Evaluation of UDDI as a Provider of Resource Discovery Services for OGSA-based Grids

Edward Benson, Glenn Wasson, Marty Humphrey
{ mah2h@virginia.edu | wasson@virginia.edu | humphrey@cs.virginia.edu }
Computer Science Department, University of Virginia, Charlottesville, VA 22904

Abstract

Grid computing involves networks of heterogeneous resources working in collaboration to solve problems that cannot be addressed by the resources of any one organization. A pervasive problem for Grid users is how best to discover the resources they need given dynamic Grid environments. UDDI, the Universal Description, Discovery and Integration framework, is an OASIS standard for publishing and querying discovery information for Web services, which to date, has received surprisingly little analysis as a discovery mechanism for Web service-based Grids, e.g. those based on the Open Grid Services Architecture (OGSA). This work identifies issues that must be addressed in order to make UDDI meet the requirements of OGSA discovery. We examine the performance implications of these issues using a freely available implementation of UDDI version 2. Based on our experimental results, we conclude that UDDI can be used for OGSA discovery, but the cost may be prohibitive for large Grids.

1. Introduction

While Grid computing technology offers the ability to connect large, diverse groups of widely distributed resources to address complex problems, these same issues of scale and geographic distribution require a sophisticated mechanism by which Grid users can find available resources that meet their requirements. Often, users will not know the exact names of the resources they wish to use, but will instead know only the abstract properties that those resources must possess. The discovery problem then, is the problem of how to map a user's requirements to a set of resources that meet those requirements.

This material is based upon work supported by the National Science Foundation under Grant No. 0203960, Grant No. 0426972, and Grant No. 0438263. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

While several solutions for this discovery problem have been used, e.g. LDAP [12] or MDS [4], as Grid computing moves toward a Web services-based substrate, such as Grids based on the Open Grid Services Architecture (OGSA [5]), it make sense to evaluate the Web service world's standard mechanism for discovery, UDDI [3][10]. UDDI, the Universal Description, Discovery and Integration framework provides a means of publishing and organizing information about resources and subsequently querying that information to "discover" resources based on client-specified information.

Simply, the widely-accepted approach of next-generation Grids is to utilize tooling and run-time systems provided by commercial vendors (e.g., Microsoft, Sun, IBM) and open-source projects (e.g., Apache) for service/client development and inter-service communication; the potential value of UDDI is, by utilizing UDDI, next-generation Grids could similarly leverage this existing/emerging broader support for discovery. For example, just as Visual Studio.NET (VS.NET) has an "Add Web Reference" that easily generates proxies to existing Web services via WSDL retrieval and processing, VS.NET also has integrated processing of UDDI registries. However, to date, Web service-based Grids, such as those based on the emerging Open Grid Services Architecture (OGSA), have not utilized UDDI as a discovery mechanism and it remains an open question whether UDDI is appropriate for this task, as there have been surprisingly few published studies on the utility of UDDI for Grids.

The questions identified and addressed by this paper are:

- What are the issues involved in using UDDI as a resource discovery mechanism for OGSA-based Grids?
- If UDDI does not "natively" meet the requirements of OGSA, what modifications are necessary/sufficient to overcome these limitations?
- What are the performance implications of these modifications?

We examine these issues by using a freely available implementation ("jUDDI" [9]) of the most-widely-utilized version of the UDDI standard, which is version 2 (v2). We focus on Version 2 of the UDDI OASIS standard because it is the dominant implementation available today and for the foreseeable future, both commercially and in open-source projects. Based on our experimental results, we conclude that UDDI can be used for OGSA discovery, but the cost may be prohibitive for large Grids.

The remainder of this paper is organized as follows. Section 2 describes the discovery requirements of OGSA-based Grids and section 3 describes the UDDI protocol and information infrastructure. Section 4 discusses the issues in using UDDI for Grids and proposes solutions for these issues. Section 5 evaluates the performance of those solutions and therefore quantifies the "cost" of UDDI. Section 6 discusses related work in resource discovery and Section 7 provides our analysis and recommendations about using UDDI based on our experience. We also discuss how changes made to the latest version of the UDDI specification may make it more suitable for discovery in Grids, although not without limitations.

2. Resource Discovery in OGSA-based Grids

The Open Grid Services Architecture (OGSA) represents a new vision of computing that merges the worlds of Grid computing and Web services. The OGSA working group [11] of the Global Grid Forum [6] has defined a standard set of roles that a set of services must fill in order to perform canonical Grid tasks. This paper focuses on two of those roles related to resource discovery, the Candidate Set Generator (CSG) and the Information Service (IS).

The Information Service component of OGSA maintains a catalogue of dynamically varying information about resources in the Grid. It can be queried by various components of the architecture to discover resources appropriate to a given task. One of the primary users of the Information Service is the Candidate Set Generator. The CSG uses data stored in IS to generate candidate lists of resources with the functional properties required for a given operation. For example, the CSG may be used to discover machines with the correct architecture to execute a binary or storage resources that support GridFTP transfers. However, the IS also maintains information useful for selection of resources based on non-functional properties, such as load or available memory. It can be used (by a scheduler) in conjunction with the CSG's output to select the "best" candidate.

3. UDDI: Design and Use

UDDI is an OASIS standard protocol that defines a "standard method of publishing and discovering network-based software components in a Service Oriented Architecture (SOA)" [10].

UDDI provides its functionality through four principle entities: the *businessEntity*, *businessService*, *tModel*, and the *bindingTemplate*. The businessEntity is the largest container within UDDI. It contains information about any organizational unit which publishes services for consumption. The designers of UDDI envisioned businessEntities as being UDDI representations of actual businesses that choose to offer services over the Internet [capitalize?]. In the Grid context, businessEntities can be used to hierarchically separate and form relationships between different organizational groups within a Grid or multiple Grids.

Each service that a businessEntity offers is described by a businessService object. These objects provide information about the service name, categorization, and any other useful details added in a variety of attribute lists. The information is purely descriptive and does not include any instructions for accessing or using the service.

The bindingTemplate object represents the link between abstract businessService descriptions and actual endpoints at which these services may be accessed. Like all objects in UDDI, bindingTemplates are given universally unique identifiers, known as UUIDs, which are used as a key of reference. Each businessService object stores the UUID keys of bindingTemplates within the same businessEntity that provide instances of that service.

BindingTemplates may provide specialized information about a particular businessService endpoint through use of tModels. The tModel is the standard way to describe specific details about a particular businessService or bindingTemplate in UDDI. tModels contain lists of key-value pairs used for description and may be associated with multiple objects in the UDDI database. They may also contain placeholders for URIs which point to descriptive information external to the UDDI registry.

4. Issues in Using UDDI for OGSA Discovery

UDDI was designed as a business directory system and has some limitations that complicate resource discovery in Grid computing. Namely, these are 1) a lack of explicit data typing for information in the UDDI directory, 2) difficulties in handling dynamic information (such as CPU load) that requires frequent updating and 3) the limits of the UDDI query model. This section addresses each of these limitations in turn and proposes work-around solutions.

4.1 Lack of Explicit Data Typing

The ability to associate data types with resource metadata is fundamental for resource discovery in Grid environments. Data typing allows not only more strict matching of resource information with resource requirements, but allows a more diverse variety of comparison operations, e.g. greater or less than, than simple equivalence for untyped values.

While UDDI contains many complex data types, such as the businessEntity and the tModel, it maintains little notion of type for the data contained within these objects. The tModel structure, for instance, is the fundamental container of metadata that can be attached to an object within UDDI. tModels contain two collections into which metadata can be placed – the identifierBag and the categoryBag – each containing zero or more keyedReference objects. Each keyedReference object is essentially a key-value pair in which both the key and the value *must be* strings. An example keyedReference is shown in Figure 1.

```
<keyedReference

tModelKey="uddi:cs.virginia.edu:sampleKRef"
     keyName="SOME_ATTRIBUTE"
              keyValue="364.3"
/>
```

Figure 1. Example UDDI keyedReference Object

String values are appropriate in the business world for which UDDI was designed. In that context, categorization of products and services (through discrete string values) is the common use case. However, Grid environments and the scientific community require classifiers based on continuous variables, and so the string-only keyedReference pairs hinder UDDI's ability to provide a search model capable of fulfilling the basic queries of Grid users, such as performance-based resource selection.

To include the types of continuous variables, such as system load and memory size, as meaningfully searchable items within a UDDI registry, these variables must be flattened into enumerated sets of predefined buckets into which the data will be placed. System load, for example, might be described with a classification scheme in which machines are associated with an element of the set { [0-0.5], [0.5-1.0], … ,[9.5-10.0], [10+] }.

This unavoidable approach has several drawbacks. First is the reduced specificity with which users will be able to search for services on the Grid. In many cases Grid administrators will want to define their enumerated sets with unevenly spaced buckets to give a higher resolution to the possible ranges that are more important for performance. Such non-standard enumerations also complicate searching because all clients must know the range breakdowns in order to formulate their queries.

Finally, range-based searches are complicated by this method. A user query specifying a system load less than 2, for example, would have to be translated into a query with a series of "OR" statements encompassing all buckets between [0-0.5] and [1.5-2.0]. Support for range queries adds the requirement that the process which translates from continuous variables to "OR" clauses understand the ordering of buckets in the range enumeration.

UDDI provides support for wildcard-based searching similar to that offered in SQL queries. Users have the option of single-character wildcards with the '_' character and multiple character wildcards with the '%' character. Using these two operators and the method of using ordered sets to replace continuous variables, we can potentially generate simpler queries by mapping the names of the range sets to appropriate strings. For example, suppose machine memory size is represented by one of the following ranges, { [0-255], [256-512], [513-1024], [1025-2048], [2048+] }. We can perform queries of the form "find me a machine with X amount of memory *or more*" by storing not the textual representation of the range in which the machine's memory falls, e.g. [513-1024], but rather a string representation of the index of the range within the range set. So, if a machine's memory fell in the [513-1024] range, we would store a string like "AAABB", where the number of A's indicates the index of the desired range (in this case, it is the third in the range set). B's are then used to fill out the string until it has as many characters as there are possible ranges (in this case 5). Such a formulation allows wildcard queries like "AAA%" to find any machine that has a memory size of 513 MB or more.

This enumeration-based scheme with wildcard extensions does not make up for UDDI's inability to handle typed data, but it does allow UDDI to provide an approximation of the metric-based searching that the Grid community expects from a resource discovery service.

4.2 Dynamic Service Data

UDDI is targeted at not just Web services discovery but also a broad array of uses including everything from industry directories to product information databases. One attribute in common with all of the intended uses of UDDI is relatively static data. Perhaps because of this assumption, UDDI has no built-in notion of dynamic service information or any other mechanisms in which the context of stored data changes over time.

The implications of this inability to the Grid community are quite large. This makes it difficult for

UDDI to make available such important time varying information as CPU load. However, potentially more serious is the inability to represent transient resources. While some Grids could consist of dedicated machines running in a tightly-managed environment, many Grids leverage the ability to draw resources from home, work, and other "dual-use" computers which are not full-time available to the Grid. In this type of environment it must be assumed that the availability of resources is not only unpredictable, but also that this unpredictability will put the burden of maintaining up-to-date availability information in the resource discovery service. In other words, resource providers can not be expected to remove appropriate serviceBinding records from a UDDI registry when the associated resources become unavailable.

This "dangling bindings" problem can hinder the performance of a Grid environment as it stands to clutter an accurate registry of resource information with false records. As the number of resources in a Grid grows, the likelihood that service discovery attempts will return dangling bindings increases.

As a work around to this problem, resource providers may send a periodic "heartbeat" update message to the UDDI server at defined intervals. This update message will refresh a lastUpdateTime field inside a tModel that is associated with all bindingTemplates in UDDI owned by that resource provider. Because of UDDI's lack of support for data typing (and hence queries such as "updates more recent than X"), the last update time must be represented as an interval rather than a string representation of a literal timestamp. While fortunately this approach does not require that all clocks in the Grid be synchronized (a daunting task), it does require that the background heartbeat process be able to apply a transformation that converts the sender's local time into one that matches the global time intervals used by the UDDI registry (and hence recognized across the Grid).

Assuming this approach, Grid users can avoid receiving candidate sets of resources that contain stale or unavailable resources by including a filter in the query requiring only machines that have a lastUpdateTime equal to the current (and possibly previous) time interval. Including the previous time interval trades the possibility of erroneously receiving recently disconnected machines for the assurance that one is searching the whole pool of providers, not just those that have chosen to update during the portion of the current time interval prior to the search.

Following this practice, the probability of a stale result within a candidate set is measurable. At any query time *s* local to the beginning of an update interval of length *t*, the probability of picking a machine that is no longer available is $\frac{t-s}{t}p$, where p is the probability that a given machine will abruptly disconnect from the Grid, with all machines on the Grid choosing to send updates at an even distribution across the entire update interval. This equation makes the simplifying assumption that machines disconnect only during update intervals in which they did not and will not send an update.

The specification for UDDI version 3 does not incorporate any way to provide the updating framework needed to replace the practice suggested here, but this paper demonstrates a method of achieving the same capability with a measurable amount of error. After deploying and observing a Grid using UDDI, administrators can take advantage of the measurability of this type of error to optimize the update interval length.

4.3 Search Model

For UDDI to be successful as a resource discovery service in Grid environments, it must be able to respond to requests such as "find a resource that can perform task X on a machine with properties W, Y, and Z with guarantees A and B." To date, the only available implementations of UDDI conform to version 2 of the specification and so this section discusses that version (subtle, but important differences exist in the discovery models of version 2 and version 3 and these are discussed in section 6).

Ideally, a host offering resources to a Grid should be able to publish a series of bindingTemplates to UDDI, each of which represent the endpoint of a particular Web service being offered at that host. Each bindingTemplate would then contain the UUID key of a tModel maintained by that host which contains both dynamic and static performance information of that host as well as the time of the last update of this information. At regular intervals, the host would update the information stored in this tModel, thus changing the metadata associated with all of the bindingTemplates that reference it.

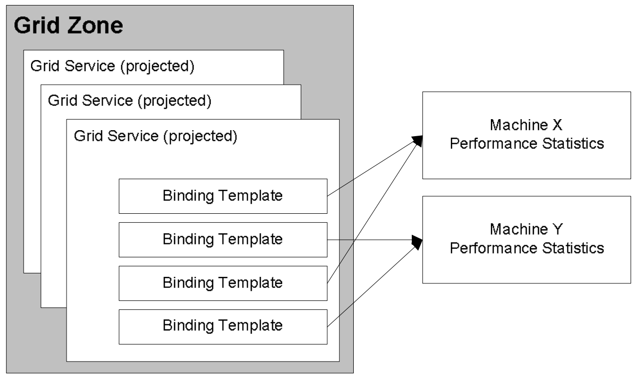

Figure 2. Desired UDDI Organization

Grid users wishing to locate bindingTemplates for a particular service would specify which system performance characteristics are required for the job at hand, and UDDI would do a "deep search" of both bindingTemplates and their associated system performance tModels. Figure 2 illustrates this organization.

UDDI version 2 does not contain the functionality to create such a desired discovery scenario. The find_binding API call in UDDI version 2 only allows clients to specify the UUID keys of desired tModels, not place query criteria upon the key-value pairs within these tModels. Two separate approaches for querying against tModel can be taken, but both have side effects that degrade performance.

The first method is to map tModel identifying keys to labels that represent various enumerated intervals for the selected performance metric. In other words, the key is the value. Free disk space, for example, would be described by a set of tModels that represent each of the possible range intervals that have been established for this metric. The keys for these tModels would be, for example, "100-200MB free" or "1+GB free". These keys, or possibly some well-known formula for constructing them, must be known by all machines on the Grid. At each update interval, resource providers associate each of their bindingTemplates with the proper tModel for each metric that the resource provider wishes to report. Under this scheme, there is no central place where an administrator could go to view a machine's current status – the administrator could only view this information by examining the associations created between the global metric tModels and a bindingTemplate owned by the machine. Figure 3 illustrates this design.

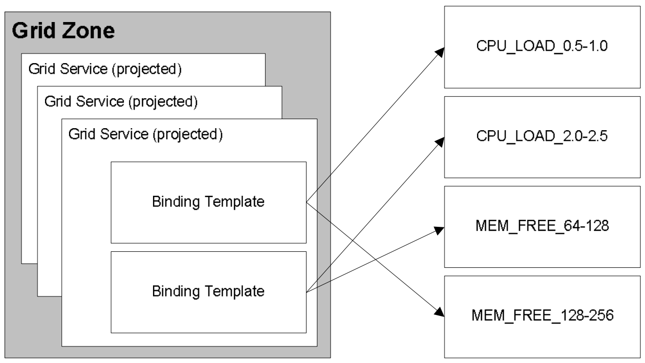

Figure 3. Association of Performance Data with bindingTemplates in UDDI v2

This technique accomplishes the task of allowing machine metrics to be integrated into bindingTemplate queries, but at a performance cost due to the difficulty of updating. Instead of each machine updating a single performance tModel during update intervals, each machine must update each of its bindingTemplate records. For Grids with many general-purpose resource providers each offering the use of many Web services, the extra updates required (equal to the average number of bindingTemplates per host) per update interval can noticeably impact the performance of the UDDI server.

A second method of associating machine attributes with bindingTemplates involves a two-step searching process. Each resource provider maintains all performance data within one tModel and associates its bindingTemplates with this tModel. Each update interval, the provider need only update this one record, but Grid users must perform two queries to find an acceptable set of binding endpoints. The first query searches the contents of all tModels in the UDDI registry and returns a list of those matching the machine requirements for a particular task. The second query searches all bindingTemplates in the registry using the tModel keys returned as the results of the first search as filtering criteria.

As the size of the Grid and the number of stored tModels grows, the set returned by the first search may grow large enough to cause noticeable delays from the perspective of the Grid user. One potential solution to reduce the size of the tModel list returned by the first query is to put back-references within each machine's performance tModel which contain the keys of the businessServices that the machine offers. These back-references can be used as filters in the first search so that only the tModels of machines which host the desired service are returned.

Consideration is due before using back-references to limit the search results, however, because this practice seems to violate the intended separation of information within UDDI. tModels provide descriptive data about bindingTemplates, but are arguably not supposed to contain any knowledge of the entities that reference them.

5. Performance

Section 4 outlined the three main hurdles in applying UDDI version 2 as a resource discovery service to OGSA Grids and how these hurdles may be overcome or at least reduced. This section outlines the measured performance of a publicly-available UDDI implementation ("jUDDI" [9], which is compliant with UDDI version 2), in a simulated Grid environment that was set up at the University of Virginia. In our experiments below, we are careful to distinguish properties/assertions that we believe can be attributed to the UDDI specification (and thus *all* implementations, from our judgment) vs. those that should be attributed to the specific implementation of the UDDI specification that we studied (jUDDI).

Figure 4 illustrates the experimental setup. The experiments were conducted in a 100 Mbps LAN environment. The bottom of the figure shows a jUDDI server running in an Apache/Linux environment on a 1.4 GHz AMD Opteron machine with 2 GB of memory. The top of the figure shows the remaining machines utilized (up to 14), each running client software implemented in C# on Microsoft Windows XP Professional. The client software was divided into two layers. The lower layer ("UDDI client") wrapped the UDDI API and translated it into a Grid-centric API for resource providers and Grid users. The intention of this layer was to take the initial steps to create a standard resource discovery service API that would function with any off-the-shelf UDDI implementation running behind it. In OGSA terms, this software layer uses UDDI to implement the behaviors which characterize the Candidate Set Generator and the Information Service.

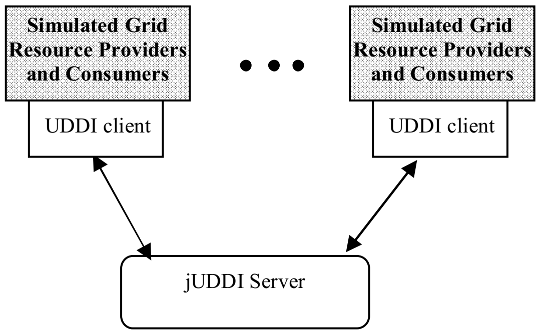
Simulated Grid Resource Providers and Consumers
UDDI client
Simulated Grid Resource Providers and Consumers
UDDI client
jUDDI Server

Figure 4. Architecture used for UDDI Evaluation

On top of this layer, we designed a system that simulates resource providers and Grid users. This simulator was used to construct a number of different Grid environments and measure the performance of UDDI under different levels of sustained activity. We use shading in Figure 4 to denote the simulated pieces, to reinforce that we did not simulate the UDDI server, only the clients/services interacting with the actual UDDI server.

We used three metrics to measure the performance of UDDI (specifically, the jUDDI implementation of UDDI) as a provider of resource discovery services: system load on the UDDI hosting machine, mean update time for service provider information, and mean query time experienced by Grid users. The first metric gives a rough indication of how much activity the UDDI server is experiencing under the simulated Grid conditions, while the second two metrics gauge how this duress will be felt by providers and users of the Grid.

System load was measured as a function of both resource provider update frequency and search frequency. Figure 5 shows the results of the average system load of the UDDI server when subjected to resource provider update frequencies between five and twenty-five updates per second. Each "update" consisted of one simulated resource provider updating information about a random three out of a possible ten resources (meaning that we were simulating a Grid environment in which only an arbitrarily small number of the total possible services were actually instantiated on any particular grid node).

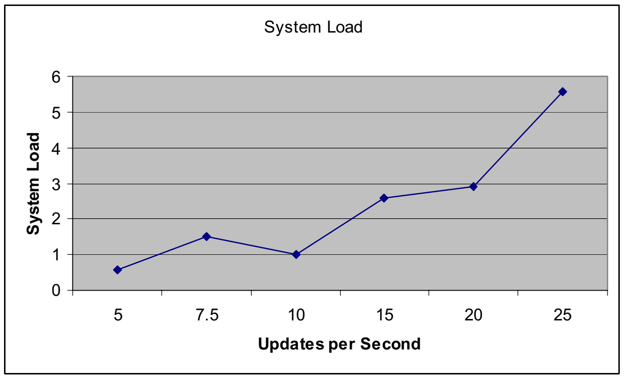

Figure 5. UDDI System Load as a Function of Update Rate

The results of this test show that server load increased roughly linearly with update frequency. A test of 35 updates per second was performed but was not included in Figure 5 because jUDDI performance slowed to such an extent that only a portion of the attempted updates were able to successfully complete within their target update intervals. During this unsuccessful test, the load average rose to an average value of 8.9. Note that our UDDI server is a dedicated, "average-spec" desktop machine circa 2003.

The time required to perform an update was also measured to gauge how increased load on the UDDI system would be experienced by providers submitting new resource information. An update was defined as one resource provider refreshing the tModels and tModel-bindingTemplate associations that represent its current performance characteristics. The time required to perform system updates was measured using ten machines each simulating up to thirty-five resources. Each resource provider offered a random three out of a possible ten services and updated its system performance metrics once every ten seconds.

The results of this experiment show that update time remains relatively constant as long as the number of updates per second remains below twenty. After twenty, a steep rise occurs. Figure 6 shows a summary of the data recorded during these tests. Each data point represents the ten-minute average of all measured update times at that level of activity.

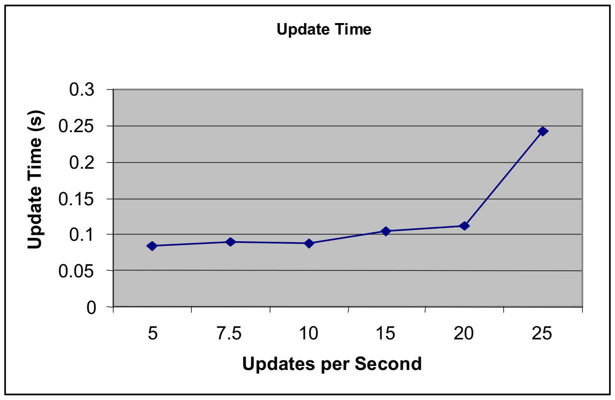

Figure 6. UDDI Update Time as a Function of Update Rate

A final test performed with a target of thirty-five updates per second was not incorporated into this graph because only a fraction of the updates were able to take place during each update interval due to slowed performance.

While it is important to keep in mind that these results apply only to jUDDI, we believe the shape of these curves will apply to any implementation of UDDI v2 (and jUDDI is a popular, open-source implementation worthy of study in its own right). Our tests show that for moderate to large Grids, maintaining accurate resource information may push UDDI beyond its limits, and so a closer look at how the number of updates per second can be reduced is needed, which we consider in the remainder of this section.

This project evaluated the first of the two methods for associating machine performance data with bindingTemplates, depicted in Figure 3. This required that each resource provider update three bindingTemplates each update interval instead of one performance tModel. Had the second of the two proposed methods been chosen, only a single tModel update would have been required, reducing the number of update messages by a factor of three for our particular test environment. However, this would have required the UDDI client to perform two queries to discover appropriate resources -- one to find applicable resources and one to find which of those currently has appropriate performance characteristics.

As a test of the UDDI client experience, we measured the time required to perform a search upon the resource information stored in UDDI as the number of simultaneous searches increased. The average time required to perform bindingTemplate searches on the UDDI registry was measured using twelve machines, two of which simulated resource providers and ten of which simulated Grid users. The two machines acting as resource providers simulated a total of twenty-five providers each offering three randomly chosen services out of a possible ten services. Each simulated resource provider updated its system metrics every ten seconds. The ten machines simulating Grid users performed between five and fifty-five queries on the UDDI data per second. Each query requested a randomly chosen service out of the ten available and limited the query with a desired system load average. Each level of search frequency was sustained for ten minutes. The data points in Figure 6 represent the 10-minute average of the number of seconds required to complete queries as experienced by the ten machines simulating Grid users. Figure 7 shows the results of these tests.

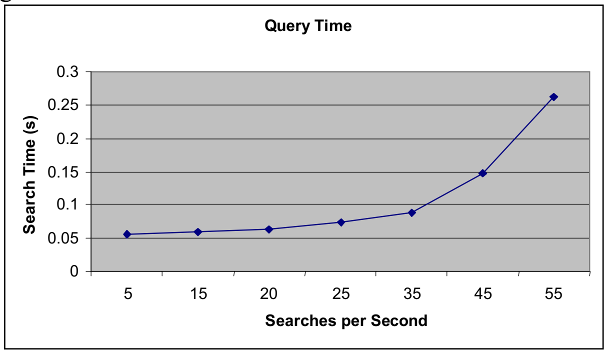

Figure 7. UDDI Query Time as a Function of Simultaneous Searches

Because the user only performs a UDDI query once, at the beginning of each job request say, these average search-times would seem to be acceptable. This information can be of further use in helping Grid administrators anticipate the expected performance of a business or campus Grid as the number of users grows (especially important, perhaps, if multiple universities with campus Grids make the decision to merge their Grids).

6. Related Work

This paper assesses the utility of using UDDI for resource discovery in grids in order to see if existing, widely deployed, commercial and open source UDDI implementations can be utilized for this task. However, there are several other resource discovery methods currently employed in the Grid community. The Lightweight Directory Access Protocol (LDAP) is a simplified version of the X.500 Directory Access Protocol (DAP) which specifies a means of organizing and accessing information directories over the Internet. LDAP is often used in organizations as a means of storing personnel, service, and network topology information. Users of LDAP access information organized in a hierarchical directory tree. Each level of the tree contains attribute-value pairs of information as well as links to lower levels. While LDAP by itself is not a candidate for the role of resource discovery solutions in OGSA Grids (at heart it is a means of storage and organization, not of description and discovery), LDAP's flexibility has made it the choice for a number of resource discovery solutions, including MDS, the Monitoring and Discovery System [4] used by the Globus toolkit [7].

The Globus Toolkit's Monitoring and Discovery System (MDS) uses LDAP to publish information about the current state of resources in a Grid environment. An Index Service provides the capabilities of the OGSA Information Service and includes data refreshing mechanisms that prevent stale data from being returned in query results. MDS's Trigger Service monitors MDS's data catalog for certain preset conditions, providing a means for asynchronous alarms and warnings to be sent to interested parties.

Carnivore [8] is a registry service from the International Virtual Observatory Alliance. Carnivore allows clients to query XML records of resources using the XQuery language. While this gives Carnivore clients powerful query abilities, it does not allow the use of existing UDDI clients.

A new standard, WS-Discovery [2], is a recent addition to the Web services stack that offers a decentralized approach to service discovery. Devices following the WS-Discovery protocol multicast discovery requests to a multicast group and receive responses from resource providers within that group. To prevent unnecessary multicast traffic, "discovery proxies" can join groups to act as central, unicast-based points of reference for discovery queries. In Grid environments, WS-Discovery will most likely be useful as a complimentary technology to MDS and/or UDDI-based discovery. Large scale multicasting can generate a large amount of traffic and can be unreliable across different domains and organizations. However, WS-Discovery services could be used for local discovery and then provide their information to a Grid-wide catalog such as MDS or UDDI.

While this paper has outlined deficiencies with UDDI version 2 and solutions involving a wrapper around a UDDI version 2 service, these same issues are being addressed at the standards level by the UDDIe project at Cardiff University [1]. UDDIe is currently exploring ways in which UDDI version 2 can be extended to provide support for data typing and dynamic service data in a way that does not break compatibility with non-UDDIe client software. The capabilities of UDDI are often conceptualized as the Yellow, White, and Green pages of service discovery. UDDIe adds what it calls the "Blue Pages" to store quality of service and dynamic metadata about businessService records within a UDDI registry.

While the UDDIe project has met with success at extending the UDDI framework, a drawback to this approach is that any solution which requires the modification of UDDI server code and APIs removes a key reason why UDDI is a good candidate for resource discovery in Grid environments. That is, a large part of the attractiveness of UDDI stems from the fact that it is a well known, supported industry standard. A Grid-centric resource discovery solution utilizing UDDI "as-is" automatically benefits from the rich development community and resources already surrounding this technology.

Lastly, the Blue Pages of UDDIe are implemented as a series of attributes that may be appended to businessService records in the UDDI registry; this new capability does not extend to tModels, which remain unchanged from UDDI version 2. Since users of UDDIe only benefit from the addition of typed data in businessService records, the query model still does not permit the users to make use of typed data to differentiate between different providers of a service. Users can, however, use the benefits of typed data when comparing the service-level characteristics of several Grid services which might accomplish equivalent tasks [1].

Finally, the UDDI committee of OASIS has recently released a new version of the UDDI specification (version 3) [3], which was ratified by OASIS on February 3, 2005. UDDI version 3 adds to the bindingTemplate discovery API through the addition of the `find_tModel` argument in the `find_binding` API call. Once implemented, this change will allow each provider's performance information to be stored in a single tModel (depicted in Figure 2) and will allow this information to be used as criteria for bindingTemplate searches through a single API call and no back-references. UDDI version 3 also supports enhanced security features, such as support for digital signatures, on all objects. Richer replication capabilities have also been added, allowing multiple UDDI servers, across several organizations, to each replicate portion of the other's data. Still lacking in UDDI version 3, however, is data typing. The structure of keyedReference object remains the same in version 2, and so numerical, range-based queries are still not supported.

While work is being done to begin implementing this new specification, all current open-source and commercial implementations of UDDI are based on the version 2 standard. Version 3 implementations will more closely match Grid requirements, but may still not be sufficient.

7. Conclusions / Discussion

UDDI is an important component of the Web services stack that was designed to be used in a wide variety of discovery scenarios. As Grid computing moves toward a

Web services-based infrastructure, it makes sense to evaluate UDDI as a part of the OGSA architecture, specifically UDDI's utility as a Candidate Set Generator and an Information Service.

We have found that UDDI suffers from two primary limitations in this context that stem from the issues discussed in Section 4. First, it lacks a rich query model due to its lack of explicit data typing and its inability to easily perform bindingTemplate queries based on the values contained within associated tModels. This makes the inclusion of both functional requirements and performance requirements within the same query cumbersome, and therefore complicates Candidate Set Generation. Second, UDDI is not well equipped to handle environments that contain resource providers with unpredictable availability because of its limited support for the expiration of stale data. This makes UDDI non-ideal as an Information Service which must catalog the current state of dynamic Grid systems, but can be easily circumvented by building this functionality into the provider and user software that accesses UDDI.

While we believe that UDDI version 2 is not an ideal solution for Grid computing discovery services, we have suggested a number of methods that address its chief limitations and bring it closer to what is needed to fulfill the roles of Candidate Set Generator and Information Service. These methods have been implemented at the user-level, and thus can be used with any standards-compliant version of UDDI version 2 or beyond.

The costs of using UDDI and the methods developed in Section 4 as an OGSA resource discovery service have been quantified and explained in Section 5. It is important to note that even if Grid administrators/users were tolerant of the update and query times presented in Figures 5 and 6, the jUDDI implementation begins slowing dramatically for update/query rates greater than those shown in the graphs. In other words, even if UDDI was acceptable to the user community, it does not scale well to handle large numbers of Grid resources. While other UDDI implementations might have better performance/scaling characteristics, they would still suffer from UDDI's model.

Though implementations of UDDI version 3 should become available relatively soon and will contain structural changes that permit increased updating and querying performance, they will still lack the support for typed and time-sensitive data required for level of service desired for OGSA-based Grids. We therefore conclude that UDDI, and in particular UDDI version 2 as implemented by jUDDI, is only appropriate for small Grids in which scalability and precise performance reporting is secondary to the industry support and ease of installation that accompany this technology.

8. REFERENCES

[1] A. Ali, "About UDDIe". Cardiff University School of Computer Science. http://www.wesc.ac.uk/projects/uddie/uddie/about/index.htm. Accessed June 2005.

[2] J. Beatty, G. Kakivaya, D. Kemp, T. Kuehnel, B. Lovering, B. Roe, C. St. John, J. Schlimmer, G. Simonnet, D. Walter, J. Weast, Y. Yarmosh, P. Yendluri. Web services Dynamic Discovery (WS-Discovery). October 2004. available at: http://msdn.microsoft.com/library/en-us/dnglobspec/ html/ws-discovery1004.pdf

[3] T. Bellwood, L. Clément, C. von Riegen, et al. UDDI Version 3.0.1: UDDI Spec Technical Committee Specification, Dated 2003-11-14. Organization for the Advancement of Structured Information Standards, 2003.

[4] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman. Grid Information Services for Distributed Resource Sharing. Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, August 2001.

[5] I. Foster, C. Kesselman, J. Nick, S. Tuecke. Grid Services for Distributed System Integration. *Computer*, 35(6), 2002. *Computer*, 35(6), 2002.

[6] Global Grid Forum. http://www.ggf.org. Accessed June 2005.

[7] Globus Project. http://www.globus.org. Accessed June 2005.

[8] M. Graham. CARNIVORE: Open Source Registry. http://nvo.caltech.edu:8080/carnivore/doc/Carnivore.pdf. Accessed October 2006.

[9] jUDDI: Java Implementation of the UDDI specification. http://ws.apache.org/juddi/. Accessed June 2005.

[10] OASIS. UDDI Executive White Paper. http://uddi.org/pubs/uddi-exec-wp.pdf, accessed June 2005.

[11] Open Grid Services Architecture Working Group (OGSA-WG). https://forge.Gridforum.org/projects/ogsa-wg. accessed June 2005.

[12] W. Yeong, T. Howes, S. Kille, Lightweight Directory Access Protocol. Request for Comments: 1777. ISODE Consortium, March 1995.