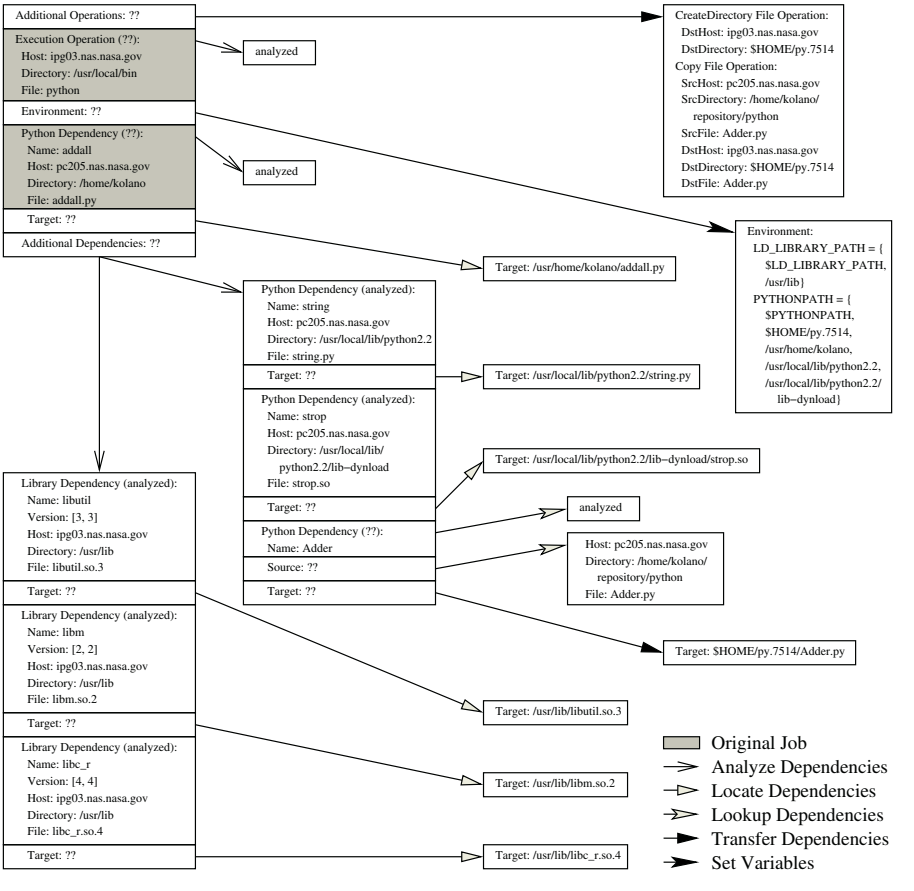

Fig. 3. Stages of job transformation.

Unlike ELF executables and libraries and Java class files, Perl and Python programs do not contain explicit dependency information. These types of programs must either be textually searched for relevant module usage (e.g. "use" or "require" in Perl and "import" in Python) or must be partially evaluated using features of their corresponding compilers such as introspection. The Naturalization Service analysis code is based on the Perl Module:: ScanDeps[3] module, which uses the former approach, and the Python modulefinder module[4], which uses the latter.

After this stage, Figure 3 shows that three Python dependencies have been added to the example job based on the analysis of "addall.py" as well as three library dependencies from the analysis of the "python" executable. The execution operation and all of its dependencies have been marked as "analyzed" with the exception of the Adder Python dependency, whose source could not be found, thus could not be analyzed.

4.2 Dependency Location

Even though different resources may have different sets of software installed, there is a good chance that they also share a significant base of common software. Every piece of software that does not have to be transferred equates to a decrease in the time an application must wait to execute. Thus, after determining the software on which a particular application depends, the Naturalization Service next determines if the software already exists on the target system. While this problem is not undecidable as the number of files on a system is finite, it is impractical to search every file on a system. Thus, the search space must be limited by path variables. Since there is no guarantee these paths are complete or that software is stored in standard locations, there is no guarantee that this procedure will find a particular file even if it actually exists on the system. This problem is amplified through the use of the Globus GRAM as jobs executed by the GRAM job manager do not necessarily run under the user's default shell, so do not incorporate the user's default environment. In this case, even if the user has the path variables set up appropriately, the job still might not be able to find all existing software. In some cases, the user may not have a permanent account on a system, thus may not have the environment set up properly to begin with. Also, the default shell might not be the shell that the user actually uses. This is common on systems where the user cannot control the default shell or where more advanced shells such as bash are not allowed as login shells.

To compensate for this problem, two strategies are employed. First, paths are added according to the Filesystem Hierarchy Standard [12] for Unix systems. This guarantees that most common software will be located as all major Unix distributions conform to this standard. Next, the locator gathers user-defined and system-default environment variable settings from standard shells including bash, csh, ksh, sh, tcsh, and zsh. A variable "var" can be read from a shell

[3] Available at `http://search.cpan.org/perldoc?Module::ScanDeps`.
[4] Available as part of the Python 2.3 base distribution at `http://www.python.org`.

"<sh>" using "<sh> -c 'echo $var"'. Variables gathered include LD_LIBRARY_PATH and variants, PATH, CLASSPATH, JAVA_HOME, PERLLIB and variants, PYTHONHOME, and PYTHONPATH. Once the paths are set, files are located by type, using "ls" for executables and libraries and the corresponding interpreter for Java, Perl, and Python dependencies. It is assumed that if the interpreter is not available, then no dependencies of that type exist on the system.

After this stage, Figure 3 shows that all dependencies of the example job have been located on the target system except for the Adder Python dependency.

4.3 Dependency Lookup

Ideally, after the analysis and location stages, every dependency has either been located on the target system or a source for it has been found during analysis. Since this cannot be guaranteed, however, a final attempt is made to find any unresolved dependencies in a software catalog. This catalog utilizes the Local Replica Catalog (LRC) and Replica Metadata Catalog (RMC) of the European DataGrid (EDG) project [8]. The LRC stores one-to-many mappings from logical file names (LFNs) constructed from software type, name, supported operating system, and version to Globally Unique Identifiers (GUIDs). The RMC stores many-to-one mappings from these GUIDs to physical file names (PFNs) where the associated software actually resides. Note that the roles traditionally taken by these components have been reversed to accommodate the uniqueness constraints imposed on both sides of LRC/RMC mappings by the EDG implementation.

Since dependency analysis has already been performed by this stage, the software catalog also stores the pre-identified dependencies of each PFN, which are recursively added as dependencies and looked up as necessary. In this case, the LRC maps each PFN to a set of GUIDs, each of which is mapped by the RMC to an LFN identifying a specific dependency.

Using a catalog instead of a repository allows for a flexible approach to software management. As long as a resource is accessible to the file transfer mechanisms of the IPG Job Manager, the software on that resource can be utilized by the Naturalization Service. If an organization desires a permanent repository, it can dedicate a set of resources to the task with an appropriate repertoire of software and map LFNs into the file systems of those resources. Otherwise, the LFNs can simply point to the locations of software on existing systems. The design also allows users to manage personal software repositories. The Naturalization Service provides a user interface to add and remove mappings from LFNs in a personal namespace based on their grid identity to the PFNs of choice. Thus, users can maintain a collection of software that they frequently use on their personally selected resources, which will be utilized by the Naturalization Service as a source for the software required by their jobs. For a given LFN, the current implementation first selects the user's PFN, if it exists, or if not, selects the first matching PFN from the main catalog. Future versions of the Naturalization Service will perform more intelligent selection based on locality, reliability, etc.

After this stage, Figure 3 shows that the one dependency without a source or target location, the Adder Python dependency, now has a source. In addition, it has been marked as "analyzed" based on its dependency information in the software catalog, which indicated that no additional software was required.

4.4 Dependency Transfer

Transferring the dependencies to the target system is relatively straightforward. One issue, however, is making sure that an appropriate directory hierarchy is created for Java, Perl, and Python dependencies. For example, Java expects non-jar'd class files to be located in a directory structure based on the class name. Thus, to copy the class file "FooBar.class" associated with a class named "foo.bar.FooBar" to a directory "/basedir" in CLASSPATH, it must actually be copied to "/basedir/foo/bar/FooBar.class". Otherwise, Java will not be able to find the class. The Perl and Python cases are similar.

Another issue at this stage is dependency reuse. A job may contain a sequence of execution operations on the same machine. The dependencies of different execution operations may overlap, thus should only be transferred once before the first operation that requires them. The Naturalization Service keeps track of which files need to be transferred and copies them at the appropriate stage.

After this stage, Figure 3 shows that two file operations have been added to the example job to create a directory for and to copy the one dependency without a target location, the Adder Python dependency, to the target system.

4.5 Variable Setup

The final step in establishing the execution environment for a job is setting the environment variables of each execution operation so that all its dependencies can be located during execution. At this stage, as many dependencies as possible either have a location where they currently reside on the target system or a location where they will reside after a transfer from elsewhere. Thus, the Naturalization Service simply adds each dependency's location to the path variable appropriate for that dependency's type. As in the previous stage, care must be taken when handling Java, Perl, and Python dependencies. For the example in the previous section, if a Java class named "foo.bar.FooBar" has a future location of "/basedir/foo/bar/FooBar.class", the CLASSPATH variable must contain "/basedir" and not "/basedir/foo/bar" for Java to properly find the class. For these cases, the Naturalization Service traverses the location back the appropriate number of directories based on the name. Again, this also applies to Perl and Python modules, but not to Java jar files.

After this stage, Figure 3 shows the execution operation of the example job now has environment settings based on the existing and created locations of its dependencies. At this point, the example job has been fully transformed.

5 Implementation

An initial prototype of the Naturalization Service has been implemented in Java with the dependency analysis and location modules written as Bourne shell

scripts. The Naturalization Service runs as an Open Grid Services Infrastructure (OGSI) compliant service within the Open Grid Services Architecture (OGSA) framework [7]. In the OGSA model, all grid functionality is provided by named *grid services* that are created dynamically upon request. The newest version of Globus, version 3.0 (GT3), is the reference implementation of OGSI and provides all of the functionality of GT2 as grid services.

Figure 2 shows the current implementation of the Naturalization Service. In this figure, a client application uses the Naturalization Service client API to request the establishment of the execution environment for a given job. The Naturalization Service client converts the Java job object into XML for transmission to an Apache Tomcat server running an OGSI container. The OGSI container creates an instance of the Naturalization Service and invokes its "establishEnvironment" method with the given job. The Naturalization Service uses the OGSI GRAM service to execute the analysis script on each host with files requiring analysis in parallel. All jobs are executed using the grid credentials of the client application user, thus users are not given any additional privileges beyond what they normally have. After all dependencies have been gathered, the location script is then executed in parallel on each execution operation host with unresolved dependencies. For any dependencies that could not be located or for which no source could be found, instances of the EDG LRC and RMC are queried in an attempt to find a source. At this point, the Naturalization Service sets up the return job to copy dependencies as necessary and sets the environment variables appropriately. The job is returned in XML to the Naturalization Service client, which converts the job back into a Java object for the client application.

The Naturalization Service has been fully tested on FreeBSD systems and the analysis and location scripts have been tested on IRIX, SunOS, and FreeBSD. It has not yet been deployed in the NASA IPG as GT3 is not mature enough for production IPG usage. The Naturalization Service also has not yet been integrated with the IPG Resource Broker or Job Manager, which are built on top of GT2, but will be integrated with the next versions of these services, which are currently being implemented and will be OGSI-compliant.

6 Conclusions and Future Work

This paper has described the IPG Naturalization Service, which is an OGSI-compliant grid service that has been implemented to automatically establish the execution environment for user applications. The Naturalization Service analyzes applications to determine their software dependencies, locates the software on the target system, if possible, or elsewhere, if not, arranges the transfer of software as necessary, and sets the environment variables to allow each application to find its required software. The Naturalization Service has a flexible design that gives the user considerable control over job processing including choosing which steps to perform and managing the source for frequently used software in a personal software catalog. The Naturalization Service allows users to execute jobs on resources that may have been previously unsuitable due to missing soft-

ware dependencies with no or minimal user intervention. The end result is an increase in user productivity by significantly reducing setup time and hassles and increasing the pool of available resources, allowing for faster turnaround times.

There are a number of directions for future research. One inefficiency of the current design is that if two different jobs require the same dependency on the same resource, the Naturalization Service will copy the dependency twice. One solution for this would be to cache software on resources for use by later jobs. More study is necessary to determine how and when this can be done while preventing malicious or accidental modifications to cached software.

Another area for further study is providing additional dependency types and analysis capabilities. Additional types include shell scripts, makefiles, and data dependencies. Additional capabilities include determining "cross-type dependencies" such as executables invoked from Perl scripts. While the general case is undecidable, this analysis may be possible for simple invocation styles that occur frequently in practice (e.g. system("/bin/ls", @args)).

Implementation issues to be addressed in future versions include full IPG deployment, advanced software installation including packages and compilation in addition to basic file transfer, and full dependency version and feature support.

References

1. Allen, G.: Experiences From the SC'02 Demos. Global Grid Forum 7, Mar. 2003. Available at `http://www.zib.de/ggf/apps/meetings/gab-allen.pdf`.
2. Argonne National Laboratory: Extending the ACTS Toolkit for Wide Area Execution: Supporting DOE Applications on Computational Grids, Distributed Systems Laboratory, 1999. Available at `http://www.mcs.anl.gov/dsl/preport.htm`.
3. Carzaniga, A., Fuggetta, A., Hall, R.S., Heimbigner, D., van der Hoek, A., Wolf, A.L.: A Characterization Framework for Software Deployment Technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, Univ. of Colorado, 1998.
4. Erwin, D.W., Snelling, D.F.: UNICORE: A Grid Computing Environment. 7th Intl. Euro-Par Conf., Aug. 2001.
5. Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Intl. J. Supercomputer Applications. 11(2) (1997) 115-128.
6. Foster, I., Kesselman, C. (eds.): The GRID: Blueprint for a New Computing Infrastructure. Morgan-Kaufmann, San Francisco, CA (1999).
7. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.
8. Guy, L., Kunszt, P., Laure, E., Stockinger, H., Stockinger, K.: Replica Management in Data Grids. Global Grid Forum Informational Document, GGF5, July 2002.
9. Johnston, W.E., Gannon, D., Nitzberg, B.: Grids as Production Computing Environments: The Engineering Aspects of NASA's Information Power Grid. 8th IEEE Intl. Symp. on High Performance Distributed Computing, Aug. 1999.
10. Kon, F., Yamane, T., Hess, C., Campbell, R., Mickunas, D.: Dynamic Resource Management and Automatic Configuration of Distributed Component Systems. 6th USENIX Conf. on Object-Oriented Technologies and Systems, Jan. 2001.
11. Rogers, S., Tejnil, E., Aftosmis, M.J., Ahmad, J., Pandya, S., Chaderjian, N.: Automated CFD Parameter Studies on Distributed Parallel

Computers. Information Power Grid Wkshp., Feb. 2003. Available at http://www.ipg.nasa.gov/workshops/workshop2003/rogers.ppt.
12. Russell, R., Quinlan, D. (eds.): Filesystem Hierarchy Standard – Version 2.2 Final. May 2001. Available at http://www.pathname.com/fhs.
13. Smith, W., Lisotta, A.: IPG Services. Information Power Grid Wkshp., Feb. 2003. Available at http://www.ipg.nasa.gov/workshops/workshop2003/smith.ppt.
14. Sun Microsystems: The Java Virtual Machine Specification. 2nd edn. (1999). Available at http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.
15. Unix System Laboratories: System V Application Binary Interface. 3rd edn. Prentice Hall, Englewood Cliffs, NJ (1993)