A Unified Framework for Periodic, On-Demand, and User-Specified Software Information

Paul Z. Kolano
NASA Advanced Supercomputing Division, NASA Ames Research Center
M/S 258-6, Moffett Field, CA 94035 U.S.A.

E-mail: kolano@nas.nasa.gov

Abstract

Although grid computing can increase the number of resources available to a user, not all resources on the grid may have a software environment suitable for running a given application. To provide users with the necessary assistance for selecting resources with compatible software environments and/or for automatically establishing such environments, it is necessary to have an accurate source of information about the software installed across the grid. This paper presents a new OGSI-compliant software information service that has been implemented as part of NASA's Information Power Grid project. This service is built on top of a general framework for reconciling information from periodic, on-demand, and user-specified sources. Information is retrieved using standard XPath queries over a single unified namespace independent of the information's source. Two consumers of the provided software information, the IPG Resource Broker and the IPG Naturalization Service, are briefly described.

1. Introduction

Although grid computing can increase the number of resources available to a user, not all resources on the grid may have a software environment suitable for running a given application. To provide users with the necessary assistance for selecting resources with compatible software environments and/or for automatically establishing such environments, it is necessary to have an accurate source of information about the software installed across the grid. Existing solutions require manual entry of software information imposing a significant administrative burden. To provide a scalable solution adequate for large, multi-organization grids, a software information service must support:

- true software resource discovery integrated with the tools used by administrators to install software
- user-specified information for locating personal software installations
- extensibility for new types of information as they become needed/available

This paper presents Swim, the Software Information Metacatalog, which is a software information service for the grid built on top of a general framework for reconciling information from periodic, on-demand, and user-specified sources. Software information is periodically gathered from native package managers on FreeBSD, Solaris, and IRIX as well as RPM package managers on multiple platforms including Linux.

Information that is too expensive to gather automatically or is about software in non-standard locations such as personal directories is collected on-demand when necessary. This information is collected under the requesting user's grid identity, thus using their permissions and their allocations, but is cached for their own and the common good. Users may also manually enter information tagged with their grid identity to assist services using this information in finding their own personal software installations. Information is retrieved using standard XPath [4] queries over a single unified namespace independent of the information's source.

Swim is part of NASA's Information Power Grid (IPG) project [12]. The goal of the IPG is to develop new technologies to facilitate the use of the grid and enable scientific discovery. Several prototype services have been implemented including the Execution Service for submitting and managing jobs, the Naturalization Service [13] for automatically establishing the execution environment for user applications, the Surfer framework [14] for selecting and ranking resources, and Swim, which is the subject of this paper.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the periodic, on-demand, and user-specified information framework. Section 4

presents the software information service built using this framework. Section 5 discusses two applications that use this service. Finally, section 6 presents conclusions and future work.

2. Related Work

Several projects address subsets of the issues addressed in this work. The Monitoring and Discovery Service (MDS) [5] of the Globus Toolkit [7] provides grid resource information. MDS has a pluggable architecture that allows new information providers to be integrated into the system. Information can be cached in a back-end XML database. While MDS has much of the required functionality for a software information service, it is mainly designed to support queries and updates of periodically-generated information. It does not have direct support for user-specified information nor does it support true on-demand information retrieval in which the same provider may be executed with different arguments based on the contents of the query itself.

A very basic MDS provider for software information is described in [16], where all installed software of interest on a system must be manually entered into a configuration file, which can then be queried through standard MDS mechanisms. A similar approach is taken by the Uniform Interface to Computing Resources (UNICORE) [6], where platform-independent abstract job operations can be translated into concrete operations for a specific system by replacing abstract software names with concrete paths from the static configuration file for that system. These approaches require significant administrative overhead as the list of installed software must be updated whenever software is installed, removed, or upgraded on a system.

The Repository in a Box (RIB) [2] is a toolkit for building software metadata catalogs. Software information is structured according to the built-in Basic Interoperability Data Model or a custom model defined by the administrator and is accessible through automatically generated web pages. RIB is intended to set up software repositories, thus does not have any mechanisms for automatic software discovery required by a software information service.

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. These approaches greatly increase the ability of system administrators to provide a consistent and stable set of software across an organization's resources. The software information maintained by these tools is vital for automatic software discovery. None of these tools by themselves, however, have the flexibility required of a grid software information service. Resources on the grid may be administered by different organizations, each of which may use its favorite non-interoperable administration tools. Even within the same organization, software may not be installed by the same tool, may be compiled directly from a source distribution, or may be installed by a user for personal use. In these cases, information must be gathered from separate sources, which, in general, is not supported by these types of tools.

A key decision in designing a software information service is how the software information will be structured. A variety of XML schemas have been proposed for describing software. The Open Software Description Format [11] describes software packages and their dependencies for automating software distribution over the internet. The Grid Software Object Specification [16] describes software as grid objects with a list of basic attributes. The Glue Computing Element Schema [1] has elements for listing the software installed on a system and for low-level file attributes. All of these approaches are targeted at a single domain and are not general enough for use in a comprehensive software information service.

Replica management systems such as Reptor [10] 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. While much of this functionality is also suitable for managing software across grid resources, it does not include automatic software discovery, which is critical for scalable software information services.

3. Pour Framework

Pour is a new framework for **P**eriodic, **O**n-demand, and **U**ser-specified **R**econciliation of information. Namely, Pour has the ability to receive periodic information updates, collect information on-demand as needed, and accept user-specified information while presenting a single unified view of the information to the user. Information is processed exclusively in XML and is stored in an XML database for later retrieval. XML databases offer significant advantages over traditional relational databases in such dynamic, heterogeneous information streams. New sources and types of information can be easily integrated into the system without requiring a new schema and/or a complete restructuring of existing data. Any database conforming to the XML:DB API[1] may be used.

Like other information services such as MDS, Pour supports a hierarchical caching architecture for scalability. Namely, Pour repositories may be arranged hierarchically with higher level repositories fetching information from lower level repositories when data is not cached in the local database.

[1] `http://www.xmldb.org`

The primary functionality of Pour is exposed to the user in the XPath query interface. Pour queries return a list of XML DOM nodes satisfying the given XPath. Depending on the XPath specified and the contents of the Pour database, query processing may be as simple as a database lookup or as complex as a series of queries down a Pour hierarchy to a set of Pour repositories that must compute the requested information on-the-fly before the appropriate results are returned. This complexity is invisible to users, who may write any valid XPath and receive results integrated from across the relevant periodic, on-demand, and user-specified sources.

Pour is implemented in Java as an Open Grid Services Infrastructure (OGSI) compliant service within the Open Grid Services Architecture (OGSA) framework [9]. In the OGSA model, all grid functionality is provided by named *grid services* that are created dynamically upon request. The reference implementation of OGSI is the Globus Toolkit [7], 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). Individual components of Pour are described in the following sections.

3.1. Spouts

Pour is a framework for building high-level information services, but does not define any specific types of information itself. New types of information and the methods used to collect them are described by *spouts*, which can then be easily integrated into the system using a single configuration line. Each spout defines the XML namespace for information it supplies. This includes the XML namespace URI (e.g. http://ipg.nasa.gov/swim), the XML prefix used for all attribute and element names (e.g. swim), and the name of the root element for all XML documents produced (e.g. software). To illustrate these concepts, figures 1, 2, and 3 show sample documents produced by the Swim spout, which is described in section 4.

In addition to the XML namespace, a spout must define how it produces its periodic, on-demand, and user-specified information. For periodic information, the spout defines the set of URIs for grid services whose service data it should subscribe to. For on-demand information, the spout defines a set of *collectors* used to fetch specific subsets of the information in its namespace. Finally, for user-specified information, a spout defines the XML Document Type Definition (DTD) that should be used to validate any user-specified information to guarantee consistency. The roots of all documents added to Pour are tagged with three attributes: "pour:source", "pour:user", and "pour:time" to indicate how the document was produced, who produced it, and when it was produced, respectively.

As long as all three types of information in a given spout use the same basic XML structure, information about the same elements can be produced and stored independently. Information is eventually integrated through the use of XPath queries. Namely, queries search across the documents of all three sources and produce a list of unified elements, which to the user, appears as though they were produced from a single source. Thus, for example, in the Swim spout, an XPath query for the information about a specific file may result in a set of basic attributes from the periodic source, a set of dependencies from the on-demand source, and a comment from a user-specified source. Details of periodic, on-demand, and user-specified information handling are given in the following sections.

3.2. Periodic Information

In the OGSA framework, grid services maintain information about themselves in the form of *grid service data*. This data can be directly queried at any time or can be pushed as an XML document to other services that subscribe to this information when any or all of the data changes. The periodic component of Pour obtains its information by subscribing to relevant service data in specific grid services. Using this model, services providing information to Pour can be configured independently of Pour itself. This configuration includes items such as the methods used to generate service data and the rate at which it is generated.

Each spout specifies the URIs of the grid services for which periodic information should be collected. Since a service may collect information for other purposes than just the spout, Pour only subscribes to the data in the spout's XML namespace. When updated grid service data arrives, it is given a "pour:time" timestamp, the "pour:source" attribute is set to the URI of the corresponding grid service, and the "pour:user" attribute is set to "gov.nasa.ipg.pour.Periodic". It is assumed that the services specified in the spout generate truly periodic data. In other words, that each update contains the latest version of the same information so that the last update from the same service can be replaced. For example, the periodic component of Swim described in section 4.1 satisfies this assumption since each update contains the latest view of the software installed on a particular host. Information from the last update can be overwritten since any software reported in that update that does not appear in the latest update must have been uninstalled. Using this assumption, the database can maintain a fairly stable size and will contain only the freshest periodic information.

3.3. User-Specified Information

Users are allowed to add/remove XML documents to/from the Pour database as desired. To maintain the consistency of the information in the database, user supplied documents are validated against the DTDs defined in each spout in the system. If no DTD is found for which the document can be validated, the document is rejected.

Before inserting a document into the database, it is timestamped, the "pour:source" attribute is set to "user", and the "pour:user" attribute is set to the grid identity of the submitting user (e.g. /O=Grid/O=National Aeronautics and Space Administration/OU=Ames Research Center/CN=Paul Kolano). Users can only remove their own documents as determined by the user attribute. This attribute can also be utilized by users to restrict query results to only the documents they or the framework submitted.

3.4. On-Demand Information

Information that is too expensive to periodically collect or that is related to individual users can be collected on-demand. On-demand processing, which can be disabled on a per query basis if desired, occurs according to the *collectors* defined in each spout. A collector defines a set of XPath prefixes for which it has information and a set of XPath restrictions that the query XPath must satisfy. These restrictions can include specific attributes or values that must be defined and/or specific values they must take. For example, one of the collectors of the Swim spout of section 4 provides the following XPath prefix:

- /swim:software/swim:file/swim:dependencies

and has the following XPath restrictions:

- /swim:software/swim:file[@swim:host]
- /swim:software/swim:file[@swim:path]

These restrictions mean that the query XPath given by the user must have values for the attributes "swim:host" and "swim:path". A collector also defines the length of time its information can be considered valid. When an XPath query is made by the user for which there is no information in the database or for which information exists, but the "pour:time" attribute is too far in the past, the given XPath is parsed and stripped of all but the absolute paths requested. In addition, the attribute/element values required of each subpath are stored in an argument map. For example, the query "/swim:software/swim:file[@swim:host='host1.nas.nasa.gov' and @swim:path='/some/file']/swim:dependencies" has a single absolute path "/swim:software/swim:file/swim:dependencies" and two attribute mappings "/swim:software/swim:file[@swim:host] => host1.nas.nasa.gov" and "swim:software/swim:file[@swim:path] => /some/file". In this case, since the path requested has a collected prefix as its prefix and both path restrictions are met, the given collector can be executed.

There are two built-in types of collectors. The first type supports hierarchical caching, which is one form of on-demand information. A collector of this type must specify the set of Pour grid service URIs at the next lower level in the hierarchy. When invoked, this collector simply passes the query on to these URIs for processing. Note that with the use of the XPath prefixes and restrictions defined in a collector, sophisticated hierarchies can be built use different URIs for different subsets of information.

The second type of collector supports collection using the Globus GRAM. A collector of this type must specify an executable (typically a shell or perl script) to run and a host to run it on given the argument map. This executable is then run using GRAM on the given host with the appropriate arguments derived from the argument map. A local Globus Access to Secondary Storage (GASS) server is used to transfer the executable and retrieve the XML output. Since the collector may gather more information than was requested in the original XPath query, the final step of processing is to evaluate the XPath against the collected results, which returns the information the user requested.

A benefit of using the GRAM service is that the collection occurs under the user's grid identity, thus the collection executable runs with the user's permissions and the time to compute the information is charged against the user's allocations. In this way, the grid infrastructure can pay for general-purpose information applicable to all users using the periodic mechanism, while specialized information that may only be of use to to a specific user is paid for by the user who requires it. On-demand information is donated for the common good after it is collected, so other users may benefit from each other's cached results.

When information is retrieved on-demand, the collected information is cached in the database with the "pour:source" attribute set to the collector class name and the "pour:user" attribute set to the grid identity of the submitting user. Thus, users that run the same query several times will only pay the price of collection once. Users do not need to be aware that this processing is occurring and do not need to change their queries in any way as it is completely based on the contents of their original XPath query.

4. Swim

Swim is a Software Information Metacatalog built on top of the Pour framework. Swim consists of a spout and a set of collectors that provide information about software installed across the grid. Figures 1, 2, and 3 show a sample of the types of information provided by Swim. The two main classes of information provided are for software packages and software files. The software package information describes which packages of which types have been installed on each system along with supporting information such as a short text comment and each package's dependencies. The software file information describes which executables and supporting libraries have been installed on each system. File information is currently only reported for Executable and Linking Format (ELF) executables and shared libraries, Java classes, Perl scripts and modules, shell scripts, and Python scripts and modules. Two applications that already utilize this information are described in section 5.

4.1. Periodic Information

Swim uses the Globus Toolkit Version 3 (GT3) MDS (also called the Index Service) as the source of its periodic information. The Index Service can be easily configured to run a script periodically on the Index Service host. The Swim spout subscribes to information in the Swim XML namespace, which is forwarded from the Index Service when it changes. New results overwrite previous results stored in the database. System administrators can choose how often to gather information by using the appropriate Index Service configuration. As software information does not change rapidly, it is not necessary to gather information more than once every few days. The Index Service also serves as a redundancy mechanism in case the main Pour repository fails. In this case users can bypass Swim entirely and directly query individual MDS servers, although they will lose the benefits of on-demand and user-specified information.

The Swim script invoked by the Index Service utilizes a set of Perl modules that have been developed to collect software information from different platform types. The main source of information is from the package managers used on each system. Swim collects information from native package managers on FreeBSD, Solaris, and IRIX, as well as RPM package managers on multiple platforms including Linux. It is advantageous to use package managers since in most cases they are the tools used by administrators to install the software in the first place. Since not all software is available or installed in package form, however, Swim also crawls the set of relevant paths from the Filesystem Hierarchy Standard [17], which defines the standard filesystem structure used by all major Unix distributions. Using these two techniques, the vast majority of software installed on a system will be located. Figure 1 shows a small sample of the periodic information that is automatically gathered.

```
<swim:software xmlns:swim="http://ipg.nasa.gov/swim"
  xmlns:pour="http://ipg.nasa.gov/pour"
  pour:time="1076266586754"
  pour:user="gov.nasa.ipg.swim.spout.Swim"
  pour:source="http://keko.nas.nasa.gov:8080
    /ogsa/services/base/index/IndexService">
 <swim:package swim:host="keko.nas.nasa.gov"
   swim:name="Mesa-3.4.2_2" swim:type="native">
  <swim:os>freebsd</swim:os>
  <swim:arch>i386</swim:arch>
  <swim:version>3.4.2_2</swim:version>
  <swim:comment>
   A graphics library similar to SGI&amp;apos;s OpenGL
  </swim:comment>
  <swim:dependencies>
   <swim:package swim:name="imake-4.2.0_1"
     swim:type="native" swim:os="freebsd"
     swim:arch="i386" swim:version="4.2.0_1"/>
   <swim:package swim:name="freetype2-2.1.2"
     swim:type="native" swim:os="freebsd"
     swim:arch="i386" swim:version="2.1.2"/>
   <swim:package swim:name="XFree86-libraries-4.2.1_1"
     swim:type="native" swim:os="freebsd"
     swim:arch="i386" swim:version="4.2.1_1"/>
  </swim:dependencies>
 </swim:package>
 <swim:file swim:host="keko.nas.nasa.gov"
   swim:path="/usr/X11R6/lib/libMesaGL.so">
  <swim:name>libMesaGL.so</swim:name>
  <swim:type>elf_shared</swim:type>
  <swim:os>freebsd</swim:os>
  <swim:arch>i386</swim:arch>
  <swim:size>467028</swim:size>
  <swim:mode>755</swim:mode>
  <swim:owner>root</swim:owner>
  <swim:group>wheel</swim:group>
 </swim:file>
</swim:software>
```

Figure 1. Swim periodic sample

As mentioned above, Swim only gathers information on the specific file types that encompass executable software and supporting libraries. To distinguish between these types and the other types that comprise the majority of files on a system, the Swim scripts use a custom pure Perl implementation of the Unix "file" command that has a subset of its functionality, but is smaller, faster, and more portable. Files

with an appropriate type are further analyzed to gather additional information, which is then formatted in XML and returned with the other results back to the Index Service.

New package managers can be integrated into the system in a modular fashion with relatively little work using the existing modules as templates. The key pieces are the commands for retrieving the names of all installed packages and for listing the detailed information about a specific package. A parser must be written to gather specific fields after which the common routines for retrieving file information and outputting the appropriate XML can be called.

4.2. On-Demand Information

Swim currently supports two on-demand collectors. The first collector gathers dependency information about specific files. This analyzer is based on the dependency analyzer developed in previous IPG Naturalization Service work [13] and gathers the specific software that is required for the correct execution of ELF executables and libraries, Java classes, and Perl and Python modules. Section 3.4 shows the XPaths that are applicable to this collector. Figure 2 shows the results obtained from running this collector on the file "/usr/X11R6/lib/libMesaGL.so" located on host "keko.nas.nasa.gov", which results in a single shared library dependency "libXThrStub.so.6". Section 5.2 describes one of the uses of this collector in more detail.

```
<swim:software xmlns:swim="http://ipg.nasa.gov/swim"
  xmlns:pour="http://ipg.nasa.gov/pour"
  pour:time="1076307760781"
  pour:user="/O=Grid/O=National Aeronautics and Space
   Administration /OU=Ames Research Center/
   CN=Paul Kolano"
  pour:source="gov.nasa.ipg.swim.collector.Dependencies">
 <swim:file swim:host="keko.nas.nasa.gov"
  swim:path="/usr/X11R6/lib/libMesaGL.so">
  <swim:dependencies>
   <swim:file swim:name="libXThrStub.so.6"
    swim:type="elf_shared" swim:version="unknown"/>
  </swim:dependencies>
 </swim:file>
</swim:software>
```

Figure 2. Swim on-demand dependencies

The second collector is an experimental collector for locating a given Perl module using the Comprehensive Perl Archive Network (CPAN)[2]. The idea of this collector is that even if a specific module cannot be located anywhere on the grid, it can still be located using an existing external internet repository. This collector provides the following XPath prefix:

- /swim:software/swim:file

and has the following XPath restrictions:

- /swim:software/swim:file[swim:type = 'perl']
- /swim:software/swim:file[swim:name]

When this collector is executed, it runs a Perl script that uses Perl's CPAN module to contact an available CPAN server and find the path to the distribution containing the given module. A set of source URIs is then constructed based on the ftp servers used in the CPAN configuration. This information is formatted appropriately in XML and returned to Swim. Figure 3 shows the results obtained from using this collector on a Perl module name "File::Type".

```
<swim:software xmlns:swim="http://ipg.nasa.gov/swim"
  xmlns:pour="http://ipg.nasa.gov/pour"
  pour:time="1076310278764"
  pour:user="/O=Grid/O=National Aeronautics and Space
   Administration /OU=Ames Research Center/
   CN=Paul Kolano"
  pour:source="gov.nasa.ipg.swim.collector.CPAN">
 <swim:file>
  <swim:name>File::Type</swim:name>
  <swim:type>perl</swim:type>
  <swim:sources>
   <swim:archive swim:type="src_perl" swim:version="0.12"
    swim:uri="ftp://cpan.cse.msu.edu/authors/id/
     P/PM/PMISON/File-Type-0.12.tar.gz"/>
   <swim:archive swim:type="src_perl" swim:version="0.12"
    swim:uri="ftp://archive.progeny.com/CPAN/authors/id/
     P/PM/PMISON/File-Type-0.12.tar.gz"/>
  </swim:sources>
 </swim:file>
</swim:software>
```

Figure 3. Swim on-demand CPAN sample

The eventual goal is to develop a comprehensive set of collectors for all of the supported file types using repositories such as RpmFind[3] and Solaris Freeware[4] for executables and shared libraries, the Vaults of Parnassus[5] for Python modules, etc. This information will be extremely valuable in automatically establishing user environments as described in section 5.2.

[2] http://www.cpan.org

[3] http://www.rpmfind.net

[4] http://www.sunfreeware.com

[5] http://www.vex.net/parnassus/

4.3. Performance

Table 1 shows Swim periodic information results obtained over a small grid testbed consisting of 9 systems. The results include the number of software packages and files located as well as the average XML document size, the average collection time, and the average database insertion time using the eXist[6] XML database. As can be seen, a significant number of software files were located, which was only a fraction of the files inspected. Manually configured software information services simply could not support this volume of information. Collection time was reasonable enough to run every day if desired. Documents were inserted in minimal time even though they were fairly large.

Table 2 shows Swim query results when hosted on a 2.4 GHz Pentium 4 running Linux with 512 MB of memory and using the eXist XML database. The database was filled with the periodic information collected in table 1. The results show the XPath used for the query (with the "swim:" prefix removed), the time the query took, and the number of results obtained. The results are given in pairs where in the first query, the data is not cached locally so must be collected on-demand. The collectors used are the dependency collector, the CPAN collector, and the hierarchical caching collector, respectively. As can be seen, queries are very fast when the information is cached locally. The dependency and CPAN collectors are over an order of magnitude slower when the information must be collected on-demand. Most of this overhead is due to the use of the Globus GRAM and not from the executables used to collect the information.

5. Swim Applications

Two applications have already been developed to take advantage of the information provided by Swim: the IPG Resource Broker and the IPG Naturalization Service. These applications are discussed in more detail below.

5.1. IPG Resource Broker

The IPG Resource Broker is a grid service for selecting and ranking grid resources based on user-specified constraints and preferences. The Resource Broker is built using Surfer [14], which is an extensible brokering framework that can be customized to any grid environment by adding information providers knowledgeable about that environment. A Surfer provider has been written for Swim to select software resources. This allows users to not only find the compute resources that have a particular piece of software installed, but also allows them to find the exact path to that software. Figure 4 shows an example Resource Broker request to find a compute resource running Linux with at least 128 CPUs and an ELF executable "java" on the same host that is at least version 1.3.1 and is world readable and world executable.

```
Resource:
  Id: c1
  Type: ComputeResource
  Constraint:
      freeCpus >= 128
   && operatingSystem == "Linux"
  Ranking:
      freeCpus
```

```
Resource:
  Id: s1
  Type: SoftwareResource
  Constraint:
      name == "java"
   && type == "elf"
   && version == "1.3.1"
   && mode % 10 == 5
   && host == $c1.host
```

Figure 4. Resource Broker request

Figure 5 shows the Swim query that is automatically constructed by the Surfer provider to support this request. Note that the complex restriction on the software's mode is handled by the provider once the initial results of the query have been returned. The restriction on the host names between resources is handled by the framework itself.

```
/swim:software/swim:file
  [swim:name='java'][swim:type='elf'][swim:version='1.3.1']
```

Figure 5. Resource Broker query

5.2. IPG Naturalization Service

The IPG Naturalization Service [13] is a grid service for automatically establishing the execution environment for user applications. In order to establish an execution environment, the Naturalization Service (1) determines the software that the user application requires, (2) provides a location for that software on the execution host either by finding already existing software on that host or by finding a source for the software elsewhere on the grid and copying it to the execution host, and (3) sets environment variables based on the provided software locations.

The original implementation of the Naturalization Service had its own software catalog based on the Local Replica Catalog and Replica Metadata Catalog of the European DataGrid project [10], which stored manually-specified software location and dependency information. The dependencies of user applications were analyzed using shell scripts that were executed by the Globus GRAM service. These dependencies were then located on the execution host using another set of shell scripts. Any dependencies that could not be located were then looked up in the software catalog.

The original implementation suffered from three major drawbacks. First, the location and dependency information

[6] http://www.exist-db.org

Platform	FreeBSD	IRIX	Linux	Solaris	Totals
Systems	1	3	4	1	9
Total Software Packages	356	1948	2588	561	5453
Total Software Files	4260	40863	28123	4460	77706
Avg. XML Size	1.73 MB	1.56 MB	2.57 MB	2.05 MB	18.7 MB
Avg. Collection Time	174 sec	592 sec	140 sec	1097 sec	3607 sec
Avg. Insertion Time	3.74 sec	3.71 sec	5.96 sec	6.68 sec	45.4 sec

Table 1. Swim periodic results

Query XPath	Query Time	Results
/software/file[@host='evelyn.nas.nasa.gov'][@path='/usr/lib/libcil.so.3']/dependencies/file	30.6 sec	22
/software/file[@host='evelyn.nas.nasa.gov'][@path='/usr/lib/libcil.so.3']/dependencies/file (cached)	1.01 sec	22
/software/file[type='perl'][name='File::Type']/sources/archive	38.2 sec	10
/software/file[type='perl'][name='File::Type']/sources/archive (cached)	1.02 sec	10
/software/file[type='elf'][name='java'] (cached remote)	2.31 sec	21
/software/file[type='elf'][name='java'] (cached local)	1.04 sec	21

Table 2. Swim query results

was not cached, so the user had to pay the price of GRAM's overhead every time. Second, the software catalog had to be manually populated and kept up-to-date. Finally, the software catalog was tightly coupled to the Naturalization Service so did not readily lend itself to other applications such as the Resource Broker.

With the information accessible from Swim, this service has been considerably enhanced. Figure 6 shows the Swim queries used to extract the dependency and location information required by the Naturalization Service. Since the dependency information is computed on-demand, the user may experience a delay in the first query, but afterwards will obtain the cached results immediately. Software information is automatically gathered from across the grid periodically, thus there is minimal administrative overhead, although the user still has the option of manually specifying personal software installations so the Naturalization Service will have more accurate information.

Once all of the external repository collectors of section 4.2 have been implemented, the Naturalization Service will be able to offer more advanced functionality. Specifically, required software not found on the local grid will be located in an appropriate external internet repository. Once found, the software can be temporarily installed on-the-fly as necessary using the appropriate installation mechanisms (e.g. using a package manager, compiled from source, etc.).

1. Find dependencies of a file with a given path on a specific host:
```
/swim:software/swim:file
  [@swim:host='host1'][@swim:path='/path1']
    /swim:dependencies
```
2. Find location of a file with a given name, type, and version on a specific host:
```
/swim:software/swim:file[@swim:host='host2']
  [swim:name='name2'][swim:type='type2']
    [swim:version='version2']
```
3. Find locations of a file with a given name, type, and version:
```
/swim:software/swim:file
  [swim:name='name3'][swim:type='type3']
    [swim:version='version3']
```

Figure 6. Naturalization Service queries

6. Conclusions and Future Work

This paper has described a new software information service for grid computing called Swim, the Software Information Metacatalog. Swim is built using a unified framework for integrating periodic, on-demand, and user-specified information where on-demand processing is initiated during queries depending on the XPath given and the current contents of the local XML database. Swim repositories can be arranged hierarchically for scalability and have a modular architecture for integrating new information sources, which include grid services as well as custom collectors using the Globus GRAM. Swim's use of

the Globus MDS provides built-in redundancy as users can query MDS servers directly for software information about specific hosts.

The most important contribution of Swim is the new set of software information it automatically provides to grid users. Software information is a critical component of seamless computing across multiple systems and organizations. Until now, an adequate software information source with true automatic software discovery based on the tools used by systems administrators was not available. With such a source available, resource brokers can more accurately select compatible resources and more existing resources can be automatically made compatible, which, in the end, results in greater user productivity.

There are several directions for future research. Additional package managers will be incorporated into periodic collection scripts. These will include the native Perl manager, the native Python manager, and the Globus Packaging Toolkit. Additional on-demand collectors will also be developed. Besides the external repository collectors mentioned in section 4.2, collectors will be implemented to gather Unix "stat" information for a given file and to compute an MD5 or similar hash of a given file to verify integrity. Additional collectors will be added as necessary.

References

[1] Andreozzi, S.: Glue Computing Element Schema Version 1.1. Mar. 2003. Available at `http://www.cnaf.infn.it/~sergio/datatag/glue/v11/CE/GlueCE_DOC_V_1_1.htm`.

[2] Browne, S., McMahan, P., Wells, S.: Repository in a Box Toolkit for Software and Resource Sharing. Technical Report UT-CS-99-424, Dept. of Computer Science, Univ. of Tennessee, May 1999.

[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, Apr. 1998.

[4] Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C Recommendation, Nov. 1999. Available at `http://www.w3.org/TR/xpath`.

[5] Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. 10th IEEE Intl. Symp. on High Performance Distributed Computing, Aug. 2001.

[6] Erwin, D.W., Snelling, D.F.: UNICORE: A Grid Computing Environment. 7th Intl. Euro-Par Conf., Aug. 2001.

[7] Foster, I., Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit. Intl. J. Supercomputer Applications. 11(2) (1997) 115-128.

[8] Foster, I., Kesselman, C. (eds.): The GRID: Blueprint for a New Computing Infrastructure. Morgan-Kaufmann, San Francisco, CA (1999).

[9] 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, Jun. 2002.

[10] Guy, L., Kunszt, P., Laure, E., Stockinger, H., Stockinger, K.: Replica Management in Data Grids. Global Grid Forum 5 Informational Document, Jul. 2002.

[11] van Hoff, A., Partovi, H., Thai, T.: The Open Software Description Format (OSD). W3C Note, Aug. 1997. Available at `http://www.w3.org/TR/NOTE-OSD`.

[12] 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.

[13] Kolano, P.Z.: Facilitating the Portability of User Applications in Grid Environments. 4th IFIP Intl. Conf. on Distributed Applications and Interoperable Systems, Nov. 2003.

[14] Kolano, P.Z.: Surfer: An Extensible Pull-Based Framework for Resource Selection and Ranking. 4th IEEE/ACM Intl. Symp. on Cluster Computing and the Grid, Apr. 2004.

[15] Miller, J.: Grid Software Object Specification. Feb. 2001. Available at `http://www-unix.mcs.anl.gov/gridforum/gis/reports/software/software.pdf`.

[16] Miller, N.: A Software Installation Information Provider for MDS 2.x. Apr. 2003. Available at `http://gldap.mcs.anl.gov/neillm/mds/info-providers`.

[17] Russell, R., Quinlan, D. (eds.): Filesystem Hierarchy Standard – Version 2.2 Final. May 2001. Available at `http://www.pathname.com/fhs`.