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, an accurate source of information about the software installed across the grid is needed. Existing software information services and general-purpose information frameworks are inadequate for this task. 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, an accurate source of information about the software installed across the grid is needed. Existing solutions require manual entry of software information imposing a significant administrative burden, do not allow for inclusion of user-specific software installations, and only support a very limited number of software attributes.

In general, the information frameworks currently in existence tend to focus on either information about computational resources or information about data resources. In the former case, this includes automatic collection of attributes common across all such resources such as available CPUs, available memory, CPU load, etc., which are relatively small in number but highly dynamic, thus require frequent updates to maintain accuracy. In the latter case, this includes metadata with widely varying characteristics, which may be large in size and be defined and updated by individual users relatively infrequently as data is generated. These characteristics lead to designs centered around their respective goals and notions of scalability.

Software information occupies a middle ground. Like computer information, different pieces of software share many common attributes that can be automatically collected. Like data information, the volume of information stored in each update may be very large as a single computer may have tens of thousands of software executables, libraries, etc. installed, but these updates can be performed relatively infrequently. In addition, users may wish to publish information about their personal software installations. Unlike either, software may have many attributes that can be derived automatically, but may be deemed too expensive to compute for every software instance. This middle ground requires a new information framework as computer-oriented frameworks are not built to handle large volumes of information nor allow user-specified information while data-oriented frameworks have no support for automatic collection and leave users completely on their own when publishing information.

The contributions of this paper are twofold. First, we present Pour, a new general-purpose information service framework for **P**eriodic, **O**n-Demand, and **U**ser-Specified Information **R**econciliation. Pour is designed to accom-

*This work is supported by the NASA Advanced Supercomputing Division under Task Order A61812D (ITOP Contract DTTS59-99-D-00437/TO #A61812D) with Advanced Management Technology Inc.

modate “middle ground” information with support for high volume, low frequency periodic updates, user-specified updates, and automatic updates collected on-demand when needed. Information is retrieved using standard XPath [3] queries over a single unified namespace independent of the information’s source. Periodic information can be published using secure, lightweight information updates suitable for inclusion in cron jobs, subscriptions to grid service data published by existing services, or embedded Java objects. Information deemed too expensive to gather periodically or specific to individual users is collected on-demand when necessary based on a novel XPath parsing strategy. 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 publish information using specifically tailored on-demand queries or manually created XML documents.

Second, we present Swim, the **S**oft**w**are **I**nformation **M**etacatalog, which is a software information service for the grid built on top of Pour. Swim provides true software resource discovery integrated with the tools used by administrators to install software. In particular, software information is periodically gathered from native package managers on FreeBSD, Solaris, and IRIX as well as the RPM, Perl, and Python package managers on multiple platforms. Swim has additional facilities for collecting on-demand information about arbitrary software on any grid-enabled resource including software dependencies and Unix “stat” information as well as locating Perl distributions from the Comprehensive Perl Archive Network (CPAN)[1].

Pour and Swim are part of NASA’s Information Power Grid (IPG) project [13]. 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 [18] for submitting and managing jobs, the Naturalization Service [14] for automatically establishing the execution environment for user applications, the Surfer framework [15] for selecting and ranking resources, and Pour and Swim, which are the subjects of this paper.

This paper is organized as follows. Section 2 describes related work. Sections 3 and 4 present Pour and Swim, respectively. Section 5 describes two Swim applications. 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 [8] provides grid resource information using a pluggable architecture that allows new information providers to added and allows information to be cached in a back-end XML database. MDS was mainly designed to support small amounts of frequently updated computer 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. The MDS also cannot handle the large (>1 MB) XML documents needed for software information due to its use of Document Object Model (DOM) objects, which utilize memory inefficiently.

A very basic MDS provider for software information is described in [16], where all installed software of interest on a system can be manually entered into a configuration file, which is then 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 Relational Grid Monitoring Architecture (R-GMA) [4] is an extensible information service framework that allows information to be produced from many sources including databases and streams and then integrated using queries based on a subset of SQL. R-GMA has many of the same goals and capabilities as Pour, but has a much different architecture based on registries that store the producers associated with different subsets of information, which can then be queried individually by consumers. It is also based on a relational database model and SQL instead of an XML database model and XPath.

Several data-oriented projects are relevant to this work. The Mobius Project [12] is a general-purpose metadata catalog based on XML supporting XPath queries across multiple databases and XML schema validation of user-specified data. Full replica management systems such as Reptor [11] 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. The Repository in a Box (RIB) [1] 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. None of these approaches provide mechanisms for automatically collecting information on-demand.

Installers, package managers, and application management systems [2] are used to manage the software installed on standalone systems and systems on the same network. While the software information they maintain is vital for

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

automatic software discovery, none of them have the required flexibility. Namely, 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 be installed by different tools, may be compiled directly from source, or may be installed individually by users. Information about such installations is not maintained by these tools.

3. Pour Framework

Pour is a new framework for **P**eriodic, **O**n-Demand, and **U**ser-Specified Information **R**econciliation that accepts periodic information updates, collects information on-demand as needed, and accepts user-specified information while presenting a single unified view 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 since 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[2] 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 querying lower level repositories when data is not cached in the local database.

The primary functionality of Pour is exposed to the user through its query interface. A query consists of any number of XPaths, where each XPath returns a list of the XML strings that satisfy it. Queries are batched together when appropriate to optimize performance. Depending on the XPaths 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 compute the requested information on-the-fly before the appropriate results are returned. This complexity is invisible to users, who may utilize any valid XPath to retrieve 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 [10]. 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 [8], 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.

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

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*. Each spout defines the XML namespace for information it supplies, which includes the XML namespace URI, the XML prefix used for all attribute and element names, and the name of the root element for all XML documents produced.

In addition to the XML namespace, a spout must define how it produces its periodic, on-demand, and user-specified information through *pumps* and *drains* that are hooked into the system. A pump represents pull-based information that is actively collected from external sources by Pour itself. A drain represents push-based information that is produced elsewhere before flowing into the system. An XML configuration file describes all of the spouts, drains, and pumps to be incorporated into the system.

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.

3.2. Drains

3.2.1. Periodic Drains

In Pour, periodic information is defined to be information that comes from a trusted source at an unknown frequency such that any previous information from the same source can be completely overwritten. This model allows periodic sources to be configured entirely independent of Pour itself, while allowing the database to maintain a fairly stable size. Pour supports three periodic update mechanisms based on OGSA notifications, keyed hashes, and embedded Java objects.

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. An *OGSA notification drain* obtains its information by subscribing to relevant service data in specific grid services. Each drain specifies the URI of a grid service for

which periodic information should be collected. Since a service may collect information for other purposes than just the drain, the drain only subscribes to the data in its spout's XML namespace. Using this drain, Pour can act as a higher level information service built on top of other already existing grid information services.

In many cases, it is inefficient to keep a grid service or other process continuously running on a system for the purpose of gathering information. This is especially inefficient for high volume, low frequency updates, which may use significant memory resources that are not released between updates. For these cases, it is desirable to periodically publish information using a lightweight mechanism such as cron jobs that completely release all resources between updates. Since periodic information is defined to be trusted, however, a suitable authentication mechanism must be utilized to guarantee integrity. Standard grid credentials are not appropriate since a cron job may not be owned by a user that has a grid identity (e.g. the root user) or may not be able to create a grid proxy without user interaction. A *keyed drain* provides a secure, lightweight periodic update mechanism based on cryptographic hashes. Each drain defines a secret key and a keyed hash algorithm (e.g. HMAC-MD5). To publish periodic information, the caller must provide an XML document and a source id as well as the combined hash of those items. If the hash is valid for the key and algorithm of some keyed drain within the spout associated with the document's XML namespace, it is accepted as a periodic update and processed accordingly.

The two drain types above assume that information is produced outside of Pour and then is published into the framework. The third drain type, an *embedded drain*, allows periodic information to be produced within Pour itself by an arbitrary Java class. This class is instantiated by Pour and granted direct access to the internal update mechanisms of the framework. The Java class can obtain its information from any source and/or by any means and add it to the database with the lowest overhead possible. Using embedded drains, Pour can serve as a completely self-contained information service, if desired.

3.2.2. User Drains

While it may be desirable to allow user-specified information, in most cases it is also desirable to limit this information in scope and form to maintain database consistency. A spout can allow specific types of information to be added by plugging in any number of *user drains*, each of which specifies an XML schema definition (XSD) [7] and a length of time the information is valid. Any information added by the user must conform to the XSD of some user drain in the system or else it is rejected. After the given amount of time has elapsed, the information is purged from the database. Thus, users can add information to the system, but are limited to adding only the information that has been deemed relevant to the spout and in the appropriate form.

Before inserting a document into the database, it is tagged with the grid identity of the submitting user (e.g. /O=Grid/O=National Aeronautics and Space Administration/OU=Ames Research Center/CN=Paul Kolano). This tag is used by other Pour operations. Specifically, users can only remove information they have previously added and can optionally restrict query results to exclude information submitted by other users.

3.3. Pumps

Information that can be automatically collected, but 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, is triggered according to the *pumps* defined in each spout. A pump 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 elements that must be defined and/or specific values they must take. For example, consider a simple self-populating information service that can determine the operating system and architecture of any computer system on the grid given its host name. Suppose the XML structure for this information consists of a top-level "computer" element with an attribute "name" and subelements "os" and "arch". In this case, the relevant pump prefixes would be "/computer/os" and "/computer/arch" and the one pump restriction would be "/computer[@name]", meaning that the query XPath given by the user must have a value for the attribute "name".

When an XPath query is performed by the user for which there is no information in the database, 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 a argument map. For example, the query "/computer[@name='host1.nas.nasa.gov']/os" has a single absolute path "/computer/os" and a single argument mapping "/computer[@name] => host1.nas.nasa.gov". In this case, since the path requested begins with a collected prefix and the name restriction has been met, the given pump can be triggered. Information is gathered from all pumps that are relevant to the query and is valid for a configurable amount of time.

On-demand information provides a secondary benefit of allowing users to publish information without requiring detailed knowledge of XML. If a suitable pump is available, the user only needs to supply an appropriate query to trigger on-demand collection, which will create the relevant XML document automatically and add it to the database.

Pour currently provides two types of built-in pumps. The

Pour pump supports hierarchical caching, which is one form of on-demand information. Each pump of this type must specify a Pour grid service URI at the next lower level in the hierarchy. When invoked, this pump simply passes the query on to this URI for processing. Note that by using the XPath prefixes and restrictions defined in a pump, sophisticated hierarchies can be built that use different URIs for different subsets of information.

The *GRAM pump* supports collection using the Globus GRAM. A pump of this type must specify an executable (typically a shell or Perl script) to run and a host to run it on based on the given 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. The executable may gather a superset of the information requested, thus its output is filtered against the original XPath before being returned to the user. GRAM pumps benefit significantly from the query batching performed by Pour. In particular, if a batch of queries contains separate queries resulting in GRAM jobs to the same host, those jobs are combined into a single GRAM invocation to minimize overhead.

A benefit of using the GRAM service is that the collection occurs under the user's grid identity, thus the executable runs with the user's permissions and the time spent computing 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 through the periodic mechanism, while specialized information that may only be of value 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 tagged with the grid identity of the submitting user and cached in the database, 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 queries.

4. Swim

Swim is a **S**oft**w**are **I**nformation **M**etacatalog built on top of the Pour framework. Swim consists of a spout with a set of drains and pumps that provide information about the software packages and the software files installed across the grid. Figure 1 shows sample instances and the flow of information through the periodic, on-demand, and user-specified aspects of Swim. 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 reported for Executable and Linking Format (ELF) executables and shared libraries, Java classes, shell scripts, and Perl and Python modules. Swim incorporates a user drain allowing users to specify their own software installations. The top right of figure 1 shows a sample instance of user-specified information, which is validated against the XSD of the user drain and, if valid, is added to the database.

4.1. Periodic Information

Swim uses a keyed drain for periodic information. System administrators can choose how often to gather software information based on their own detailed knowledge of system operations. The Swim script invoked by the keyed drain 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 the RPM, Perl, and Python package managers on multiple platforms. 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.

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 through the keyed drain mechanism. The bottom right of figure 1 shows a sample instance of the periodic information generated by Swim, which is validated against the secret key of the keyed drain using the supplied hash and, if valid, is added to the database.

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 elements are the commands for retrieving the names of all installed pack-

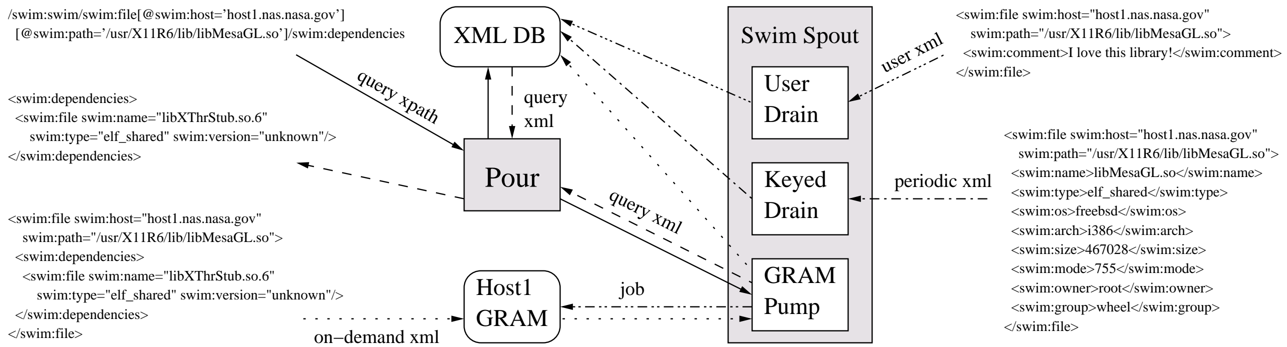

Figure 1. Swim information flow

ages and for listing the detailed information about a specific package. A parser must be written to gather relevant 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 provides several forms of on-demand information through a set of GRAM pumps. The first pump gathers dependency information (prefix "/swim/file/dependency") for a specific file (restriction "/swim/file[@path]") on a specific host (restriction "/swim/file[@host]"). This analyzer is based on the dependency analyzer developed in previous IPG Naturalization Service work [14] and gathers the software that is required for the correct execution of ELF executables and libraries, Java classes, and Perl and Python modules. In the worst case, this analysis can take several minutes, thus is not suitable for incorporating into the periodic routines, which examine potentially tens of thousands of files. Section 5.2 describes one of the uses of this pump in more detail. The top left of figure 1 shows a sample query used to request the dependencies of the library "/usr/X11R6/lib/libMesaGL.so" on host "host1.nas.nasa.gov". If the XML database cannot satisfy the query, it is routed to the GRAM service on the appropriate host by the GRAM pump, which runs a script to generate the information. The raw information, which may be a superset of what was requested, is added to the database before being filtered and passed back to the user.

The second pump gathers the same information gathered by the periodic routines (prefix "/swim/file") about a specific file (restriction "/swim/file[@path]") on a specific host (restriction "/swim/file[@host]"). This pump allows users to add personal software installations, which are not gathered during periodic collection, into Swim without the need to manually write XML documents.

The third pump is an experimental pump for locating a given Perl module using the Comprehensive Perl Archive Network (CPAN). The idea of this pump is that even if a specific module cannot be located anywhere on the grid, it can still potentially be located in an external internet repository. This pump runs a Perl script based on Perl's CPAN module to locate sources (prefix "/swim/file") for a Perl module (restriction "/swim/file[type='perl']") with a given name (restriction "/swim/file[name]"), from which a set of source URIs is constructed, formatted in XML, and returned to Swim. The eventual goal is to develop a comprehensive set of pumps 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., which will be extremely valuable in automatically establishing execution environments as described in section 5.2.

4.3. Performance

Table 1 shows Swim periodic information results obtained over a small grid testbed consisting of 12 systems. The results include the average per system of the number of packages, the number of files, the XML document size, the collection time, and the 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 acceptable time given their fairly large size.

Table 2 shows a breakdown of the time taken by the OGSI layer, the Pour framework, the back-end database, and the Globus GRAM to satisfy different types of queries

[3] `http://www.rpmfind.net`
[4] `http://www.sunfreeware.com`
[5] `http://www.vex.net/parnassus`
[6] `http://www.exist-db.org`

Measure \\ Platform	FreeBSD	IRIX	Linux	Solaris
Systems	1	4	6	1
Avg. Packages	365	715	592	603
Avg. Files	7412	4722	9780	7634
Avg. XML Size	3.42 MB	2.35 MB	4.67 MB	3.68 MB
Avg. Collection	196 sec	534 sec	686 sec	1209 sec
Avg. Insertion	14.4 sec	9.3 sec	21.7 sec	17.5 sec

Table 1. Swim periodic results

when Pour was hosted on a 2.4 GHz Pentium 4 running Linux with 512 MB of memory. An eXist XML database was used for the back-end, which was already filled with the periodic information collected in table 1. The table shows results in seconds for three types of queries covering both single and batch cases. The queries were chosen such that they resulted in either (1) data cached in the local database, (2) data collected by a GRAM pump, or (3) data collected by a Pour pump. As can be seen, single queries are very fast when the information is cached locally and also fairly fast when collected using a Pour pump. The GRAM pump, however, introduces more than an order of magnitude slowdown. In the batch case, 10 different queries were chosen such that they would be processed on the same GRAM host or by the same Pour URI if not cached in the database. All three query types benefit from batching with GRAM pump queries benefiting the most since all of the queries could be processed with just a single GRAM invocation.

Query \\ Component	OGSI	Pour	DB	GRAM	Total
Cached	0.27	0.02	0.57	-	0.86
GRAM Pump	0.29	0.11	0.60	11.7	12.7
Pour Pump	0.47	0.25	1.21	-	1.93
(10) Cached	0.58	0.06	6.19	-	6.83
(10) GRAM Pump	0.60	0.22	6.38	11.7	18.9
(10) Pour Pump	0.64	0.26	12.2	-	13.1

Table 2. Swim query breakdown (secs)

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.

5.1. IPG Resource Broker

The IPG Resource Broker is a grid service for selecting and ranking resources based on user-specified constraints and preferences. The Resource Broker is built using Surfer [15], 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 both find the compute resources that have a particular piece of software installed as well as finding the exact path to that software. Figure 2 shows an example Resource Broker request to find a compute resource running Linux with at least 128 free CPUs and a version 1.3.1 ELF executable named "java" on the same host that is both 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 2. Resource Broker request

Figure 3 shows the Swim query that is automatically constructed by the Surfer provider to support this request. The restriction on the host names between resources is handled by the framework itself as it involves information from two separate providers.

```
/swim:swim/swim:file[swim:name = 'java' and swim:type = 'elf' and
  swim:version = '1.3.1' and swim:mode mod 10 = 5]
```

Figure 3. Resource Broker Swim query

5.2. IPG Naturalization Service

The IPG Naturalization Service [14] 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.

Figure 4 shows the Swim queries used to extract the dependency and location information required by the Naturalization Service. Note that the first query will trigger the dependency GRAM pump if the information has not already been cached in the database. Since many files share the

same dependencies (e.g. libc) and users utilize many of the same applications and libraries, much of the information collected by one user can immediately benefit many other users without paying any additional GRAM overhead. Since Swim allows user-specified information, users can add information about their own personal software installations, which will be utilized by the Naturalization Service whenever possible.

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 locations of a file with a given name, type, and version:
   /swim:software/swim:file[swim:name='name2']
   [swim:type='type2'][swim:version='version2']

Figure 4. Naturalization Service Swim queries

Once all of the external repository pumps 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.).

6. Conclusions and Future Work

This paper has described a new information service framework called Pour for **P**eriodic, **O**n-Demand, and **U**ser-Specified Information **R**econciliation, and a new software information service for grid computing called Swim, the **S**oft**w**are **I**nformation **M**etacatalog. Pour has been specifically designed to accommodate high volume, low frequency information updates with a novel mechanism for automatically gathering information on-demand when necessary based on the contents of XPath queries posed by users. Information types can be added using *spouts*, each of which may have its own set of push and pull-based information sources represented by *drains* and *pumps*, respectively. Drains support both periodic updates based on grid service data, keyed hashes, and/or embedded Java objects as well as user updates validated against XML schemas. Pumps support on-demand updates via Pour hierarchies for scalability as well as based on the Globus GRAM, which can be used to construct self-populating information services without the need for additional continuously running processes. This flexibility allows Pour to be used in a variety of configurations including a general-purpose grid-enabled database, a metadata catalog, a single resource information service, a multiple resource aggregating information service, as well as its full form with information unified across periodic, on-demand, and user-specified sources.

Swim provides a new set of information for the grid that its currently lacks. Namely, it provides extensive information about the software installed on all grid resources, which is a critical component of seamless computing across multiple systems and organizations. Swim supports true automatic software discovery based on the tools used by systems administrators to install software. Two applications have already been developed that take advantage of this information. The IPG Resource Broker allows users to select computational resources with specific software installed as well as determining the specific location of that software on the resource. The IPG Naturalization Service automatically establishes the execution environment for a user application by locating all of the dependencies of that application and installing them if necessary.

There are several directions for future research. XQuery syntax will be investigated to determine the feasibility of adding another alternative on-demand capability. Additional package managers such as the Globus Packaging Toolkit will be incorporated into the periodic collection scripts. Additional on-demand pumps such as computing the MD5 hash of a given file to verify integrity will be added. Finally, A GSI-SSH pump with functionality similar to the GRAM pump will be investigated to determine if a lower overhead can be achieved for on-demand queries.

References

[1] 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.

[2] 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.

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

[4] Cook, A. et al.: R-GMA: An Information Integration System for Grid Monitoring. 11th Intl. Conf. on Cooperative Information Systems, Nov. 2003.

[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] Fallside, D.C. (ed.): XML Schema Part 0: Primer. W3C Recommendation, May 2001. Available at `http://www.w3.org/TR/xmlschema-0`.

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

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

[10] 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.

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

[12] Hastings, S., Langella, S., Oster, S., Saltz, J.: Distributed Data Management and Integration: The Mobius Project. Global Grid Forum Semantic Grid Applications Wkshp., Jun. 2004.

[13] 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.

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

[15] 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.

[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`.

[18] Smith, W., Hu, C.: An Execution Service for Grid Computing. Technical Report NAS-04-004, NASA Advanced Supercomputing Division, Apr. 2004.