Ontology-Based Grid Index Service for Advanced Resource Discovery and Monitoring

Said Mirza Pahlevi and Isao Kojima

National Institute of Advanced Industrial Science and Technology (AIST),
Grid Technology Research Center,
Tsukuba, Ibaraki 305-8568, Japan

Abstract. This paper proposes a framework for advanced Grid resource discovery and monitoring. In the framework, a service's state data are mapped into ontologies so that service owners may enrich them with semantic and other useful data, while keeping the state data unchanged. The Index Service, based on the OGSA framework, aggregates and maintains the ontology data, and allows users to query the data by using ontology-based query languages. The Index Service also provides a Continual Query mechanism that enables users to submit ontology queries as subscription messages and to be notified when the query results change. This paper also proposes an automatic ontology update mechanism, to keep the ontology data up-to-date.

1 Introduction

The Open Grid Services Architecture (OGSA) [1] constitutes a conceptual framework for Grid computing that is based on Web services concepts and technologies. It provides standard mechanisms for discovering Grid service instances. The mechanisms define a standard representation for information about Grid service instances, denoted as *serviceData*. Globus Toolkit 3.2 (GT3.2)[1] [3] offers a Base Service, known as the *Index Service*, that provides the functionality within which serviceData can be collected, aggregated, and queried. Clients access the aggregated serviceData by using either of two mechanisms, *findServiceData* (a pull operation) or *subscription-notification* (a push operation). Clearly, the Index Service offers users considerable help with respect to resource discovery, selection, and monitoring.

The elements of serviceData (SDEs) are defined in a service's definition of the service interface, by means of an XML schema. As a result, serviceData constitutes a structured document but imposes no *semantic constraints* on the meaning of the document. Because it has no semantic constraints, serviceData

[1] GT3.2 implements the Open Grid Service Infrastructure (OGSI), which addresses detailed specifications for OGSA. Recently, the OGSI has been refactored to the WS-Resource Framework (WSRF) [2]. Since the WSRF retains, essentially, all of the OGSI concepts, the framework proposed here can easily be adapted to the WSRF.

P.M.A. Sloot et al. (Eds.): EGC 2005, LNCS 3470, pp. 144–153, 2005.

cannot easily be shared across applications or domains, nor can it easily be processed by automated agents/tools. The lack of semantic constraints also makes it difficult to implement automated reasoning, which could improve resource discovery and selection. For example, suppose there are Grid Data Service Factories (GDSFs)[2] that provide access to relational databases that contain, in turn, information about computer-related books. Each factory provides access to one relational database, and each database covers one of the following topics: compilers, firewalls, and biometrics. To facilitate resource discovery, the GDSFs provide a *databaseSchema* SDE that contains logical schema of all tables existing in the database. Since the SDE contains no semantic information, it would be difficult to perform automatic resource discovery and selection. For example, it would be difficult to automatically find factories that provide access to tables that have a column containing a specific domain value, such as a book's title. It would be difficult to find factories that provide access to databases containing books that address a specific topic such as biometrics, or a more general topic such as computer security, a topic that is related to both biometrics and to firewalls.

In order to enrich serviceData with semantic constraints, one could directly modify the SDE definition in the service specification. However, that would not constitute a suitable approach because of the following. First, the modification would affect all clients that refer to the serviceData. Second, adding "nonstandard" elements to the SDE definition would require communicating the new elements to clients and, currently, there is no efficient way to publish the semantics of new elements to clients. Third, the user may not have the access privileges to update the service specification or to recompile and/or redeploy the service.

In GT3.2, the subscription-notification can be performed only for the entire SDE. An Index Service client cannot specify a SDE's part, or element of interest, in order to receive notification only when that part changes. The client is forced to accept notification messages pertaining to the entire SDE, regardless of whether the monitored value, which may be a small part of the SDE, has changed.

This paper proposes a framework for creating, maintaining, querying, and monitoring *semantic serviceData/SDEs*. Some domain-specific ontologies are defined, and SDE values of services are mapped onto the property values of the ontologies. In the framework, each service has a *Service Data Provider* (SDP), which stores the service's semantic serviceData in an SDE. The Index Service aggregates the semantic serviceData by subscribing to the SDE and storing the serviceData in an ontology repository. To keep the semantic serviceData up-to-date, an automatic ontology update mechanism is proposed. The mechanism makes use of the subscription-notification mechanism and stores access paths to the original SDE values in the semantic serviceData. Index Service clients formulate queries with an ontology-based query language for information discovery and exploration of the serviceData's semantic parts.

[2] GDSFs are the part of the OGSA-DAI software [4] that implements the Grid Data Service Specification [5].

The Index Service also provides a *Continual Query* (CQ) mechanism for advanced resource monitoring. A CQ is a standing query that monitors the update of interest and returns results whenever the update has satisfied a specified condition. With the mechanism, a client can send ontology queries as subscription messages and receive (new) query results whenever monitored values have changed.

The rest of the paper is organized as follows. Section 2 briefly describes some semantic web technologies. Section 3 describes related work. Our proposed framework and its implementation are presented in Section 4. We conclude our paper in Section 5.

2 Resource Description Framework (RDF) and Ontology

The Resource Description Framework (RDF) [6] is a W3C Recommendation that is particularly intended for representing metadata about Web resources. RDF statements, also called *triples*, consist of three parts: *subject*, which identifies the item the statement is about; *predicate*, which identifies the property of the item that the statement specifies; and *object*, which identifies the value of that property. The RDF Schema (RDFS) extends the RDF standard by providing the means to specify vocabularies/terms used in RDF statements. To do this, the RDFS pre-specifies a body of terminology, such as `Class`, `subClassOf`, and `Property`, which forms a basis for building a hierarchical structure.

RDF and RDFS standards are widely used to build ontologies. An ontology (or '*shared understanding*') consists of explicit formal specifications of terms in the domain, and the relations between them. An ontology allows people or software agents to share a common understanding of the structure of information.

For effective storage and query of ontologies, use of a high level query language is essential. The numerous existing ontology repositories include Jena [7] and Sesame [8]. For querying ontologies, the repositories support query languages such as RDQL [9] and SeRQL [10].

3 Related Work

Ontology-based Matchmaker (OMM) [11] is an ontology-based resource selector for solving resource matching in the Grid. It consists of three components: *ontologies*, used for expressing resource advertisements and job requests; *domain background knowledge*, which captures additional knowledge about the domain; and *matchmaking rules*, which define the matching constraints based on the ontologies and background knowledge. In OMM, resource providers periodically advertise their resources to a matchmaker by sending advertisement messages, based on a resource ontology. Requesters formulate a job request to the matchmaker, basing it on a request ontology. On receiving a job request, the matchmaker activates the matching rules to find a list of potential matches.

Our work differs from the OMM, in the following aspects. First, the OMM requires resource providers to construct advertisements and to send them, peri-

odically, to the matchmaker, whereas our system uses an automatic service state mapping and ontology update operation. Second, OMM uses different ontologies for resource advertisements and job requests, while ours does not. Using different ontologies for the two kinds of processes results in the need to define rules that match advertisements and job requests. As a result, updating ontology vocabularies results in a modification of the matching rules. In addition, the OMM does not allow easy use of other rule-based engines, because the matching rules and background knowledge must be transformed into the engine's rules. By contrast, our system can easily use the existing ontology repositories and explore the power of given ontology-based query languages. Third, the OMM lacks a CQ mechanism, so the requestor cannot easily and effectively monitor the resource advertisements.

Several approaches have already been suggested for adding semantics to the Web service standard for improved service discovery. METEOR-S [12] is a framework for semi-automatically marking up Web service descriptions with ontologies. It provides algorithms to match and annotate WSDL files with relevant ontologies. By contrast, UDDI-M^T [13] deals with Web service directories. It enables users to specify metadata for the information stored in the directories. The metadata are stored (locally) in a Jena repository and can be queried using RDQL. UDDIe [14] also adds metadata to the directories data but with a service leasing mechanism. These systems do not deal with the service state of Grid services but, rather, with Web service standards.

Continual Query for Web scale applications has been studied in the literature. The goal is to transform a passive web into an active environment. OpenCQ [15] allows users to specify the information they want to monitor by using an SQL-like expression with a trigger condition. Whenever the information of interest becomes available, the system immediately delivers it to the users. The rest of the time, the system continually monitors the arrival of the desired information and pushes it to the users as it meets a threshold. NiagaraCQ [16] goes further, by grouping CQs that have similar structures so they can share common computation. Our CQ mechanism is different in that it monitors ontology instances/data[3]. Hence, our system uses RDQL to formulate CQs. Furthermore, the mechanism is adapted to the Grid environment because it uses the OGSA subscription-notification mechanism to deliver the CQ execution results.

4 Proposed Framework

4.1 General and Service State Ontologies

To add semantic information to serviceData, we map the SDE values into ontology property values. Ontologies used in this framework can be categorized

[3] The term 'ontology instance' corresponds to the RDF (facts), while the term 'ontology' (without instance) corresponds to the RDF schema.

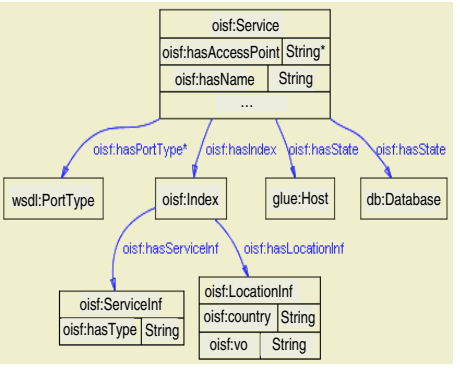

Fig. 1. General Ontology

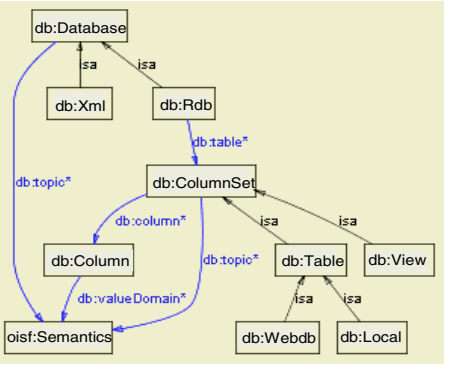

Fig. 2. Database Service Ontology

into two types: *General Ontology* and *Service State Ontology*. The former is for service-general information, and the latter is for service-specific information.

Fig. 1 shows part of the General Ontology. The part shown defines certain classes and properties as including general service information, such as service location (country and VO names), service types (e.g., computing and database service), service portType, service access point(s), and service name. Note that the *oisf:hasState* property refers to Service State Ontology instances. Currently, we define two Service State Ontologies: Database Service Ontology (shown in Fig. 2) and Computing Service Ontology. The Database Service Ontology is based on a database logical schema defined by the CIM-based Grid Schema Working Group of GGF [17]. The Computing Service Ontology is based on the GLUE schema defined by the EU-DataTAG and US-iVDGL projects [18].

4.2 SDE Value Mapping and Semantic ServiceData Creation

SDE values (in serviceData) are mapped into property values of a Service State Ontology. To this end, each SDE value is associated with a specific class in the ontology, called the `SDEInfo` class. This class has the following properties.

- *oisf:value*, which stores the SDE value.
- *oisf:valueState*, which stores the characteristics of the SDE value ('*static*' or '*dynamic*'). This property is used to efficiently update the *oisf:value* property value when the corresponding SDE value changes (see Section 4.3).
- *oisf:definedIn*, which refers to a *wsdl:portType* class instance.
- *oisf:hasSDEMapping*, which refers to an `SDEMapping` class instance. This class instance stores the access path information of the SDE value, such as the SDE name, the XPath expression to retrieve the value from the SDE, and the XML namespace mapping used in the XPath expression.

Fig. 3 shows the instances of `db:Rdb`, `db:Driver` (i.e., `SDEInfo` class), and `oisf:SDEMapping` classes defined in the Database Service Ontology. `db:Driver_`

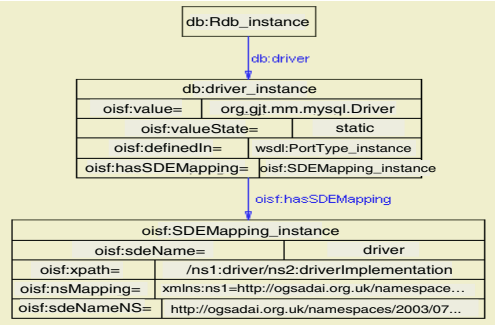

Fig. 3. SDE mapping

Algorithm: Maintain *semanticData* SDE.
Input: Ontology instances (semantic serviceData), *inst*.
Method:
1: Put *inst* into *semanticData* SDE and publish the SDE to Index Service;
2: Extract SDE names contained in Service State Ontology instances ($\in$ *inst*);
3: Subscribe to the extracted SDEs;
4: On receiving an SDE update message, *mes*,
5: $sName \leftarrow$ extract SDE name from *mes*;
6: Get `SDEInfo` instance ($\in$ *inst*) where *value*(`SDEInfo`.*oisf:valueState*) ='dynamic';
7: For each obtained `SDEInfo` instance, `sInfo`,
8: `sMapping` $\leftarrow$ get its `SDEMapping` instance ($\in$ *inst*);
9: If *value*(`sMapping`.*oisf:sdeName*) = *sName*,
10: *xpath* $\leftarrow$ get XPath expression from `sMapping`;
11: *val* $\leftarrow$ apply *xpath* to *mes*;
12: `sInfo`.*oisf:value* $\leftarrow$ *val*;

Fig. 4. *semanticData* SDE maintenance algorithm

`instance` corresponds to the *driver* element content in the serviceData of a database service (i.e., GDSF). It stores a driver implementation value in the *oisf:value* property. The XPath expression in `oisf:SDEMapping_instance` specifies how to retrieve the value from the *driver* element.

Ontology instances (semantic serviceData) of a service are created as follows. General Ontology instances are created manually/semi-automatically by a resource/service owner, using ontology editors such as Protëgë [19]. Service State Ontology instances are created, largely automatically, by the use of a simple parser. The parser reads a mapping file that maps each SDE value to a corresponding `SDEInfo` class in the ontology.

4.3 Service Data Provider (SDP)

To hold ontology instances, each service instance is associated with a Service Data Provider (SDP). An SDP is a transient service that has a special SDE, known as *semanticData*, to store semantic serviceData of the represented service.

Fig. 4 shows an algorithm used by an SDP to automatically maintain its *semanticData* SDE. Note that *value*(`inst`.*oisf:prop*) denotes the value of property *oisf:prop* of class instance `inst`. On receiving ontology instances, an SDP stores the instances into the *semanticData* SDE and publishes the SDE to Index Service (line 1). Publishing the SDE to Index Service makes the SDE available to be queried by clients. The SDP then extracts all distinct SDE names contained in the Service State Ontology instances (line 2). This is accomplished by getting *oisf:sdeName* property values from `SDEMapping` class instances. Next, the SDP sends subscription messages to the (represented) service in order to subscribe to the extracted SDEs (line 3). When the service updates an SDE value, the SDP receives a notification message containing the updated SDE (line 4). The SDP then extracts the SDE name from the message (line 5).

Next, the SDP updates the stored ontology instances (lines 6–12). To do this, it first gets all `SDEInfo` class instances for which the *oisf:valueState* property values are 'dynamic' (line 6). Then, for each obtained `SDEInfo` instance, the SDP gets the `SDEMapping` instance referred by the `SDEInfo` class instance (i.e., *value*(`SDEInfo`.*oisf:hasSDEMapping*)) (line 8). If the `SDEMapping` instance corresponds to the SDE contained in the notification message, then the SDP gets the XPath expression from the mapping instance (line 10), applies the expression to the notification message (line 11), and finally puts the obtained result/value[4] into the *oisf:value* property of the `SDEInfo` instance (line 12).

An SDP will be destroyed when the represented service stops. Before the SDP is destroyed, an unpublish message is sent to the Index Service to remove the service's semantic serviceData from the Index Service's repository.

4.4 Ontology-Based Grid Index Service (Ont-GIS)

Repository Maintenance. The Index Service (Ont-GIS) has an ontology repository, which it allows clients to query by using an ontology-based query language. Queries are formulated based on the General and Service State Ontologies. We use Jena [7] as the ontology repository and RDQL [9] as the query language.

Fig. 5 shows the Ont-GIS architecture. Ont-GIS collects *semanticData* SDEs from several SDPs and stores their contents in the repository. The content of each *semanticData* SDE is stored as a Jena persistent model[5]. On receiving a publish request from an SDP, the Subscription Manager pulls the *semanticData* SDE from the SDP and stores the content by passing it to the Update Manager. The Subscription Manager then subscribes to the *semanticData* SDE and waits for a notification change message.

On receiving a notification change message, the Subscription Manager passes *update data* to the Update Manager and CQ Manager. Based on the update data,

[4] Since the XPath expression always points to a specific SDE value, the result of its application is always singular.

[5] A Jena Model is a (Java) class instance that provides operations to manipulate ontology instances stored in the model. A model backed by a database engine is called a persistent model.

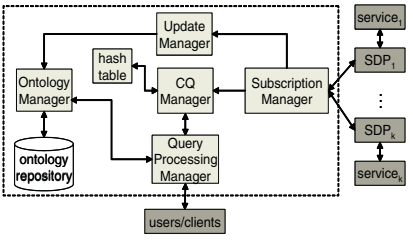

Fig. 5. Ont-GIS Architecture

the Update Manager performs an ontology update operation with the assistance of the Ontology Manager. The CQ Manager executes CQs that are related to the update data. Update data consists of an (RDF) resource name, a model ID, and the updated value. The resource name identifies a resource (i.e., `SDEInfo` class instance) for which the *oisf:value* property value has changed, and the model ID identifies the model in which the resource is stored.

Query Execution. An RDQL query is executed in a simple manner. On accepting a query, the Query Processing Manager executes the query by the help of the Ontology Manager and returns the execution results to the users/clients.

On the other hand, execution of a CQ requires two steps. Fig. 6 shows the algorithms used by the CQ Manager. The CQ Manager has a *hash table* to store CQs and query execution information. Each location in the hash table is a *set* so that it can store more than one value. $InsertH(k, val)$ is a hash function that inserts value val, based on key k. $LookupH(k)$ retrieves an entry (a set) from the hash table based on key k.

The query subscription process (left algorithm) constitutes the first step in CQ execution. A client sends a query subscription request to the Query Processing Manager, which, in turn, passes the request to the CQ Manager. The CQ Manager then creates a *unique SDE* in Ont-GIS's serviceData (line 1), executes the CQ (which is included in the request) with the help of the Query Process-

Algorithm: Register a CQ.
Input: A CQ, cq.
Output: An SDE name.
Method:
1: Create a unique SDE, $uSDE$;
2: $res \leftarrow$ execute cq;
3: $mSet \leftarrow$ IDs of models that match cq;
4: For each model ID $m \in mSet$,
5: $infSet \leftarrow$ get monitored resources of m from res;
6: $InsertH(m, \langle cq, infSet, uSDE \rangle)$;
7: Return $uSDE$;

Algorithm: Execute a CQ.
Input: resource rn and model ID m.
Method:
1: $tSet \leftarrow LookupH(m)$;
2: If $tSet$ is not empty,
3: $mtSet \leftarrow$ all tuple from $tSet$ s.t. the monitored resource set contains rn;
4: For each tuple $t \in mtSet$,
5: $res \leftarrow$ execute CQ contained in t;
6: Put res into SDE mentioned in t;

Fig. 6. CQ registration and execution algorithm

ing Manager (line 2), and gets Jena model IDs from the execution results (line 3). Next, the CQ Manager updates its hash table. For each matched model ID, it inserts a tuple into the hash table, with the model ID as a key (lines 4–6). The tuple consists of the given CQ, the monitored resource name set extracted from the CQ execution results for the model, and the created unique SDE name. A monitored resource is a resource (i.e., `SDEInfo` class instance) for which the *oisf:valueState* property value is 'dynamic'[6]. Finally, the CQ Manager returns the unique SDE name to the client (line 7). A client receiving the SDE name will (immediately) subscribe to the SDE to receive the CQ execution results.

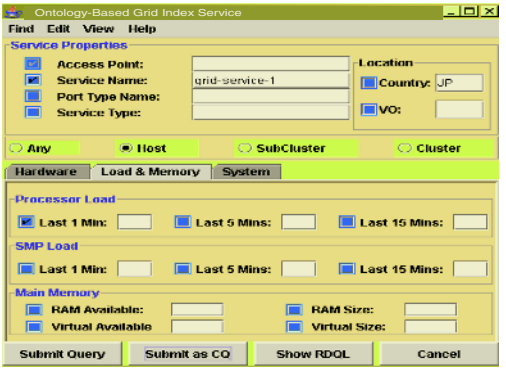

Fig. 7. A GUI for clients

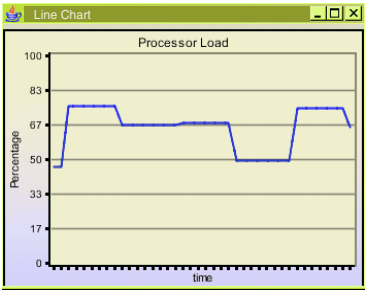

Fig. 8. Line chart of processor load

The second step is that of the CQ query execution process (right algorithm). When the CQ Manager receives the *updateData* (i.e., resource name and model ID) from the Subscription Manager, it uses its hash table to search for CQs that monitor the changes of the given model (line 1). If it finds any, the CQ Manager then gets all tuples, where the monitored resource set contains the given resource name (line 3). Finally, the CQ Manager executes all CQs contained in the tuples and puts the execution results into the corresponding SDEs (lines 4–5). Since clients whose CQs are executed have subscribed to the SDEs, they receive the execution results through notification messages sent by Ont-GIS. It is important to note that because the execution results are delivered to the clients through the OGSA subscription-notification mechanism, Ont-GIS can also collaborate with the GT3.2's Index Service.

4.5 Implementation

We have implemented the proposed framework using Java. Currently, Ont-GIS is deployed in GT3.2's service container and is able to collect semantic serviceData from computing and database services. Fig. 7 shows a graphical user interface (GUI) that helps clients construct an RDQL query. The upper and lower search

[6] Since CQ always deals with dynamic SDE values, we assume CQ contains triple (?r, *oisf:valueState*, 'dynamic'), where ?r is a variable.

forms of the GUI correspond to the General and Computing Service Ontology, respectively. The GUI also allows clients to submit an RDQL query as either a pull or a push query (CQ). Fig. 8 shows a line chart of the execution of a CQ, which monitors the last-1-minute processor load of a resource of the type `Host`.

5 Conclusions and Future Work

Two clear benefits of enriching serviceData with semantic information are those of automatic resource discovery and improved search results. Besides allowing users to explore the semantic parts of the serviceData, Ont-GIS also provides the CQ mechanism for efficient and effective resource monitoring. This mechanism uses the OGSA subscription-notification mechanism that enables it to cooperate with the GT3.2's Index Service. We are now considering a distributed architecture for Ont-GIS. This reason for this is that the Grid is distributed by nature and, as such, always deals with a great number of resources.

References

1. Foster, I., et al.: The physiology of the grid: An open grid services architecture for distributed systems integration. Globus Project (2002)
2. http://www.globus.org/wsrf/.
3. http://www-unix.globus.org/toolkit/.
4. http://www.ogsadai.org.uk.
5. http://www.cs.man.ac.uk/grid-db/documents.html.
6. http://www.w3.org/RDF/.
7. http://jena.sourceforge.net/.
8. Broekstra, J., et al.: Sesame: A generic architecture for storing and querying RDF and RDF schema. In: Proc. of ISWC 2002. Volume 2342. (2003) 54–68
9. http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.
10. http://www.openrdf.org/doc/users/ch05.html.
11. Tangmunarunkit, H., et al.: Ontology-based resource matching in the grid—the grid meets the semantic web. In: Proc. of SemPG03. Volume 2870. (2003) 706–721
12. Patil, A., et al.: METEOR-S web service annotation framework. In: Proc. of the World Wide Web Conference. (2004) 553–562
13. Miles, S., et al.: Personalised grid service discovery. In: Proc. of 19th Annual UK Performance Engineering Workshop. (2003) 131–140
14. ShaikhAli, A., et al.: UDDIe: An extended registry for web services. In: Proc. of Workshop on Service Oriented Computing: Models, Architectures and Applications. (2003)
15. Liu, L., et al.: Continual queries for internet scale event-driven information delivery. Knowledge and Data Engineering **11** (1999) 610–628
16. Chen, J., et al.: NiagaraCQ: a scalable continuous query system for Internet databases. In: Proc. of the ACM SIGMOD Conf. (2000) 379–390
17. https://forge.gridforum.org/projects/cgs-wg.
18. http://www.cnaf.infn.it/ sergio/datatag/glue/.
19. http://protege.stanford.edu.