Adding OWL-S to UDDI, implementation and throughput

Naveen Srinivasan, Massimo Paolucci , Katia Sycara

Robotics Institute, Carnegie Mellon University, USA
{naveen,paolucci,katia}@cs.cmu.edu

Abstract. The increasing availability of web services demands for a discovery mechanism to find services that satisfy our requirement. UDDI provides a web wide registry of web services, but its lack of an explicit capability representation and its syntax based search provided produces results that are coarse in nature. We propose to base the discovery mechanism on OWL-S. OWL-S allows us to semantically describe web services in terms of capabilities offered and to perform logic inference to match the capabilities requested with the capabilities offered. We propose OWL-S/UDDI matchmaker that combines the better of two technologies. We also implemented and analyzed its performance.

1 Introduction

Web Services have promised to change the Web from a database of static documents to an e-business marketplace. Web Service technology are being adapted by Business-to-Business applications and even in some Business-to-Consumer applications. The widespread adoption of web services is due to its simplicity and the data interoperability provided by its components namely XML [7], SOAP [10] and WSDL [11].

With the proliferation of Web Services, it is becoming increasingly difficult to find a web service that will satisfy our requirements. Universal Description, Discovery and Integration [8] (here after UDDI) is an industry standard developed to solve the web service discovery problem. UDDI is a registry that allows businesses to describe and register their web services. It also allows businesses to discover services that fit their requirement and to integrate them with their business component.

While UDDI has many features that make it an appealing registry for Web services, its discovery mechanism has two crucial limitations. First limitation is its search mechanism. In UDDI a web service can describe its functionality using a classification schemes like NAISC, UNSPSC etc. For example, a Domestic Air Cargo Transport Service can use the UNSPSC code 78.10.15.01.00 to describe it functionality. Although we can discover web services using the classification mechanism, the search would yield coarse results with high precision and recall errors. The second shortcoming of UDDI is the usage of XML to describe its data model. UDDI guarantees syntactic interoperability, but it fails to provide a semantic description of its content. Therefore, two identical XML descriptions may have very different meaning, and vice versa.

Hence, XML data is not machine understandable. XML's lack of explicit semantics proves to be an additional barrier to the UDDI's discovery mechanism.

The semantic web initiative [5] addresses the problem of XML's lack of semantics by creating a set of XML based languages, such as RDF and OWL, which rely on ontologies that explicitly specify the content of the tags. In this paper, we adopt OWL-S [3], an OWL [15] based ontology, that can be used to describe the capabilities of web services. Like UDDI, OWL-S allows a web service to describe using the classification schemes. In addition, OWL-S provides a capability-based description mechanism [6] to describe the web service. Using capability-based description we can express the functionality of the web service in terms of inputs and precondition they require and outputs and effects they produce. Capability-based search will overcome the limitations of UDDI and would yield better search results.

In this paper we propose an OWL-S/UDDI Matchmaker which takes advantage of UDDI's proliferation in the web service technology infrastructure and OWL-S's explicit capability representation. In order to achieve this symbiosis we need to store the OWL-S profile descriptions inside an UDDI registry, hence we provide a mapping between the OWL-S profile and the UDDI data model based on [1]. We also enhance the UDDI registry with an OWL-S matchmaker module which can process the OWL-S description, which is present in the UDDI advertisements. The matchmaking component is completely embedded in the UDDI registry. We believe that such an architecture brings both these two technologies, working toward similar goals, together and realize their co-dependency among them. We also added a *capability port* to the UDDI registry, which can used to search for web services based on their capabilities.

The contributions of this paper are an efficient implementation of the matching algorithm proposed in [2], an architecture that is tightly integrated with UDDI, an extension of the UDDI registry and the API to add capability search functionality, preliminary experiments showing scalability of our implementation and an update of the mapping described in [1] to address the latest developments in OWL-S and UDDI.

The rest of the paper is organized as follows; we first describe UDDI and OWL-S followed by the UDDI search mechanism. In Section 3 we describe the architecture of the OWL-S/UDDI matchmaker and an updated mapping between OWL-S profile and UDDI. In Section 4 we present our efficient implementation of the matching algorithm, followed by experimental results comparing the performances of our OWL-S/UDDI Matchmaker implementation with a standard UDDI registry and finally we conclude.

2 UDDI & OWL-S

UDDI [8] is an industrial initiative aimed to create an Internet-wide network of registries of web services for enabling businesses to quickly, easily, and dynamically discover web services and interact with one another. OWL-S is an ontology, based on OWL, to semantically describe web services. OWL-S is characterized by three modules: *Service Profile*, *Process Model* and *Grounding*. Service Profile describes the capabilities of web services, hence crucial in the web service discovery process. For the sake of brevity of this paper, we are not going into the details of OWL-S and UDDI we assume that the readers are familiar with it, for more information see [3] and [8].

2.1 UDDI Search Mechanism

UDDI allows a wide range of searches: services can be searched by name, by location, by business, by bindings or by TModels. For example it is possible to look for all services that have a WSDL representation, or for services that adhere to Rosetta Net specification. Unfortunately, the search mechanism supported by UDDI is limited to keyword matches and does not support any inference based on the taxonomies referred to by the TModels. For example a car selling service may describe itself as “New Car Dealers” which is an entry in NAICS, but a search for “Automobile Dealers” services will not identify the car selling service despite the fact that “New Car Dealers” is a subtype of “Automobile Dealers”. Such semantic matching problem can be solved if we use OWL, RDF etc instead of XML.

The second problem with UDDI is the lack of a power full search mechanism. Search by Category information is the only way to search for services, however, the search may produce lot of results with may be of no interest. For example when searching for “Automobile Dealer”, you may not be interested in dealers who don’t accept a pre-authorized loan or credit cards as method of payments. In order to produce more precise search results, the search mechanism should not only take the taxonomy information into account but also the inputs and outputs of web services. The search mechanism resulted in combining the semantic base matching and the capability search is far more effective than the current search mechanism. OWL-S provides both semantic matching capability and capability base searching, hence a perfect candidate.

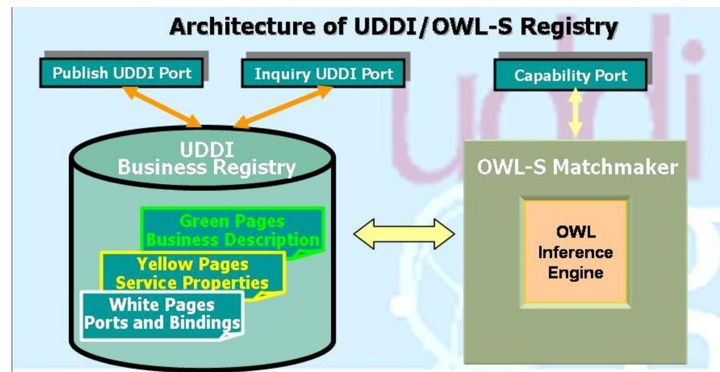

Fig. 1. Architecture of OWL-S / UDDI Matchmaker

3 OWL-S / UDDI Matchmaker Architecture

In order to combine OWL-S and UDDI, we need embed an OWL-S profile description in a UDDI data structure (we discuss this embedding in Section 3.1), and we need to augment the UDDI registry with an OWL-S Matchmaking component, for processing OWL-S profile information. The architecture of the combined OWL-S/UDDI registry

is shown in Fig 1. The matchmaker component in this architecture, unlike the previous version discussed in [2], is tightly coupled with the UDDI registry. By tightly coupled we mean the matchmaker component relies on the UDDI registry's ports (publish and inquiry) for its operations.

On receiving an advertisement through the publish port the UDDI component, in the OWL-S/UDDI matchmaker, processes it like any other UDDI advertisement. If the advertisement contains OWL-S Profile information, it forwards the advertisement to the matchmaking component. The matchmaker component classifies the advertisement based on the semantic information present in the advertisement.

A client can use the UDDI's inquiry port to access the searching functionality provided by the UDDI registry, however these searches neither use the semantic information present in the advertisement nor the capability description provided by the OWL-S Profile information. Hence we extended the UDDI registry by adding a *capability port* (see Fig 1) to solve the above problem. As a consequence, we also extended the UDDI API to access the capability search functionality of the OWL-S/UDDI matchmaker. Using the capability port, we can search for services based on the capability descriptions, i.e. inputs, outputs, pre-conditions and effects (IOPEs) of a service. The queries received through the capability port are processed by the matchmaker component, hence the queries are semantically matched based on the OWL-S Profile information. The query response contains list of *Business Service* keys of the advertisements that match the client's query. Apart from the service keys, it also contains useful information, like matching level and mapping, about each matched advertisement. The *matching level* signifies the level of match between the client's request and the matched advertisement. The *mapping* contains information about the semantic mapping between the request's IOPEs and the advertisement's IOPEs. Both these information can be used for selecting and invoking of an appropriate service from the results.

CategoryBag		
KeyedReference		
KeyName		Ticker Input
KeyValue	:	financialOntology:Ticker
TModelKey	:	UUID of OWL-S Input TModel
KeyedReference		
KeyName	:	Quote Output
KeyValue	:	financialOntology:quote
TModelKey	:	UUID of OWL-S Output TModel

Fig. 2. TModel for Stock Quote Service

3.1 Embedding OWL-S in UDDI

The OWL-S/UDDI registry requires the embedding of OWL-S profile information inside UDDI advertisements. We adopt the OWL-S/UDDI mapping mechanism described in [1]. The mechanism uses a one-to-one mapping if an OWL-S profile element has a corresponding UDDI element, such as, for example, the contact information in the OWL-S Profile. For OWL-S profile elements with no corresponding UDDI elements, it uses a T-Model based mapping. The T-Model mapping is loosely based on the WSDL-to-UDDI mapping proposed by the OASIS committee [13]. It defines special-

ized UDDI TModels for each unmapped elements in the OWL-S Profile like OWL-S Input, Output, Service Parameter and so on. These specialized TModels are used just like the way NAICS TModel is used to describe the category of a web service. Fig 2 illustrates an OWL-S/UDDI mapping of a Stock Quoting service whose input is a company ticker symbol and its output is the company's latest quotes.

In our work we extended the OWL-S/UDDI mapping to reflect the latest developments in both UDDI and OWL-S. Fig 3 shows the resulting OWL-S/UDDI mapping. Furthermore we enhanced the UDDI API with the OWL-S/UDDI mapping functionality, so that OWL-S Profiles can be converted into UDDI advertisements and published using the same API.

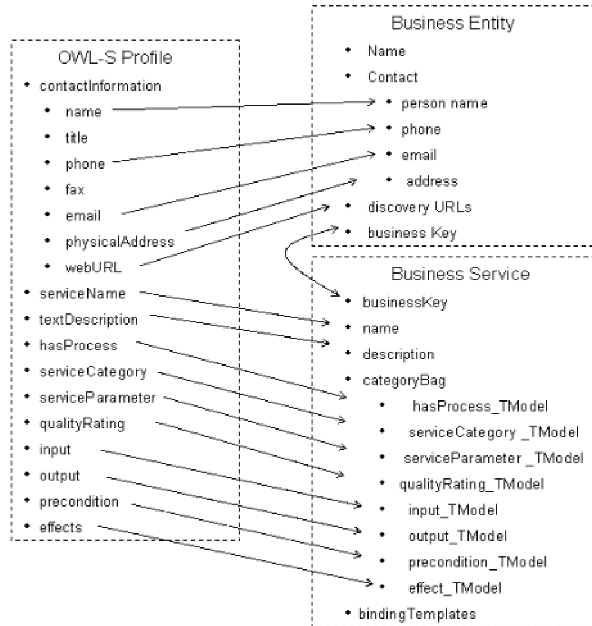

Fig. 3. Mapping between OWL-S Profile and UDDI

4 Achieving Matching Performance

A naive implementation of the matching algorithm described in [2] would match the inputs and the outputs of the request against the inputs and the outputs of all the advertisements in the matchmaker. Clearly, as the number of advertisements in the matchmaker increases the time taken to process each query will also increase. To overcome this limitation, when an advertisement is published, we annotate all the ontology concepts in the matchmaker with the degree of match that they have with the concepts in each published advertisement. As a consequence the effort need to answer a query is reduced to little more than just a lookup. The rational behind our approach is that since

the publishing of an advertisement is a one-time event, it makes sense to spend time to process the advertisement and store the partial results and speed up the query processing time, which may occur many times and also the query response time is critical. First we will briefly discuss the matching algorithm, then our enhancements in the publish and the query phase.

4.1 Matching Algorithm

The matching algorithm we used in our matchmaker is based on the algorithm presented in [2]. The algorithm defines a more flexible matching mechanism based on the OWL's subsumption mechanism. When a request is submitted, the algorithm finds a appropriate service by first matching the outputs of the request against the outputs of the published advertisements, and then, if any advertisement is matched after the output phase, the inputs of the request are matched against the inputs of the advertisements matched during the output phase.

In the matching algorithm, the *degree of match* between two outputs or two inputs depends on the match between the concepts that represents by them. The matching between the concepts is not syntactic, but it is based on the relation between these concepts in their OWL ontologies. For example consider an advertisement, of a vehicle selling service, whose output is specified as Vehicle and a request whose output is specified as Car. Although there is no exact match between the output of the request and the advertisement, given an ontology fragment as show in Fig 4, the matching algorithm recognizes a match because Vehicle subsumes Car.

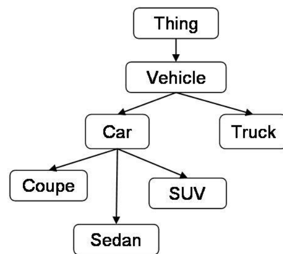

Fig. 4. Vehicle Ontology

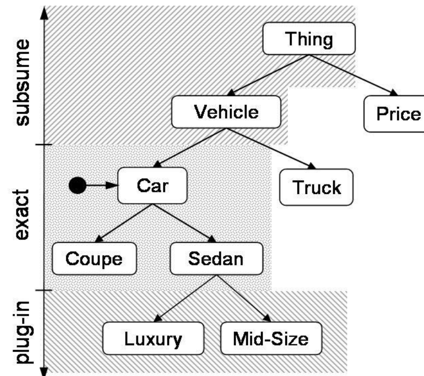

Fig. 5. Advertisement Propagation

The matching algorithm recognizes four degrees of match between two concepts. Let us assume OutR represents the concepts of an output of a request, and OutA that of an advertisement. The degree of match between OutR and OutA is as follows.

exact: If OutR and OutA are same or if OutR is an immediate subclass of OutA. For example given the ontology fragment like Fig 4, the degree of match between a request whose output is `Sedan` and an advertisement whose output is `Car` is exact.

plug in: If OutA subsumes OutR, then OutA is assumed to encompass OutR or in other words OutA can be plugged instead of OutR. For example we can assume a service selling Vehicle would also sell SUVs. However this match is inferior than the *exact* match because there is no guarantee that a Vehicle seller will sell every type of Vehicle.

subsume: If OutR subsumes OutA, then the provider may or may not completely satisfy the requester. Hence this match is inferior than the *plug in* match.

fail: A match is a *fail* if there is no subsumption relation between OutA and OutR.

4.2 Publishing Phase

Publishing of a Web service is not a time critical task; therefore we attempt to exploit this time to pre-compute the degree of match between the advertisement and possible requests. To perform this pre-computation, the matchmaker maintains a taxonomy that represents the subsumption relationships between all the concepts in the ontologies that it loaded. Each concept in this taxonomy is annotated with a two lists *output_node_ information* and *input_node_information* that specify to what degree any request pointing to that concept would match the advertisement. For example, output_node_information is represented as the following vector [<Adv1,exact>, <Adv2,subsume> , …], where AdvX points to the advertisement and "subsume" specify the degree of match. The advantage of the pre-computation is that at query time the matchmaker can extract the correct value with just a lookup with no need of inference.

More in details, at publishing time, the matchmaker loads the ontologies that are used by the advertisement's inputs and outputs and updates its taxonomy. Then, for each output in the advertisement, the matchmaker performs the following steps.

- the matchmaker locates the node corresponding to the concept, which represents the output, in the hierarchical structure let us call this node *curr_node*. The degree of match between the curr_node's concept and the output of the advertisement is *exact*, so the matchmaker updates the *output_node_ information*. Let us assume Fig 5 represents the hierarchical structure maintained by the matchmaker and let an output of an advertisement Adv_1 be 'Car'. The matchmaker updates the *output_node_information* of the 'Car' node that it matches Adv_1 *exactly*.
- The matchmaker updates the *output_node_ information* of all the nodes that are immediate child of the *curr_node* that the published advertisement matches them *exactly*. Because the algorithm states that the degree of match between output and the concepts immediate subclass are also *exact*. Following our example the matchmaker will updates the *output_node_ information* of the 'Coupe' node and the 'Sedan' node that it matches the advertisement Adv_1 exactly.
- The matchmaker updates the *output_node_ information* of all the parents of the *curr_node* that the degree of match between the nodes and the published advertisement is *subsume*. Following our example, we can see that the degree of match between the Adv_1's output concept 'Car' and the parent nodes of the *curr_node* 'Thing' and 'Vehicle' are *subsume*.
- Similarly the matchmaker updates the *output_node_ information* of all the child nodes of *curr_node* that the degree of match between the node and the published advertisement is *plug-in*. Following our example, we can see that the degree of

match the advertisement's output and the child nodes of *curr_node* 'Luxury' and 'Mid-Size' is *plug-in*.

Similar steps are followed, for each input of the published advertisement the matchmaker updates the *input_node_information* of the appropriate nodes.

As we can observe, we are performing most of the work required by the matching algorithm during the publishing phase itself, thereby spending a considerable amount of time in this phase. Nevertheless, we can show that the time spend during this phase, does not depend linearly on the number of concepts present in the data structure but in the order of *log (number of concepts)* in present in the tree structure, and hence showing that our implementation is scalable.

Since we use hierarchical data structure, the time required to insert a node will be in the order of $\log_d N$, where d is the degree of tree. Similarly time required to traverse between any two node in a particular branch will also be in the order of $\log_d N$. The time required for publishing an advertisement will be equal to the time required for classification of the ontologies used by inputs and outputs of the advertisement, plus the time required to update the hierarchical structure with the newly added concepts, plus the time required to propagate information about the newly added advertisement to the hierarchical structure. And in a best case scenario, when no ontology needs to be loaded, the publishing time will be time required for updating and propagating.

$$\text{Time}_{\text{publish}} = \text{Time}_{\text{Classification}} + \text{Time}_{\text{Update}} + \text{Time}_{\text{propagate}} \quad \textbf{(1)}$$

The time required by Racer for classifying neither directly depended on the number of concepts nor the number of advertisements present in the matchmaker. The time required by the other two operations, update and propagate, will be in the order of ($\log_d N$). Hence the publishing time does not linearly depend on the number of concepts or the advertisements present in the matchmaker.

4.3 Querying Phase

Since most of the matching information is pre-computed at the publishing phase, the matchmaker's query phase is reduced to simple lookups in the hierarchical data structure. We also save time by not allowing a query to load ontologies. Although loading ontologies required by the query appears to be a good idea, we do not allow it for the following three reasons: first, the loading of an ontology is an expensive process, furthermore the number of ontologies to load is in principle unbounded. Second, if the request requires the loading of a new ontology, it is very likely that the new concepts will have no relation with the concepts that are already present in the matchmaker, therefore the matching process would fail anyway. Third, the ontologies loaded by the query may be used only one time, and over time we may result is storing information about lot of unused concepts. Note that the decision of not loading ontologies at query time introduces incompleteness in the matching process: it is possible that the requested ontology bares some relations with the loaded ontologies, therefore the matching process may succeed. Still, the likelihood of this event is small, and the cost of loading ontologies so big that we opted for not loading them.

When the matchmaker receives a query it retrieves all *output_node_informations,* the sets of advertisements and its degree of match with the concept, of all the nodes

corresponding to the outputs of the request. For example, if the outputs of the request are ‘Car’ and ‘Price’, the matchmaker fetches the *output_node_informations* of car ONI_1 and of price ONI_2. The matchmaker then finds the advertisements that are common between the sets of advertisements retrieved,i.e. $ONI_1 \cap ONI_2$. If no intersection is found then the query fails. If common advertisements are found say ADVSo, they are selected for further processing.

The matchmaker performs a lookup operation and fetches all the *input_node_informations*, the sets of advertisements and degree of match with the concept, of all the nodes corresponding to the inputs of the request. The matchmaker keeps only the input_node_information of the advertisements that were selected during the output processing phase, other advertisements are discarded. For example let IN_1, IN_2,IN_3 be the input_node_information, then only input_node_information of advertisements $ADVSo \cap IN_1 \cap IN_2 \cap IN_3$ are kept. This input_node_information and match level of each output is used to score the advertisements that were selected during the output processing phase, i.e. ADVSo.

We can see that the time required for processing a query does not depend on the number of advertisements published in the matchmaker. As we also see the querying phase involves lookups and intersections between the selected advertisements. In our implementation lookups can performed in constant time. Hence time to process a query depends on the time to perform intersections between the selected advertisements.

$$Time_{query} = (Num_{out}+Num_{in})*(Time_{Lookup} + Time_{Intersection}) \quad (2)$$

where the Num_{out} and Num_{in} is the number of inputs and outputs of an advertisement, $Time_{Lookup}$ is the time required to extract information about an input or an output, and $Time_{Intersection}$ is the time to compute intersection between the lists extracted during the lookup time. We can see that the computation required for the querying process does not depend on the number of advertisements, and therefore it is scalable.

	Time ms	Standard Deviation
UDDI	163.98	86.17
OWL-S/UDDI	1050.77	167.96

Table 1. Publishing Time without loading ontologies

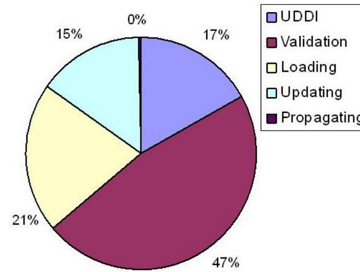

Fig. 6. Time distribution during publishing an advertisement

5 Preliminary Experimental Results

We conducted some preliminary evaluation comparing the performances of our OWL-S/UDDI registry and a UDDI registry, to show that adding an OWL-S match-

maker component does not hinder the performance and scalability of a UDDI registry. We extended jUDDI [14] an open source UDDI registry with the OWL-S matchmaking component. We used RACER [4] as to perform OWL inferences. In our experiments, we measured the processing time of an advertisement by calculating the difference between the time the UDDI registry receives an advertisement and the time the result is delivered, to eliminate the network latency time.

5.1 Performance – Publishing Time

In our first experiment we compared the time take to publish an advertisement in an OWL-S/UDDI registry and in a UDDI registry. We assumed that the ontologies required by the inputs and outputs of the advertisements are already present in the OWL-S/UDDI registry. The advertisements may have different inputs and outputs but they are present in one ontology file, hence the ontology has to be loaded only once, however our registry still have to load 50 advertisements. Table 1 shows the average time taken to publish 50 advertisements in a UDDI registry and an OWL-S/UDDI registry. We can see that the OWL-S/UDDI registry spends around 6-7 times more time, since publishing it a one-time event we are not concerned about the time taken.

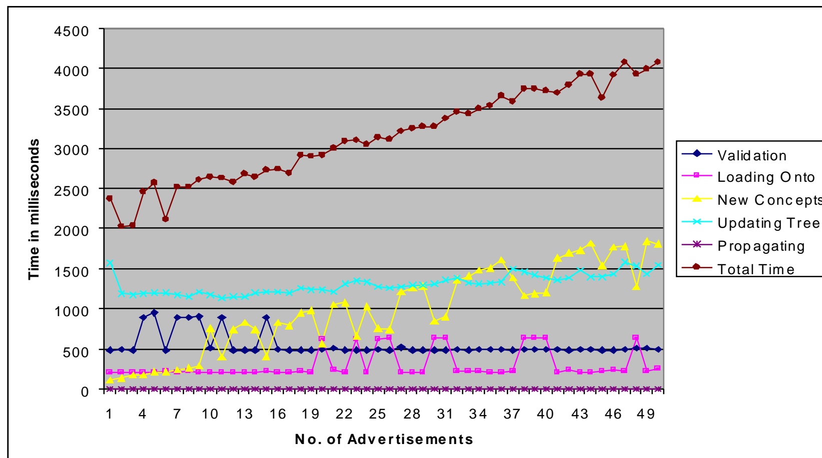

Fig. 7. Publishing time for advertisements that requires to load new ontologies

However, we took a closer look at the time taken to publish an advertisement by the OWL-S/UDDI registry. Fig 6 shows the time spent in different phases of the publishing an advertisement. Following are the 5 phases in the publishing process:

- UDDI - time required by the UDDI component to process an advertisement.
- Validation - time required by Racer to validate the advertisement.
- Loading - time required by Racer to load the advertisement
- Updating - time required to extract the ontology tree from Racer.
- Propagating - time required to propagate the input/output information.

As we can see, most of the time is spent in loading and validating ontology (around 70%) when compared to the matchmaking operations.

5.2 Performance – Ontology Loading

In the second experiment, we analyzed the performance of our registry when we load advertisements that required loading new ontology and hence significantly updating the taxonomy maintained by the matchmaking component. We published 50 advertisements that uses different ontologies to describe their inputs and outputs, in our OWL-S/UDDI registry and measured the time taken to publish each advertisement. Each of these advertisements has three inputs and one output and requires loading an ontology containing 30 concepts.

In Fig.7, we can see that the time take to publish an advertisement increases linearly with the number of advertisements, and we can also see that this linear increase is contributed by 'new-concept'. This linear increase of 'new-concept' is attributed to a limitation of the Racer system. Whenever we load a new ontology into Racer we have determine if we need to update the taxonomy maintained by the matchmaker, if so, what concepts should be updated. The Racer system does not provide any direct means to give this information. Hence we need to find out this information through a series of interactions. The new-concept in Fig 7 represents the time required to perform this operation. We can substantially reduce the time required for publishing if either Racer can provide the information directly or if we could have direct access to Racer and we maintain the taxonomy inside Racer itself. We can see that if ignore the time taken by 'new-concept', the resulting graph would not have such drastic increase in the publishing time, concurring to our discussion in Section 4.2.

	Time in ms	Standard Deviation
OWL-S/UDDI	1.306	.54

Table 2. Query processing time

5.3 Performance – Querying Time

In our final experiment, we calculated the time required to process a query. The queries we used do not load new ontologies into the matchmaker, they use the ontologies that are already present in the matchmaker. We used 50 queries each with three inputs and one output. Table 2 shows the average time required to process these queries. The small standard deviation shows that the time required to process the queries is almost constant, consistent with our discussions in Section 4.3

6 Conclusion

In this paper we have described the importance of web service discovery and the shortcomings of the UDDI's discovery mechanism. We adapted a solution to use

OWL-S in combination with UDDI, to take advantage of both these technologies. We believe such an architecture is very important in bring the effort of both Web Service and Semantic Web community together. We presented our OWL-S/UDDI matchmaker architecture and its extensions to perform capability search. We also conducted some preliminary experiments to show the scalability of our implementation.

References

1. Paolucci et al: Importing the Semantic Web in UDDI. In Proceedings of Web Services, E-business and Semantic Web Workshop, 2002
2. Paolucci et al; Semantic Matching of Web Services Capabilities. In Proceedings of the 1st International Semantic Web Conference (ISWC2002)
3. Anupriya et al: DAML-S: Web Service Description for the Semantic Web. In Proceedings of The First International Semantic Web Conference (ISWC), 2002.
4. Volker Haarslev, Ralf Möller: RACER System Description. In Proceedings of International Joint Conference on Automated Reasoning, IJCAR'2001, June 18-23, 2001, Siena, Italy.
5. Tim Berners-Lee and James Hendler and Ora Lassila: The Semantic Web. Scientific American ,volume 284, Number 5, pages 34-43, 2001
6. Katia Sycara et al, "Larks, Dynamic matchmaking among Heterogeneous Software Agents in Cyberspace", AAMAS, 5, 173-203, 2002.
7. W3C: Extensible Markup Language (XML) 1.0 (Second Edition). http://www.w3.org/ TR/2000/REC-xml-20001006,2000,
8. UDDI : The UDDI Technical White Paper, http://www.uddi.org, 2000
9. Rosetta Net. http://www.rosettanet.org, 2000
10. W3C, "SOAP Version 1.2, W3C Working Draft 17 December 2000", http://www.w3.org/ TR/2001/WD-soap12-part0-20011217/ , 2001
11. Erik Christensen et al , "Web Services Description Language (WSDL) 1.1", http://www.w3.org/ TR/2001/NOTE-wsdl-20010315, 2001
12. ISO/IEC 11578:1996 : Information technology -- Open Systems Interconnection -- Remote Procedure Call. http://www.iso.ch/ , 2001.
13. Colgrave et al : Using WSDL in a UDDI Registry, Version 2.0., UDDI TC Note, 2003.
14. jUDDI : http://ws.apache.org/juddi/
15. W3C : Web Ontology Language. http://www.w3.org/2001/sw/WebOnt/