Personalised Grid Service Discovery

Simon Miles, J. Papay, Vijay Dialani, Michael Luck, Keith Decker, Terry Payne, and Luc Moreau

Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK
{sm, jp, vkd00r, mml, ksd, trp, L.Moreau}@ecs.soton.ac.uk

Abstract. We take a broad view that ultimately Grid- or Web-services must be located via personalised, semantic-rich discovery processes. We argue that such processes must rely on the storage of arbitrary metadata about services that originates from both service providers and service users. Examples of such metadata are reliability metrics, quality of service data, or semantic service description markup. This paper presents UDDI-M^T, an extension to the standard UDDI service directory approach that supports the storage of such metadata via a tunnelling technique that ties the metadata store to the original UDDI directory. We also discuss the use of a rich, graph-based RDF query language for syntactic queries on this data. Finally, we analyse the performance of each of these contributions in our implementation.

1 Introduction

Service discovery is a critical element in large scale, open distributed systems such as the Grid infrastructure, as it facilitates the dynamic identification of resources (abstracted as services). Providers may adopt various ways of describing their services, access polices, contract negotiation details etc. However, many resource consumers also impose their own selection policies on the services they prefer to utilise, such as provenance, derived quality of service, reputation metrics etc, and consequently they need to locally manage and augment service descriptions with additional information, i.e. *metadata*. To create dynamic organisations [6] that achieve specific goals, there is a need for discovery services and associated mechanisms that support the annotation of service descriptions.

The problem of service discovery is compounded by the plethora of different types of service directories that exist. Such services may include: *public* directories such as UDDI servers hosted by IBM or Microsoft; *specialised* directories such as the I3C bioinformatics service directory; *provider-specific* directories such as a directory of all the services hosted by a research institute; or even *local* directories such as the catalog of all the services that a laboratory is hosting for its own users. For convenience, access to different service directories (including the enforcement of selection policies), and homogenisation across different representations, etc. should be transparent to the user requesting a desired service.

Against this background, we have identified some key requirements within the myGrid project [9] that can enhance the process of service discovery by making the discovery process *personalised* to the user.

1. Users (not just service providers) need to be able to attach metadata to service instances registered in service directories.
2. Users cannot be expected to systematically query all service directories in a discovery process. Instead, federating a selected set of service directories should provide a single point of contact for the discovery process.
3. Users should be able to provide a semantic description of the task they want to achieve and the discovery should match these user requirements against semantic descriptions of the published services.

We will refer to the first two techniques as *syntactic*, whereas the third is *semantic*. In this paper, we focus only on the first technique—allowing users to attach metadata to published service instances. We will consider the other two techniques in future publications.

Since the types of personalised metadata that are required will naturally vary greatly between individuals, organisations, and scientific user communities, an abstract and highly flexible representation is required. By regarding and implementing metadata items as triples that specify a relation between a subject and an object, arbitrary metadata can be described and queried via graph-based search criteria. This can be achieved through the use of RDF (the W3C Resource Description Framework) [11], which underpins the Semantic Web effort [3].

Thus we have developed UDDI-M^T which augments the functionality of existing UDDI servers by: *(i)* locally storing arbitrary metadata as triples; *(ii)* supporting an RDF-based query language RDQL[8] that in turn can offload more query processing to the discovery service; *(iii)* abstracting the interface of different discovery services into a single unified view; and *(iv)* supporting local annotation and selection policy management. As UDDI offers an already complex interface (e.g. allowing searches on business categories and service names), UDDI-M^T uses a "tunnelling technique", dispatching regular UDDI requests to a UDDI service, and intercepting UDDI-M^T metadata specific requests.

In summary, the contributions of this paper are the following.

1. First, we provide additional motivation for metadata-based service discovery;
2. Second, we describe the architecture of UDDI-M^T;
3. Third, we evaluate the performance of the tunnelling technique and we contrast a simple key-value metadata search with the graph-based RDQL queries.

The paper is organised as follows. In Section 2, we motivate the need for recording personalised metadata for service directories, discuss alternatives and justify the motivations for moving to RDQL queries. We describe our architecture in Section 3, and its implementation in Section 4, and conclude in Section 5.

2 Background and Motivation

Service discovery has played a crucial role in the evolution and deployment of distributed systems. Early distributed systems comprised collections of components (e.g. client/server or object-oriented) that were implicitly linked through function names, or linked through TCP/IP-based host and port addresses. Federated domain name servers (DNS) simplified and abstracted the use of these numeric addresses by providing a registry-based mechanism for locating the hosts. JINI [1] used a similar approach as part of its Java-based distributed infrastructure. Classes exposed and published their interfaces as *proxy objects* with a JINI discovery service. By searching for a given class-name, matching proxy objects could then be retrieved and invoked, which would in turn invoke the remote service. Whilst providing a mechanism whereby services could easily be added, removed or replaced within a system, this approach was based on an assumption that there was a shared agreement about what a given service type was called (i.e. its class name) and that there was an agreed and well defined interface. Other distributed technologies support similar principles, including DCOM, Corba, XPCOM, etc.

Web Services extend the idea of JINI services by relaxing several assumptions. Built upon Web technologies, Web Services are declared in XML and utilised Web-based protocols (such as HTTP) to publish and retrieve XML documents. The Simple Object Access Protocol (SOAP)[17] provides a transport mechanism to shuttle XML content between services or applications. The Web Services Description Language (WSDL)[16] explicitly defines the interface of a service. By adhering to these definitions, services can be produced that automatically publish WSDL descriptions that in turn are used to define the content of SOAP messages, and thus simplifying the development of interoperable components.

However, in order to utilise these published services, developers must first locate them. Unlike JINI, Web Services do not belong within well defined class hierarchies, and thus it is not feasible to locate services through class labels. Instead, the UDDI service directory [14] provides a mechanism whereby service providers can register information about themselves, such as physical address, contact phone number, etc; the types of business they are involved in; and references to the service descriptions. UDDI registries provide this information in response to white-pages queries (i.e. given the name of a service provider, what are its details) and yellow-pages queries (i.e. what service providers provide services that belong to a given pre-defined service type). Based on a set of queries, developers are able to browse through a list of service descriptions to locate a desired service. However, little support is provided for searching for a service based on a capability (i.e. signature) or user defined data.

Technical Models (*tModels*) support the specification of additional attributes that can be associated with objects stored in the UDDI repository. In their most common mode of use, tModels provide a *fingerprint* that are defined globally across a UDDI registry and refer to a technical specification adhered to by a particular service binding template. The other use for tModels is as expressions of particular identifier or classification namespaces (e.g. US Tax Code ID or the NAICS taxonomy) that can be attached to business entities, services, or binding templates. Additional information (i.e. values) can be associated with such tModel references within a UDDI entry, thus allowing metadata to be associated with these entries. Consequently, tModels may be used as follows:

1. *Concept reference to a technical specification.* For example, several businesses may adhere to a specific RosettaNet Partner Interface Processes specification [12]. Once a tModel is defined for this specification, all future UDDI business entries can indicate whether or not their associated service also adheres to this specification by including this tModel.
2. *Sets of metadata keys.* UDDI entries can be extended globally by defining tModels that correspond to new properties. Thus, a UDDI business entry can include a reference to this tModel, and associate some value to it.

This latter ability provides a powerful, but limited mechanism for augmenting service registrations with metadata, and was utilised to encode properties from the DAML-Services (DAML-S) ontology [4] within UDDI records [10].

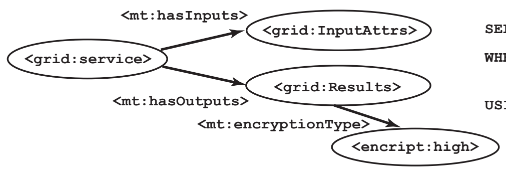
However, before tModels may be used, they need to be registered with a UDDI server and hence be unique. Whilst this can be used to map well defined specifications to tModels, it is inappropriate for specifying large numbers of locally used metadata attributes (such as a set of attributes that may be shared by a single organisation or domain). The UDDI V3 specification attempts to patch this oversight by defining a specific tModel, *general_keywords*, to allow simple unregistered key/value pairs to be attached to a UDDI entity. Ignoring the fact that there are no V3 implementations yet, this patch still is oriented towards metadata supplied by the service *providers*, not users, and allows only simple textual metadata as opposed to more complex structures. An alternative approach to storing explicit, personalised, and possibly dynamic metadata that is associated with a service description is required that addresses these deficiencies.

Many Grid projects require large numbers of service-based and domain-based attributes, some of them complex, to describe additional properties about engineering and scientific metadata. Examples include:

– Perceived reputation of a service: such information is critical to build webs of trusted services in an open environment;
– Perceived reliability of a service: such information will have more value if it is provided by a third party, and not by the service provider itself;
– Perceived quality of service of a given instance (e.g. poor network connectivity for a user may imply that external services appear to be slow);
– Price for accessing a service (the user's institution may have negotiated a local price to access a resource, e.g. ACM or IEEE digital libraries);
– Ontological descriptions of a service, which may differ if multiple ontologies or multiple interpretations of a service exist. While we may imagine that a whole scientific community shares a common ontology, the very nature of undertaking research necessarily entails that ontology revisions will be created by those who undertake this research, and who will therefore want to use them in order to characterise services within their refined ontologies.

2.1 RDF and RDQL

RDF, the W3C Resource Description Framework [11] was originally developed as a framework for describing and interchanging metadata for published data on the Web. Although built using XML, it overcomes many of the limitations of simply using XML for metadata by removing order dependency and the construction of nested tree-like data-structures. Instead, RDF represents metadata as *triples*, which can be used to construct graph-based data-structures (Figure 1(a)). The triples represent two resources (the *subject* and *object*) and a property that relates these resources (known as the *predicate*). For example, the triple *"(results, encryptionType, high)"* would contain the subject *results*, predicate *encryptionType*, and the object *high*. Both subject and object can, in turn, be related to other concepts using additional properties (or predicates), forming a directed graph. RDF extends this natural structure by allowing resources and properties to be represented by URIs, which refer to elements within RDF schema vocabularies, or *ontologies*.



```
SELECT ?x
WHERE (?x,<mt:hasOutputs>,?y),
      (?y,<mt:encryptionType>,<encript:high>),
USING grid FOR <http://www.grid.org/service#>,
      mt FOR <http://www.uddimt.org/metadata#>,
      encript FOR <http://www.mygrid.org/crypt#>
```

Fig. 1. (a) RDF Graph (b) RDQL Query

RDQL is a graph-based query language [8], that exploits the graph structure of RDF. It is an implementation of SquishQL for the Jena RDF toolkit [7]. Based on SQL, it supports queries that themselves are graphs constructed as sets of triples. Uninstantiated variables are preceded with the syntax "?". The `SELECT` clause identifies the variables to be returned, and the `WHERE` clause specifies a conjunction of triples that represent the graph. `USING` provides an abbreviation for URIs by defining a prefix. Figure 1(b) displays a graph query, which looks for all services "`?x`" with encrypted outputs.

2.2 Summary

While UDDI is the acknowledged standard directory service mechanism for Web Services, it is limited in the kind of metadata that can be stored about services, the ways in which it can be queried, and who can annotate service descriptions with metadata. Our previous work, UDDI-M, was an early attempt to associate metadata with services and maintain soft state information [5], based on *leases* à la JINI. Both ideas were reused by the Cardiff team in their service directory with QoS information [13]. With UDDI-M^T, we take a further step by regarding and implementing metadata as triples, which gives us access to Semantic Web technologies such as RDF, and powerful query languages such as RDQL over a uniform representation of information. UDDI-M^T works in conjunction with a

UDDI service to provide precisely these extra capabilities, and eventually support for personalised directory service federation and semantic service discovery.

3 Architecture

In this section, we describe the principles underlying the architecture of UDDI-M^T. Details can found at `http://www.mygrid.ecs.soton.ac.uk/software/service-directory/`. We considered the following requirements during the design:

1. UDDI-M^T should be compliant with the original UDDI [14] specification and support future development in this direction.
2. Existing client and service provider applications should be ported easily to UDDI-M^T.

The key components of the architecture are depicted in Figure 2, where we see that UDDI-M^T is the point of contact for clients, either dispatching requests to UDDI or processing them locally.

As far as the implementation is concerned, UDDI-M^T was designed to be as generic as possible. First, all incoming requests are dispatched to the appropriate handler according to their type. Second, the UDDI-M^T backend is specified by an interface, which can be implemented in different ways: currently, we are supporting a relational database and the Jena triple store [7].

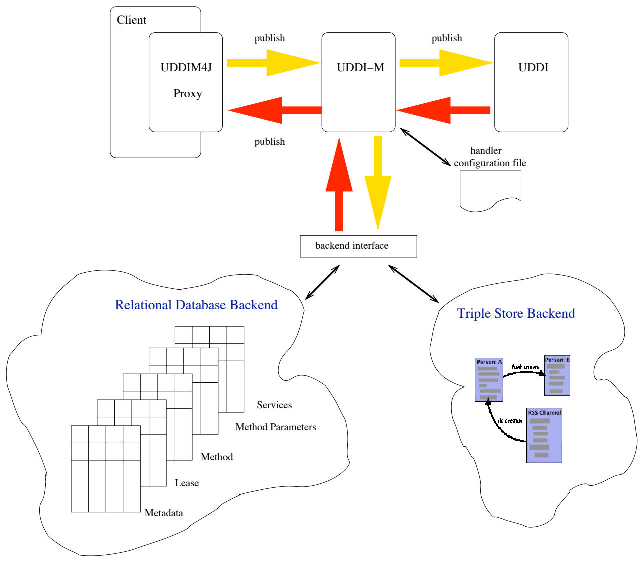


Fig. 2. Architecture of UDDI-M^T

This design assumes that all SOAP messages for the service directory issued by the client are routed to UDDI-M^T, which selectively filters incoming SOAP messages. It relies on the combination of the SOAP envelope and namespace contained in the message to dispatch the message to the appropriate handler, as specified by a configuration file loaded at initialisation time. Messages with the namespaces of UDDI versions 1 and 2 are directly tunnelled to UDDI, whereas messages with a UDDI-M^T namespace are handled locally.

All metadata-related information is stored in the UDDI-M^T backend. Its interface is implemented in two different ways. First, we use a relational database with five tables for metadata, leases, methods, their parameters and services. Second, the same information is implemented in a Jena [7], for which two implementations are possible, a relational database or the Berkeley Triple stores [2].

UDDI-M^T offers several extensions to UDDI. First, UDDI-M^T is able to associate metadata with services. Second, it supports a lease mechanism that requires services to renew their lease in order to maintain their registration in UDDI-M^T; such functionality is present in Jini [1] and is also ubiquitous in the Open Grid Services Architecture [5]. Third, UDDI-M^T is able to extract the information contained in WSDL files describing the interface. Fourth, UDDI-M^T extends the query mechanism of UDDI to allow searches of all the extra information it accumulates about services. Fifth, in the specific case of the Jena backend, UDDI-M^T allows users to express queries in the RDQL-query language [7], offering homogeneous ways of traversing the metadata graph associated with services.

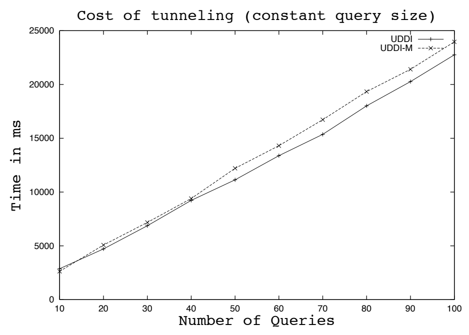
While UDDI is defined as a Web Service, a programmatic interface, UDDI4J [15], is also available for Java: it provides a client-side proxy with an API implementing the UDDI functionality, which allows programmers to abstract away from the messaging layer. We have extended this proxy, by subclassing it, with additional UDDI-M^T functions for managing leases and metadata. Thus, we are preserving clients' binary compatibility. Indeed, a UDDI proxy can be substituted for a UDDI-M^T proxy, transparently to existing clients, since the latter is a subclass of the former. Clients do not have to use the functionality provided by UDDI-M^T, they can use the existing namespace specification and the calls will be directly tunnelled to the underlying UDDI service.

4 Performance Analysis

The purpose of this section is to evaluate the performance of our design decisions. We will focus our attention on two specific aspects. First, adopting the tunnelling technique reduces the implementation effort and allows us to maintain compatibility with evolving standards, but it comes at the price of SOAP-message forwarding. In the first part of this section, we analyse the cost of tunnelling. Second, the use of metadata in a service directory allows us to reduce the computational load on clients, while performing more selective and computationally intensive queries at the server side. In the second part of this section, we analyse the cost of metadata querying, and see how the use of the RDQL language, offering extended expressiveness to the user impacts on the querying cost.

Tunnelling Cost Our hypothesis is that *the overhead introduced by the tunnelling technique is acceptable*. In order to evaluate such a hypothesis, we have set up the following experimental framework.

A UDDI-M^T service and its associated UDDI service are hosted in a Tomcat server. A client uses a UDDI4J proxy successively configured to use UDDI and UDDI-M^T. In order to avoid the cost of networking, both the client and services are run on a same machine, and communications take place through the "local-host" network device. We issue a UDDI-query that searches for a service with a specific name, for which a single instance has been registered. Figure 3 shows the overhead introduced by UDDI-M^T, which tunnels the request to UDDI. The tests were run on a Pentium 4, 1.5GHz, with 512Mb, using Tomcat 4.0 and the Registry Server 1.0_02, in the Java Web Services Developer Pack (1.0_01). The data plotted were averaged over 10 runs. The tunnelling overhead is 7.2%.



UDDI	
Mean	189.9ms
Std Dev	24.9ms
UDDI-M	
Mean	203.7ms
Std Dev	24.8ms
Abs Overhead	13.8ms
Rel Overhead	7.2%

Fig. 3. Overhead of Tunnelling

We also evaluated the cost of tunnelling as the size of query results increases, but did not obtain any significant result, as the marginal tunnelling cost was noise compared to the querying cost.

Metadata Querying Cost Our second hypothesis is that *using a triple store as an internal representation mechanism for* UDDI-M^T *is practical, and the use of the RDQL-query language can reduce communication costs, and offload the client, by performing some server-side computation.* For these experiments the associated UDDI is not involved, resulting in a commensurate reduction in query times from the previous experiments.

In Figure 4, we measured the costs of attaching a property value pair to a service already registered, which we called a *property write* operation, and of finding a service with a given property value pair, which we called a *property read* operation. We used the two different backends, a mySQL relational database and Jena with the Berkeley triple store for these experiments. For the Jena, backend we use the API to find the service with a given metadata, and we did not rely on the RDQL-query language. We plotted the results in Figure 4(a) using a logarithmic scale to differentiate the curves better. *Our purpose here is is not to compare persistent storage technologies, but to understand the cost of metadata management. We can see that read operations for both backends and the write operation in the Jena store are very similar.* We explain the higher cost for the write operation with the SQL database by the cost of storing information on disk, which is probably not measured with the triple store.

In Figure 4(b), we used the RDQL-query language to search for a service satisfying 100 properties; 20 such services were found in the system. For convenience, we again plotted the Jena read line from Figure 4(a). We can see that the RDQL-query engine processing a complex query that checks 100 properties marginally outperforms our direct use of the triple store API, which itself behaved well compared to a relational backend.

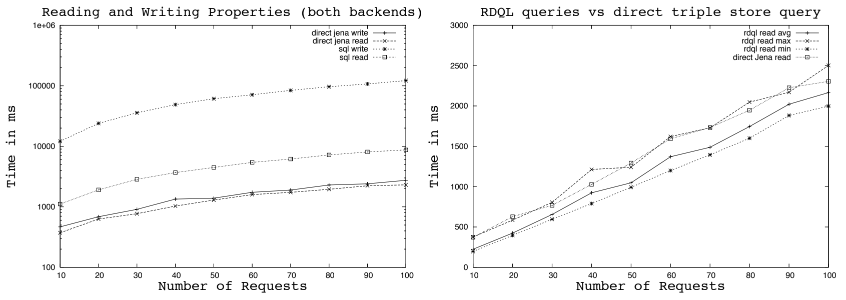


Fig. 4. (a) Property Read and Write (b) RDQL Queries

5 Conclusion and Future Work

In this paper, we have discussed the necessity of attaching metadata to services registered in service directories. Such metadata, describing functional and non-functional characteristics of services, can be provided by both publishers and consumers of a service. We have presented the architecture and the implementation of UDDI-M^T, an extension of UDDI, supporting such a facility. In addition, in order to provide a more uniform view of the metadata attached to services, we have introduced in the service directory the idea of a triple store and an associated RDF-based query language. They give opportunities to clients to offload the service searching process to the service directory servers, also reducing the communication costs. Our experimental evaluation has shown that our architectural design is practical.

Our future work will cover several further aspects so as to provide powerful, personalised service discovery. First, the information model associated with UDDI could completely be encoded in a triple store, hence providing a uniform way of querying over all information related to service descriptions. Second, multiple UDDI service directories will co-exist, and we need to be able to federate their content into a personally curated service discovery view, acting as the users' single point of contact for their service discovery. Finally, curating such a federation of service directories has to be semi-automatic in order to remain tractable: to this end, we need such federation to be defined by a policy that specifies how information should be managed.

6 Acknowledgement

This research is funded in part by EPSRC myGrid project (reference GR/R67743/01).

References

1. Arnod, O'Sullivand, Scheifler, Waldo, and Wollrath. *The Jini Specification*. Sun Microsystems, 1999.
2. Berkeley DB. http://www.sleepycat.com/, 2002.
3. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
4. DAML-S Coalition:, A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In *First International Semantic Web Conference (ISWC) Proceedings*, pages 348–363, 2002.
5. Ian Foster, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke. The Physiology of the Grid — An Open Grid Services Architecture for Distributed Systems Integration. Technical report, Argonne National Laboratory, 2002.
6. Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 2001.
7. Jena semantic web toolkit. `http://www.hpl.hp.com/semweb/jena.htm`.
8. Libby Miller, Andy Seaborne, and Alberto Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. In *First International Semantic Web Conference (ISWC) Proceedings*, pages 423–435, 2002.
9. Luc Moreau. Agents for the Grid: A Comparison for Web Services (Part 1: the transport layer). In Henri E. Bal, Klaus-Peter Lohr, and Alexander Reinefeld, editors, *Second IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002)*, pages 220–228, Berlin, Germany, May 2002. IEEE Computer Society.
10. Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara. Importing the Semantic Web in UDDI. In *Web Services, E-Business and Semantic Web Workshop*, 2002.
11. Resource Description Framework (RDF). `http://www.w3.org/RDF/`, 2001.
12. RosettaNet. RosettaNet Web Site. http://www.rosettanet.org, 2000.
13. Ali ShaikhAli, Omer F. Rana, Rashid Al-Ali, and David W. Walker. UDDIe: An Extended Registry for Web Services. In *Workshop on Service Oriented Computing: Models, Architectures and Applications at SAINT Conference*. IEEE Computer Society Press, 2003.
14. Universal Description, Discovery and Integration of Business of the Web. `www.uddi.org`, 2001.
15. UDDI4J Home Page. `www.uddi4j.org`, 2001.
16. Web Services Description Language (WSDL). `http://www.w3.org/TR/wsdl`, 2001.
17. XML Protocol Activity. `http://www.w3.org/2000/xp`, 2000.