SEMANTIC GRID RESOURCE DISCOVERY IN ATLAS*

Zoi Kaoudi, Iris Miliaraki, Matoula Magiridou
Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece
{zoi, iris, matoula}@di.uoa.gr

Erietta Liarou
Dept. of Electronic and Computer Engineering
Technical University of Crete, Greece
erietta@intelligence.tuc.gr

Stratos Idreos
CWI
Amsterdam, The Netherlands
S.Idreos@cwi.nl

Manolis Koubarakis
Dept. of Informatics and Telecommunications
National and Kapodistrian University of Athens, Greece
koubarak@di.uoa.gr

Abstract We study the problem of resource discovery in the Semantic Grid. We show how to solve this problem by utilizing *Atlas*, a P2P system for the distributed storage and retrieval of RDF(S) data. Atlas is currently under development in project OntoGrid funded by FP6. Atlas is built on top of the distributed hash table Bamboo and supports pull and push querying scenarios. It inherits all the nice features of Bamboo (openness, scalability, fault-tolerance, resistance to high churn rates) and extends Bamboo's protocols for storing and querying RDF(S) data. Atlas is being used currently to realize the metadata service of S-OGSA in a fully distributed and scalable way. In this paper, we concentrate on the main features of Atlas and demonstrate its use for Semantic Grid resource discovery in an OntoGrid use case scenario.

Keywords: peer-to-peer networks, DHT, RDF, query processing, Semantic Web.

*This work is partially funded by FP6/IST project OntoGrid.

1. Introduction

For the Semantic Grid vision [15] to become a reality, *high quality of service* must be offered to users and applications at all levels of the Grid fabric. In this paper, we concentrate on high quality of service in the provision of *resource discovery* services in Semantic Grids. Resource discovery is an important problem in Grids in general, and Semantic Grids in particular. We discuss how to achieve *high-performance, scalability, resilience to failures, robustness* and *adaptivity* in the provision of resource discovery services in Semantic Grids, and especially in OntoKit, the Semantic Grid toolkit currently under development in project OntoGrid [24].

OntoGrid (`http://www.ontogrid.net`) is a Semantic Grid project funded by the Grid Technologies unit of the European Commission under the strategic objective "Grid-based systems for Complex Problem Solving" of the Information Society Technologies programme of FP6.

Our basic assumption in this paper is that Semantic Grid resources (e.g., machines, services or ontologies) will be annotated by RDF(S) metadata. Metadata pervades the Semantic Grid and is used to describe Grid resources, the environment, provenance and trust information etc. [15]. The Resource Description Framework (RDF) and RDF Schema (RDFS) are frameworks for representing information about Web resources. RDF(S) consists of W3C recommendations that enable the encoding, exchange and reuse of structured metadata, providing the means for publishing both human-readable and machine-processable information and vocabularies for semantically describing things on the Web. Although RDF(S) was originally proposed in the context of the Semantic Web, it is also a very natural framework for representing information about Grid resources. As a result, it is used heavily in various Semantic Grid projects e.g., ^{my}Grid (`http://www.mygrid.org.uk`) or OntoGrid.

We propose to view resource discovery in Semantic Grids as *distributed RDF query answering* on top of a P2P network of Grid resource *providers* and *requesters*. Our proposal complements well-known Grid information services such as MDS4 of GT4 in two ways:

- We offer service providers and service requesters expressive *semantics-based* data models and query languages (i.e., RDF(S) and RQL instead of XML and XPath).

- We implement resource discovery using techniques from P2P systems. This allows us to achieve *full distribution, high-performance, scalability, resilience to failures, robustness* and *adaptivity*. Related experimental work is presented in [26, 28, 27].

In the context of OntoGrid, our proposal is realized with the implementation of *Atlas*, a P2P system for the distributed storage and querying of RDF(S) metadata describing Semantic Grid resources.

The rest of the paper is organized as follows. Section 6 briefly discusses related work at the crossroads of Grid and P2P computing research. Section 3 gives a short description of the various components and protocols of Atlas. Section 4 shows how to use Atlas for service discovery in OntoKit. Finally, Section 5 concludes the paper.

2. Related Work

Our research can be understood to lie at the intersection of P2P and Grid computing. Although these computing paradigms have different origins and have been developed largely independently, there has been a lot of interesting work lately at the crossroads of these paradigms [13, 34, 11].

Previous papers that explore connections among Grids and P2P networks can be distinguished in the following categories:

1 General papers that discuss the similarities and differences of P2P and Grid systems pointing out important areas where more work is needed [13, 34, 11].

2 Papers where ideas from P2P computing are used in Grid systems. Here, we can further differentiate as follows:

   (a) Works where Grid computing problems are given as a primary motivation, but the contributions are essentially in the P2P domain and can also be applied elsewhere. For example, [4, 23, 7] consider attribute-value data models that can be used to describe Grid resources (e.g., by specifying the CPU power, disk space capacity, operating system and location of a computer) and show how to evaluate queries in these models on top of DHTs (e.g., I am looking for an idle PC that runs Linux and has CPU $\geq$ 3GHz).

   (b) Works where P2P techniques are used to improve functionality in existing Grid systems e.g., resource discovery [20, 18, 19] and replica location management in Globus [8] or flocking in Condor [6].

   (c) Service-oriented application development frameworks that enhance existing frameworks for Web or Grid service computing [1, 16] with P2P protocols.

3 Papers where ideas from Grid computing are used in P2P systems. For example, [10] shows how to implement a P2P data integration framework using OGSA-DAI [2].

Our work should be classified in categories 2(b) and 2(c) above. Work with goals similar to ours that uses description logics instead of RDF(S) is reported in [17].

3. The P2P System Atlas

In Atlas, we use state of the art *distributed hash table (DHT)* technology [5] to implement a distributed system that will be able to scale to hundreds of thousands of nodes and to large amounts of RDF(S) data and queries. Nodes in an Atlas network are organized under the Bamboo DHT protocol [31]. Bamboo is a DHT based on Pastry [32] from where it takes the circular identifier space and the routing algorithms. Bamboo improves on Pastry by being able to withstand very dynamic changes in network membership i.e., it is resilient to churn [31]. Like most implementations of DHTs, Bamboo offers a very simple interface consisting of two operations: `put(ID, item)` and `get(ID)`. The `put` operation inserts an item with key `ID` and value `item` in the DHT. The `get` operation returns a pointer to the DHT node responsible for key `ID`. Our operations for storing data and querying Atlas, described below, are based on these simple operations offered by Bamboo.

Atlas nodes can enter RDF(S) data into the network and pose RQL queries. Two kinds of querying functionality are supported by Atlas: *one-time* querying and *publish/subscribe*. Each time a node poses a one-time query, the network nodes cooperate to find RDF(S) data that form the answer to the query. In the publish/subscribe scenario, a node can *subscribe* with a *continuous* query. A continuous query is indexed somewhere in the network and each time matching RDF(S) data is published, nodes cooperate to *notify* the subscriber.

The current implementation of Atlas (Atlas v0.6) supports a subset of the query language RQL [22] as we explain in Section 3.4 below. The query processing algorithm we use for one-time queries is an extension of the algorithm proposed in [9] for a smaller class of queries based on triple patterns [9]. Publish/subscribe scenarios in Atlas are handled using the algorithms in [28, 27] that are briefly discussed in Section 3.3 below but have not been fully implemented in Atlas v0.6. In the future, Atlas will also support the recently proposed RDF update language RUL for inserting, deleting and updating RDF metadata [30].

Atlas is used in OntoKit for realizing a fully distributed *metadata service*. A high level view of Atlas and the metadata service of OntoKit is shown in Figure 1.

3.1 RDF Documents and Queries in Atlas

Atlas nodes provide their data in the form of RDF documents [25]. These documents are decomposed into RDF triples that are indexed in various nodes

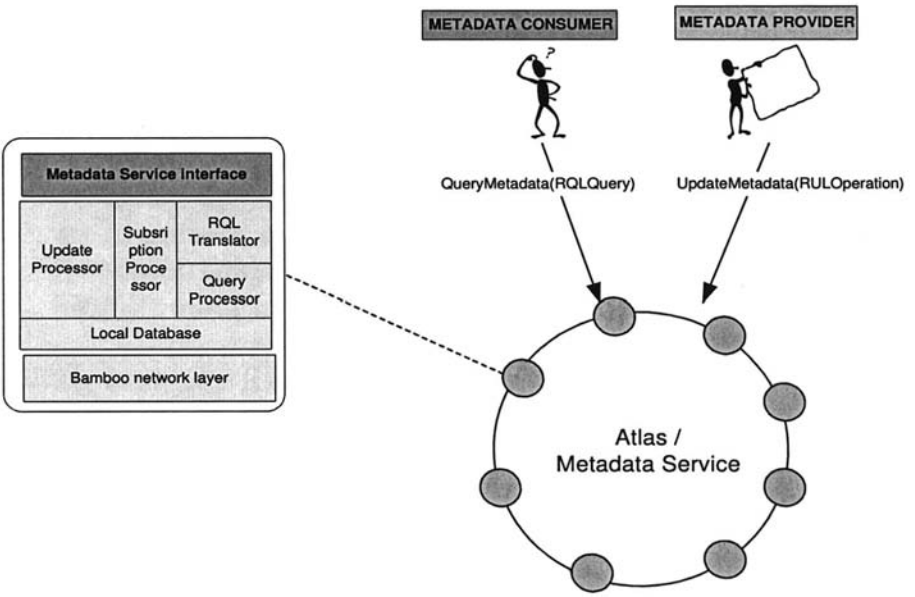

Figure 1. Atlas and the metadata service

of the network. A *triple* represents a statement about a domain and has the form $(subject, predicate, object)$ where $subject$ and $predicate$ are URIs and $object$ is a URI or a literal. We adopt the triple indexing algorithm presented in [9], where each triple is indexed on the DHT *three times*, once for its subject, once for its predicate and once for its object. For each of these storage operations we make use of the `put` operation provided by the Bamboo DHT using as key the subject, predicate or object value respectively. The key is hashed to create the identifier that leads to the appropriate node where the triple is stored.

Atlas supports internally the query language TPQL (*triple-pattern query language*) which allows the expression of *positive* (i.e., without negation) *conjunctive queries* where each conjunct is a *triple pattern*.

A *conjunctive query* q is a formula of the form

$$?x_1, \ldots, ?x_k : (s_1, p_1, o_1) \wedge (s_2, p_2, o_2) \wedge \cdots \wedge (s_m, p_m, o_m)$$

where $s_1, \ldots, s_m, p_1, \ldots, p_m$ are variables or URIs, $o_1, \ldots, o_m$ are variables, URIs or literals, $?x_1, \ldots, ?x_k$ are variables and $\{?x_1, \ldots, ?x_k\} \subseteq \{s_1, \ldots, s_m, p_1, \ldots, p_m, o_1, \ldots, o_m\}$. Variables will always start with the '?' character. The triple patterns $(s_1, p_1, o_1), \ldots, (s_m, p_m, o_m)$ are the *subqueries* of q. A query will be called *atomic* if it consists of a single conjunct.

The class of conjunctive queries can be used to express many interesting requests in P2P applications using RDF. For example, assume that a service

requester wants to discover a Web service for arranging the repair of a car. This request can be expressed as a conjunctive query as follows:

$$?x, ?y : (?x, hasServiceKeyword, \text{“}Cars\text{”}) \wedge$$

$$(?x, hasServiceKeyword, \text{“}Repair\text{”}) \wedge (?x, hasLocationURI, ?y)$$

3.2 One-Time Query Processing in Atlas

In this section, we describe the algorithm for one-time query processing in Atlas using terminology from relational databases. Each triple can be understood to be a tuple in a relation $TRIPLE(S, P, O)$ with attributes S for subject, P for predicate and O for object. Then, conjunctive queries are *select-project-join* queries over the database that consists simply of the relation $TRIPLE$. The exact query processing algorithm of Atlas is as follows.

Let n_1 be a node that wants to pose a conjunctive query q of the form introduced in Section 3.1. Node n_1 creates a message

$$queryRequest(id, triplePattern, restTriplePatterns,$$

$$partialResult, variables, returnAddress)$$

and sends it to the node with identifier id using the underlying Bamboo infrastructure. In this message, $triplePattern$ is the triple pattern of q which node n_1 chooses to be evaluated first[1], id is the identifier obtained by hashing one of the constants in triple pattern $triplePattern$, $restTriplePatterns$ is the list of remaining triple patterns of q, $partialResult$ is a relation for partial results (see below) which is initially empty, $variables$ is the list of answer variables of q, and $returnAddress$ is the IP address of node n_1.

When another node n_2 receives the above message $queryRequest$, it does the following. It first computes the bindings of the variables included in the given triple pattern by finding the triples in its local database that match $triplePattern$. These bindings form a new relation R with attributes the variables in question. If $partialResult$ is empty, then node n_2 assigns R to $partialResult$. Otherwise, n_2 computes the natural join of R and $partialResult$ (i.e., $partialResult \bowtie R$) and assigns it to $partialResult'$. Then, n_2 creates a new message

$$queryRequest(id', triplePattern', restTriplePatterns',$$

$$partialResult', variables, returnAddress)$$

[1]This choice is crucial depending on the metric one wants to optimize; in Atlas v0.6, we simply pick the first triple pattern/conjunct.

When this message is received by another node n_3, the same procedure is followed. These nodes join the relation R of the bindings they retrieve locally with the relation $partialResult$ and send a message to the next node. This procedure terminates in two possible ways. Either, the list $restTriplePatterns$ becomes empty or the relation $partialResult$ becomes empty. The latter means that the current triple pattern does not match with any triples stored locally, and thus relation R becomes empty and the join operation results in an empty relation. In both cases, a response with the results should be returned to node n_1 which issued the query. The field $returnAddress$ is used for this purpose; it remains unchanged throughout the whole procedure and refers to the IP address of node n_1.

The node n_m that determines that the query evaluation procedure is finished computes the bindings of the answer variables $?x_1, \ldots, ?x_k$. In order to do that, n_m computes the projection of relation $partialResult$ on the variables included in the list $variables$ and inserts the results in the relation $variableBindings$ i.e.,

$$variableBindings = \pi_{variables}(partialResult).$$

Then, n_m sends a response message $queryResponse(variableBindings)$ to node n_1, where $variableBindings$ is a relation with the answer to the query.

The key idea in the algorithm we described above is that we split a conjunctive query to the triple patterns that is consists of and evaluate each one at a different node of the network. In this way, we try to distribute the responsibility of answering a query to several nodes. Intermediate results flow through these nodes and finally the last one delivers the results back to the node that submitted the query. Notice that in order to determine which node will evaluate a triple pattern the algorithm uses *one* of the constants contained in it. Finally, the distributed query plan is created once, i.e., at the time that the query is submitted.

In [26], we propose an improved algorithm for the evaluation of conjunctive RDF queries on top of DHTs. In this algorithm, the distributed query plan is created *dynamically* by exploiting the values of matching triples found while processing the query incrementally. This time we use combination of constants in a triple pattern to determine which will be the node to evaluate it. By enriching the triple patterns with new values we have more combinations to use. In this way, this algorithm distributes the responsibility of evaluating a query to more nodes than the previous one. Our initial experiments show a significant improvement on load distribution but, on the other hand, there is an overhead in network traffic.

3.3 Publish/Subscribe in Atlas

In [28, 27], we propose two distributed algorithms for publish/subscribe on top of DHTs when publications are RDF triples and subscriptions are conjunctive multi-predicate queries.

In our algorithms, when a continuous query is submitted, it is *indexed* somewhere in the network and waits for triples to satisfy it. Each time a new triple is inserted, the network nodes cooperate to determine what queries are satisfied, compute their answers and create notifications for the subscribers. The case of conjunctive queries is an interesting one, since a single triple may *satisfy* a query q only *partially* by satisfying a subquery of q. In other words, more than one triples may be needed to answer a query. Moreover, since the appropriate triples do not necessarily arrive in the network at the same time, the network should "remember" the queries that have been partially satisfied in the past (e.g., by keeping intermediate results) and create notifications only when all subqueries of a given query are satisfied.

We could index queries to a globally known node or set of nodes, but this would eventually overload these nodes. In a P2P environment, we want as many nodes as possible to contribute some of their resources (storage, cpu, bandwidth, etc.) for achieving the overall network functionality. The resource contribution of each node will obviously depend on its capabilities, its gains from participating in the network, etc. In our work, we make the simplifying assumption that all nodes are altruistic, with equivalent capabilities, and, thus, can contribute to query evaluation in identical ways.

Let us now discuss the issues involved in publish/subscribe with conjunctive queries. We first consider an atomic query $q = (?s_1, p_1, ?o_1)$. We can simply assign q to the successor node x of $Hash(p_1)$ by using the constant part p_1 of the query. Triples that have predicate value equal to p_1 will be indexed to x too, where they will meet q. Assume now the atomic query $q' = (?s_2, p_2, o_2)$. We can index q' either to node $x_1 = Successor(Hash(p_2))$ or to node $x_2 = Successor(Hash(o_2))$. We prefer the second option since intuitively there will be more object values than predicate values in an instance of a given schema, which will allow us to distribute queries to a greater number of nodes. Another solution is to index q' to the node $x_3 = Successor(Hash(p_2 + o_2))$. We use the operator $+$ to denote the *concatenation* of string values. This is the best option because the possible combinations of predicate and object values will be greater than the number of object values alone, so this will lead to an even better distribution of queries.

The difficulty with arbitrary conjunctive queries is that they demand more than one conditions to be satisfied before the whole query can be satisfied. As an example, consider the query $q = q_1 \wedge q_2 \wedge q_3$. Our approach is to *split* the query to the subqueries that it consists of, and to index each subquery

separately. Then, three usually different nodes will be responsible for query processing regarding q. Each one will be responsible for a single subquery of q, e.g., nodes r_1, r_2 and r_3 will be responsible for q_1, q_2 and q_3 respectively. These nodes will form the *query chain* of q, denoted by $chain(q)$. Each one of these nodes will monitor the satisfaction of only the subquery that it is responsible for. To determine the satisfaction of q, we have to allow some kind of communication between these three nodes. In this way, as triples arrive and satisfy a subquery e.g., in node r_1, r_1 will forward *partial results* of q to r_2. Node r_2 will forward partial results that also satisfy the second subquery to r_3 and r_3 will realize that the whole query is satisfied and create a notification.

The first algorithm that we present in [28] creates a *single query chain* for each conjunctive query while the second one creates *multiple query chains* for a single query to achieve a better query processing load distribution. The first algorithm of [28] is essentially identical to the one-time query processing algorithm discussed in Section 3.2 except that, in the publish/subscribe case, it is executed in a reactive manner as matching triples arrive in the network. In [28], the two algorithms presented are experimentally evaluated for conjunctive *multi-predicate* queries (i.e., queries where the subject of all the triple patterns is the same variable $?s$ and predicates $p_1, \ldots, p_m$ are all constant). However, the general idea of these algorithms is easily extensible to support the full class of conjuctive queries as we show in the forthcoming paper [27].

3.4 The RQL-to-TPQL Translator

Atlas offers to users the ability to write queries in TPQL or in the well-known RDF query language RQL. RQL [22], which stands for RDF Query Language, is a declarative language which relies on a formal graph model that captures the RDF modelling primitives. The novelty of RQL lies in its ability to combine schema and data querying smoothly while exploiting the taxonomies of labels and multiple classification of resources. The syntax of RQL includes a set of basic queries (e.g. `Resource`, `SubClassOf()` etc.) as well as SQL-like `select-from-where` queries to iterate over RDF collections and introduce variables[2].

Consider the schema of Figure 2 which describes information about Web services in RDFS. This example is part of the core services data model used in project ^{my}Grid[3]. Suppose we want to find a Web service for arranging the repair of the car. What follows is an appropriate RQL query:

```
SELECT X
FROM {X}ns:hasServiceDescription{Y}
```

[2]RQL is implemented in ICS-FORTH's Suite `http://139.91.183.30:9090/RDF/`
[3]http://www.mygrid.org.uk

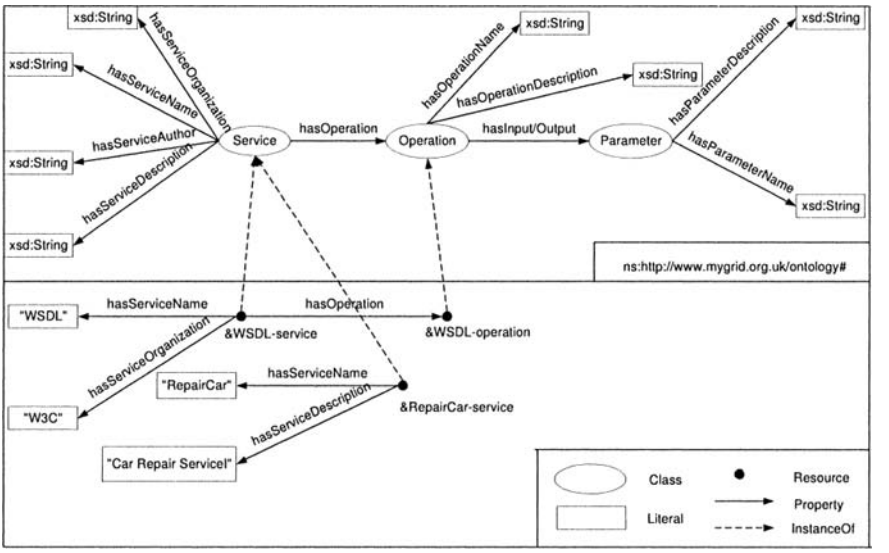

Figure 2. RDFS schema for Web Services

```
WHERE Y like "*car*"
USING NAMESPACE ns=&http://www.mygrid.org.uk/#ontology
```

In order to support RQL queries in Atlas, we have introduced a module responsible for mapping a query expressed in RQL to a query in TPQL, which is the query language supported internally by Atlas and described in Section 3.1. In Atlas v0.6, we do not support the full functionality of RQL but only *data queries with filtering conditions.*

Recall the RQL query presented earlier, about the discovery of service for arranging the repair of the car. The equivalent conjunctive query is the following:

$$?x : (?x,\ http://www.mygrid.org.uk/ontology\#hasServiceDescription,\ ?y)$$
$$\wedge\ ?y\ like\ "*car*"$$

To design the RQL-to-TPQL translator we have followed the RQL Interpreter architecture developed by ICS-FORTH [14] (see Figure 3). Our implementation has been done in Java using the Java Compiler Compiler (JavaCC) [3] parser generator.

The *syntax analyser* module receives as input a string, representing an RQL query, and returns the corresponding CNF syntax tree (if the query is valid). The syntax tree is passed to the *graph constructor* module, which creates a graph corresponding to the semantic representation of the query. These two modules are based on the code of RQL Interpreter. The *translator* module takes as input the syntax tree and graph of an RQL query and returns the

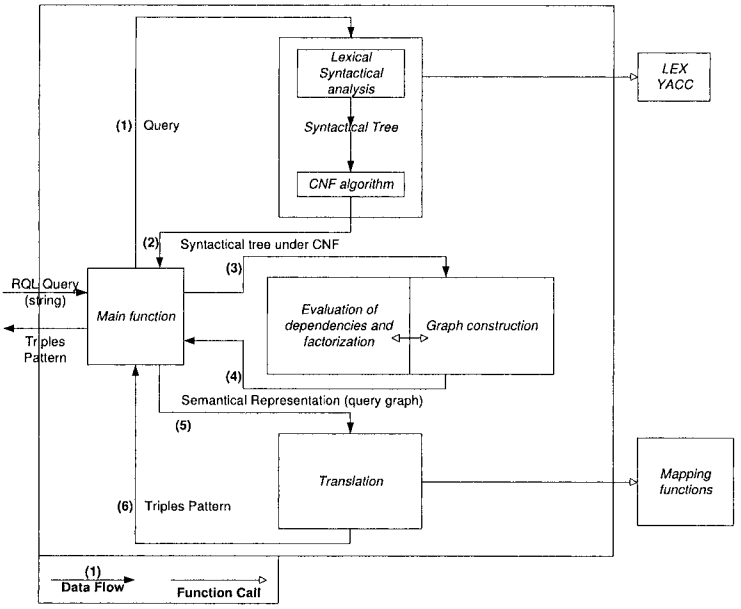

Figure 3. Module Architecture

equivalent expression in TPQL, as a list of triple patterns and constraints. It consists of a list of *mapping* functions, which implement the mapping rules between RQL and TPQL presented in [21]. The *main* module contains either a *JNI-client* and a standalone application for the management of the RQL query translator or directly creates the triple patterns data structures to be passed to the rest of the Atlas modules for query processing.

4. Atlas in Operation: Service Discovery in OntoKit

In this section, we show how Atlas can be used in OntoKit during service annotation and discovery [24]. The whole scenario is depicted in Figure 4.

OntoGrid is developing annotation technology for Grid services [33]; this technology is deployed as the *annotation service* of OntoKit. For the purposes of this section, it is also important to mention another service of OntoKit, the *ontology service* [12]. The current version of the ontology service provides a Grid interface to an RDFS store where RDFS ontologies are stored (e.g., *service ontologies* or *domain ontologies* etc.).

An ontology for services and various domain ontologies are needed in order to create a service annotation. Let us suppose that the annotation service chooses to search for an ontology about cars in order to annotate a car-repair service (the example comes from a car insurance use case studied in OntoGrid). The annotation service can pose an RQL query to the metadata service and get

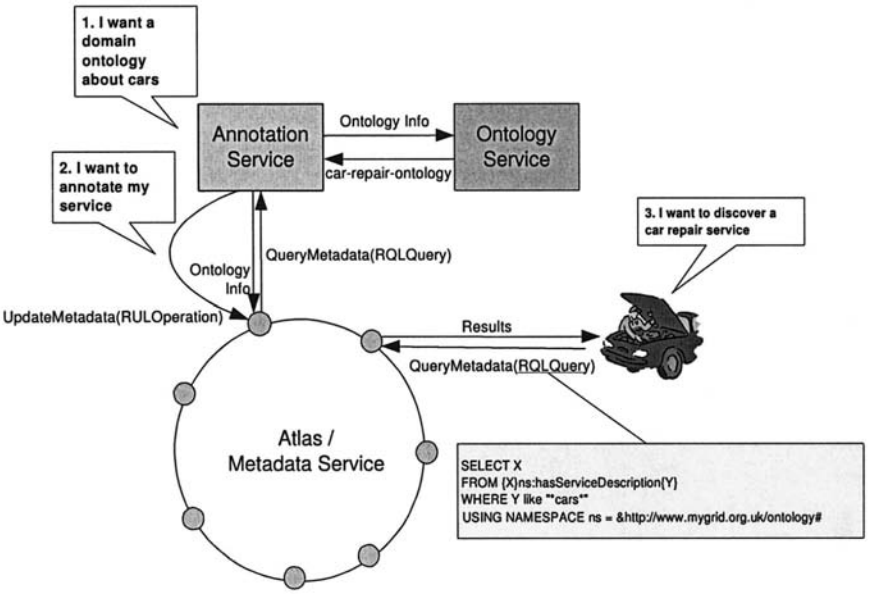

Figure 4. Using Atlas for Service Annotation and Discovery

information about such ontologies e.g., the location and description of a particular ontology – let us call it `car-repair-ontology`. After discovering information about `car-repair-ontology`, the annotation service can retrieve it from the ontology service.

If the annotation service does not know the ontology for annotating services, it has to search for such an ontology as well. An example ontology describing services that could be found in this case is the ^{my}Grid service ontology [29]. We should mention here that this step may be unnecessary if a specific service ontology has been selected for annotating services in OntoKit.

Using these ontologies, the annotation service can complete the service annotation process. The result of the annotation process will be stored in Atlas by calling the `UpdateMetadata` operation (see Figure 4). The ontology used for describing the service should have been stored previously in Atlas by calling the `StoreOntology` operation.

Let us suppose now that an OntoKit user wants to discover a service for repairing cars. This is accomplished by submitting RQL queries using appropriate service and domain ontologies (see Figure 4).

Finally, notice that after an annotation is stored, it might be necessary to be able to update it. An appropriate update operation can be expressed in RUL and executed in Atlas.

5. Conclusions

We have argued that resource discovery services for Semantic Grids can be made scalable, fault-tolerant, robust and adaptive, by exploiting distributed RDF query processing algorithms implemented on top of DHTs. We have discussed the implementation of our ideas in the system Atlas and its role in the Semantic Grid toolkit OntoKit. The implementation of Atlas was started at the Technical University of Crete and is currently continued at the National and Kapodistrian University of Athens. More information on the current version of Atlas is available in [21]. Although we have stressed performance issues, we have not provided any measurements or experimental results in this paper. Experimental results based on simulations can be found in [28] and more experimentation is underway [27, 26]. Finally, we expect to be able to analyse the performance of Atlas soon on real-world wide-area networks using the PlanetLab infrastructure.

References

[1] jxta. http://www.sun.com/software/jxta/.

[2] Open Grid Services Architecture Data Integration (OGSA-DAI). http://www.ogsadai.org.uk/.

[3] Java Compiler Compiler(JavaCC). https://javacc.dev.java.net/, 2004.

[4] A. Andrzejak and Z. Xu. Scalable, Efficient Range Queries for Grid Information Services. In *the second IEEE International Conference on Peer-to-Peer Computing (P2P2002)*, Linkoping, Sweden, 5-7 September 2002.

[5] H. Balakrishnan, M. Frans Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, 2003.

[6] A. Raza Butt, R. Zhang, and Y. Charlie Hu. A Self-Organizing Flock of Condors. In *Proceedings of Supercomputing Conference (SC)*, Phoenix, Arizona, November 2003.

[7] M. Cai, M. Frank, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. In *Proceedings of the 4th International Workshop on Grid Computing (Grid2003)*, 2003.

[8] M. Cai, A. Chervenak, and M. Frank. A Peer-to-Peer Replica Location Service Based on A Distributed Hash Table. In *the 2004 ACM/IEEE Conference on Supercomputing (SC2004)*, Pittsburgh, November 2004.

[9] M. Cai, M. R. Frank, B. Yan, and R. M. MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 2(2):109–130, December 2004.

[10] D. Calvanese, G. De Giacomo, M. Lenzerini, R. Rosati, and G. Vetere. Hyper: A Framework for Peer-to-Peer Data Integration on Grids. In *Proceedings of the International Conference on Semantics of a Networked World: Semantics for Grid Databases (ICSNW 2004)*, pages 144–157, 2004.

[11] J. Crowcroft, T. Moreton, I. Pratt, and A. Twigg. *The GRID2: Blueprint for a New Computing Infrastructure*, chapter Peer-to-Peer Technologies. 2004.

[12] M. Esteban Gutirez (ed), S. Bechhofer, O. Corcho, M. Ferndez-Lez, A. Gez-Perez, Z. Kaoudi, I. Kotsiopoulos, M. Koubarakis, M C. Suez-Figueroa, and V. Tamma. Specification and Design of Ontology Grid Compliant and Grid Aware Services. Deliverable 3.1 OntoGrid project.

[13] I. Foster and A. Iamnitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, February 2003.

[14] G. Karvounarakis. RQL. http://139.91.183.30:9090/RDF/RQL/, 2003.

[15] C. A. Goble and D. De Roure. The Semantic Grid: Myth Busting and Bridge Building. In *Proceedings of ECAI*, pages 1129–1135, 2004.

[16] A. Harrison and I. Taylor. Dynamic Web Service Deployment Using WSPeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16. Louisiana State University, February 2005.

[17] F. Heine, M. Hovestadt, and O. Kao. Towards Ontology-Driven P2P Grid Resource Discovery. In *5th International Workshop on Grid Computing (GRID 2004)*, pages 76–83, Pittsburgh, PA, USA, November 2004.

[18] A. Iamnitchi, I. Foster, and D. C. Nurmi. A Peer-to-Peer Approach to Resource Location in Grid Environments. In *Proceedings of the 11th Symposium on High Performance Distributed Computing*, Edinburgh, UK, August 2002.

[19] A. Iamnitchi, I. Foster, and D.C. Nurmi. A Peer-to-Peer Approach to Resource Discovery in Grid Environments. Technical Report TR-2002-06, University of Chicago, 2002.

[20] A. Iamnitchi and I. Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *International Workshop on Grid Computing*, Denver, Colorado, 2001. IEEE.

[21] Z. Kaoudi, I. Miliaraki, M. Magiridou, A. Papadakis-Pesaresi, E. Liarou, S. Idreos, S. Skiadopoulos, and M. Koubarakis. Deployment of Ontology Services and Semantic Grid Services on top of Self-organized P2P Networks. Deliverable D4.2, Ontogrid project, February 2006.

[22] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: A Declarative Query Language for RDF. In *Proceedings of the 11th International World Wide Web Conference*, May 2002.

[23] A. Kothari, D. Agrawal, A. Gupta, and S. Suri. Range Addressable Network: A P2P Cache Architecture for Data Ranges. In *Proceedins of the 3rd International Conference on Peer-to-Peer Computing (P2P'03)*, Linkoping, Sweden, 2003.

[24] I. Kotsiopoulos, S. Bechhofer, P. Alper, P. Missier, O. Corcho, D. Kuo, and C. Goble. Specification of a Semantic Grid Architecture. Deliverable 1.2, OntoGrid project.

[25] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Technical report, W3C Recommendation, 1999.

[26] E. Liarou, S. Idreos, and M. Koubarakis. Evaluating Conjunctive Triple Pattern Queries over Large Structured Overlay Networks. Submitted.

[27] E. Liarou, S. Idreos, and M. Koubarakis. Evaluating Continuous Conjunctive RDF Queries over Large Structured Overlay Networks. Manuscript in preparation.

[28] E. Liarou, S. Idreos, and M. Koubarakis. Publish-Subscribe with RDF Data over Large Structured Overlay Networks. In *Proceedings of the 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing (DBISP2P 2005)*, Trondheim, Norway, 28-29 August.

[29] P. Lord, P. Alper, C. Wroe, and C. Goble. Feta: A light-weight architecture for user oriented semantic service discovery. In *Proceedings of the 2nd European Semantic Web Conference (ESWC 2005)*, Heraklion, Crete.

[30] M. Magiridou, S. Sahtouris, V. Christophides, and M. Koubarakis. RUL: A Declarative Update Language for RDF. In *Proceedings of the 4rth International Semantic Web Conferece (ISWC2005)*, 2005.

[31] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conference*, 2004.

[32] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale- Peer-to-Peer Storage Utility. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Paltforms (Middleware 2001)*, November 2001.

[33] J. O. Segura, R. Benjamins, J. M. Gómez Pérez, J. Contreras, R. Salla, O. Corcho, R. González, G. Aguado de Cea, I. Álvarez de Mon y Rego, A. Pareja Lora, and R. Plaza Arteche. Specification and Design of Annotation Services. Deliverable D5.1, Ontogrid project, March 2005.

[34] D. Talia and P. Trunfio. Toward a Synergy Between P2P and Grids. *IEEE Internet Computing*, 7(4):94–96, 2003.