A Broker for OWL-S Web services

Massimo Paolucci, Julien Soudry, Naveen Srinivasan and Katia Sycara

The Robotics Institute, Carnegie Mellon University

Abstract

Brokers are widely used in distributed information systems such as Multi-agent systems and distributed databases. Yet, there has not been a detailed analysis of Brokers' architecture and no general solution has been proposed on how the Brokers' tasks have to be accomplished. In this paper, we provide a detailed analysis of these tasks, and an implementation based on OWL-S. We show that while OWL-S is adequate to provide all the information that is needed by the Broker, the straightforward implementation of the Broker using OWL-S results in a paradoxical situation. We solve this paradox by extending the Process Modeling language of OWL-S. Finally, we propose a solution to a number of issues that arise in the brokered management of the interaction between Web services such as the abstraction from queries to capabilities required to solve that query, and management of the knowledge required by the Broker to control the multi-party interaction.

Introduction

Brokers facilitate the interaction between two or more parties. For example, if two parties want to communicate, but they do not share a common language, Brokers may provide translation services, or if the two parties do not trust each other, a Broker may provide a trusted intermediary (e.g. an escrow service for e-commerce transactions). Furthermore, Brokers may provide anonymization for one (or both) of the parties, by mediating the transaction.

Not surprisingly, Brokers are one of the main discovery and synchronization mechanisms among autonomous agents [9][26]. Examples include the OAA Facilitator [18] which Brokers between OAA agents that collaborate toward the solution of a problem. Furthermore, Brokers have been widely used in many agents applications such as integration of heterogeneous information sources and Data Bases [16], e-commerce [14] [11], pervasive computing [6] and more recently in coordinating between Web services in the IRS-II framework [20]. Finally, theoretical studies [9] [26] show that Brokers can perform a range of coordination activities such as load balancing between different agents, or anonymizing between requesters and providers.

Because of its properties and its wide applicability, a Broker would be a natural candidate component for the Web Services infrastructure. However, the current Web services architecture [4] does not include Brokers with rich functionality of *discovery* and *mediation*, as part of the Web Services infrastructure.

In this paper, we provide an analysis of the requirements of a Broker that performs both discovery and mediation between agents and Web services. We show that such a Broker performs very complex reasoning tasks that include (1) the interpretation of the capability advertisements of service providers; (2) the interpretation of the requesters' queries that must be fulfilled by a service provider; (3) finding the best provider based on the requester's query; (4) invocation of the selected provider on behalf of the requester, interacting with the provider as necessary to fulfill the query, and (5) returning the query results to the requester. The accomplishment of these tasks requires ontologies to describe capabilities of Web services, their interaction patterns and the domain they operate on, and a logic that allows reasoning on those ontologies. Furthermore, we will provide a description of our implementation of a Broker using OWL-S [21].

The rest of the paper is organized as follows. In section 2, we present an overview of OWL-S. In section 3, we provide a detailed analysis of the Broker, exploring its interaction protocol and the reasoning tasks it has to accomplish. In section 4, we show how the current OWL-S specification supports the reasoning of the Broker and where this specification falls short. In section 5, we provide extensions to OWL-S to address some of the shortcomings of the current specification as regards support for Broker's reasoning tasks. In particular, we describe the *exec* extension of OWL-S. In section 6, we describe the basic features of our implementation and provide details on how we address the reasoning problems of the Broker. In section 7, we conclude.

OWL-S

OWL-S [21] is a Semantic Web Services description language that enriches Web Services descriptions with semantic information from OWL [8] ontologies and the Semantic Web [3]. OWL-S is organized in three modules: a *Profile* that describes capabilities of Web Services as well as additional features that help to describe the service. A *Process Model* that provides a description of the activity of the Web Service provider from which the Web Service requester can derive information about the service invocation. A *Grounding* that is a description of how

Seq	-	$\Pi,E[\text{return } v >>= e],\varphi) \rightarrow \Pi,(E[(e\ v)],\varphi)$
Spawn	-	$\Pi,(E[\text{spawn } e],\varphi) \rightarrow \Pi,(E[\text{return}()],\varphi),(e,\varnothing)$
$Cond^{True}$	-	$\Pi,(E[\text{cond } C\ e_1\ e_2],\varphi) \rightarrow \Pi,(E[e_1],\varphi)$
$Choice^{Left}$	$\Pi,(E[e_1],\varphi) \rightarrow \Pi',(E[e_1'],\varphi')$	$\Pi,(E[\text{choice } e_1\ e_2],\varphi) \rightarrow \Pi',(E[e_1'],\varphi')$
Atomic[1]	-	$\Pi,(E[\text{atomic } e],\varphi) \rightarrow \Pi,(E[\text{return } ()],\varphi')$

Table-1. Execution Semantics of OWL-S control structures

abstract information exchanges described in the Process Model are mapped onto actual messages that the provider and the requester exchange.

A Web Service *capability* is the description of the service functionality, i.e. what the service does. For example, the capability of Barnes and Noble, a bookseller, is to sell books. The capability of a Web Service can be viewed in two ways: first as a service category within an ontology of services (e.g. selling books is-a selling products) or as a transformation of a set of inputs to a set of outputs (e.g. selling books transforms the inputs "book title" and "book author" to the output "book invoice"). The OWL-S Profile describes capabilities of Web Services by the transformation that they produce. In order to make its capabilities known to service requesters, a service provider advertises its capabilities with infrastructure registries, or more precisely *middle agents* [26], that record which agents are present in the system. UDDI [25] is an example of a middle agent, that can make only limited use of the information provided by the OWL-S Profile. The OWL-S/UDDI Matchmaker [22] [23] is another example, which combines UDDI and OWL-S. Finally, the Broker defined in this paper is another example of a middle agent that performs both discovery and mediation.

The second module of OWL-S is the Process Model. The Process Model has two aims: the first one is to show how the provider achieves its goals, and the second to provide the *requester-provider interaction protocol.* The first goal is achieved by allowing the provider to make public a description of its computation, to the extent that the provider feels comfortable to do so. OWL-S distinguishes between two types of processes: composite processes and atomic processes. Atomic processes correspond to operations that the provider can perform directly. Composite processes are used to describe collections of processes (either atomic or composite) organized on the basis of some control flow structure. For example, a *sequence* of processes is defined as a composite process whose processes are executed one after the other. Other control constructs supported by OWL-S are *cond* for conditional expressions, *choice* for non-deterministic choices between alternative control flows, and *spawn* for spawning a new concurrent thread. Finally, OWL-S includes looping constructs like *while* and *repeat-until.*

The execution of a process produces a state transition where either some information is exchanged with some partner, or the agent produces a change in the environment. A state is defined as a tuple (φ,Π) where Π represents the set of concurrent threads, and φ the state of the thread the process is executed in [1][2]. Processes modify the state by either changing the state of their thread φ, for instance, an atomic process may read a message from a port, or modify the set of concurrent threads Π through the spawning of new threads or the closing of other threads. The formal semantics of the OWL-S composite and atomic processes is shown here in Table 1[3]. Looping constructs are implemented as combinations of sequences and conditions.

Each rule in Table 1. specifies how the execution of a process changes the overall state. Sequences of processes, expressed here by the temporal constraint *return v >>=e*, applies *e* to the results *v* of the previous step. The execution of a spawn operation, results in the beginning of the execution of a new thread $(e,\varnothing)$, while it returns no value in the current thread (return ()). The other rules specify the result of executing other types of control constructs, *$Cond^{True}$* specifies the results of the execution of a conditional statement if the condition is true; a similar rule would be used for a false condition. *$Choice^{Left}$* specifies the results of the execution of a non-deterministic selection of the first process of a list; a similar rule would be used for other choices. Finally, *Atomic* describes the results of executing an atomic process, which has an effect on the state of the current thread φ but it does not modify the set of concurrent processes $\prod$.

The last module of OWL-S is the Grounding that describes how atomic processes which provide abstract descriptions of the information exchanges with the requesters are transformed into concrete messages or remote procedure calls over the net. Specifically, the OWL-S Grounding is defined as a one to one mapping from atomic processes to WSDL [5] input and output message specifications

[1] The execution semantics presented in [1] does not include an explicit notion of atomic process, rather atomic processes are constructed as a combination of operations that receive messages, send messages, and apply functions.

[2] The execution semantics that we use was originally proposed for DAML-S 0.6. While many aspects of the language changed in the evolution to OWL-S 1.0 that we use here, the execution semantics of the basic constructs of the Process Model is still valid.

[3] We provide here a very brief explanation of the OWL-S execution semantics. A complete presentation is in [1].

The Web Services philosophy of interaction between a service requester and a service provider is that a requester would need to know the information that a service provider requires at different stages of the interaction. For example, in industrial standards, the requester-provider interaction is governed by knowledge of the provider's Web Services Description (WSD) given in WSDL, and in Semantic Web Services, the requester-provider interaction presupposes knowledge on the part of the requester of the Process Model (plus WSD) of the provider.

Overview of the Broker

Any transaction involving a Broker requires three parties. (Figure 1). The first party is *a requester* that initiates the transaction by requesting information or a service to the Broker. The second party is *a provider* which is selected among a pool of provider as the best suited to resolve the problem of the requester. The last party is the *Broke*r itself.

The protocol in Figure 1 can be divided in two parts: the advertisement protocol, and the mediation protocol. In the advertisement protocol, the Broker first collects the advertisements of Web services that are available to provide their services. These advertisements, shown in Figure 1 by straight thin lines, are used by the Broker to select the best provider during the interaction with the requester. The mediation protocol, shown in Figure 1 using thick curve lines, requires (1) the requester to query the Broker and wait for a reply while the Broker uses its discovery capabilities to locate a provider that can answer the query. Once the provider is discovered, (2) the Broker reformulates the query for that provider, and finally queries it. Upon receiving the query, (3) the provider computes the reply to the Broker and finally (4) the Broker replies to the requester.

The protocol described above shows that the Broker needs to perform a number of complex reasoning tasks for both the discovery and mediation part of its interaction. The discovery process requires two different reasoning tasks. The first one is to *abstract from the query of the requester to the capabilities required by a provider* in order to answer that query. The second process is to *compare/match* the capabilities required to answer the query with the capabilities of the providers to find the best provider for the particular query.

The mediation task of the Broker requires that the Broker must transform the query of the requester into a query to send to the provider. This process of mediation has two aspects. The first one is the efficient use of the information provided by the requester to the Broker, the second one is the mapping from the messages of the requester to messages to the provider and vice versa.

Since the requester does not know which is the relevant provider, the (initial) query it sends to the Broker and the query input that the (selected) provider may need in order to provide the service may not correspond exactly.

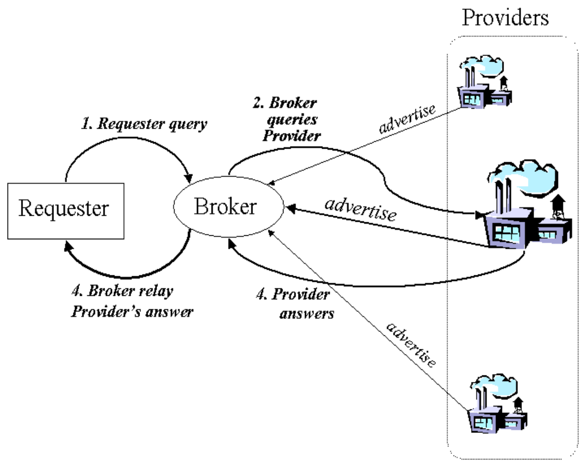

Figure-1. The Broker's Protocol

Consider the example of a requester that asks to book the cheapest flight from Pittsburgh to New York. Besides the trip origin and destination, the selected provider may expect date and time of departure. In the example, the requester never provided the departure time, and the provider has no use for the "cheaper" qualifier. It is the task of the Broker to reconcile the difference between the information that the requester provided and the information that the provider expects, by (1) recognizing that the departure time was not provided, and therefore it should be asked for, and (2) finding a way to select the cheapest flight among the ones that the provider can find.

Moreover, the Broker may have to perform the mapping between ontologies and terms used by the two parties. For example, the requester may have asked for information on IBM whereas the provider expects inputs in terms of International Business Machine Corporation. Another, more complicated mismatch may be at the level of concepts and their relations in the ontologies used for inputs and outputs of the provider vis a vis the ontological information used by the requester. For example, the requester may have asked for the weather in Pittsburgh, but instead the provider can report only the weather at major airports. The task of the Broker in this case is to infer which is the most appropriate airport, and use it in the query to the provider. Therefore, instead of asking for the weather in Pittsburgh, the Broker asks the provider for the weather at PIT, where PIT is the code of the Pittsburgh International Airport.

Finally, the Broker has the non-trivial task of translating between the different syntactic forms of the queries and replies. The examples that we discussed above assume semantic mismatches between the different messages that the Broker has to interpret and send. These messages have to be compiled in an appropriate syntactic form, and despite their semantic similarity, the messages would be realized in very different ways. The task of the Broker is to resolve syntactic differences, and to formulate messages that all the parties can understand.

In conclusion, the Broker performs a number of complex reasoning tasks that range from discovery to the interpretation, translation and compilation of messages. To accomplish these tasks, the Broker needs the support of a formal framework that allows complex reasoning about agents, what they do and how to interact with them.

OWL-S Support for the Broker

The OWL-S language and ontology provides constructs to support the Broker in both discovery and mediation between Web services. The OWL-S Profile supports the discovery process by providing a representation of capabilities of Web services and agents. The OWL-S Process Model and Service Grounding provide support for the interaction between the Broker and the requester and provider of the service.

The Service Grounding provides a mapping from the semantic form of the messages exchanged as defined in the Process Model, to the syntactic form as defined in the WSDL input and output specifications. The Grounding provides to the Broker the mapping from the abstract semantic representation of the messages to the syntactic form that these messages adopt when they become concrete information exchanges. The Broker uses this mapping to interpret the messages that it receives and compile the messages that it sends to the requester or to the provider.

A number of capability matching algorithms for OWL-S based Web services have been proposed (see [2][10][15][22]) which exploit OWL ontologies and the related logics to infer which advertisements satisfy a request for capabilities. These algorithms can be used to solve the problem of matching from the capabilities required for the query to the capabilities of the available providers.

The abstraction from the requester's query to the capabilities required, is more complicated. First of all, there is no explicit support in OWL-S for queries, nevertheless, it is easy to use the OWL Query Language (OWL QL) [7][12] which relies on the same logics required by OWL-S. The transformation is still an open problem, which, to our knowledge, has never been addressed. In section 6.1, we will propose an abstraction algorithm to transform queries into capabilities.

After selecting a provider, the Broker has access to the provider's Process Model from which it can derive the provider's interaction protocol by extracting what information the provider will need, in what order, and what information it will return. For the rest of the interaction the Broker acts as the provider's direct requester. However, this relation is not straightforward. Since the Broker acts on behalf of the requester, it must somehow transform the requester's initial query (and all subsequent messages) into a query (or a sequence of queries) to the provider. This transformation is necessary since the requester cannot "see" directly the Process Model of the provider, but interacts with the provider only through the Broker. We show how this transformation can be done in section 6.2.

Furthermore, since the requester initiated its query without having access to the provider's Process Model (since the provider was not known at the time of the requester's query initiation), the Broker needs to infer what additional information it needs from the requester. Once it has done that, it then uses this knowledge to construct a new Process Model. This new Process Model is presented by the Broker to the requester, not as the Process Model of the selected provider but as the process Model of the Broker. This makes sense since the requester interacts only with the Broker. The new Process Model indicates to the requester what information is needed and in what order. How the Broker infers the additional information it needs from the provider and how it constructs the new Process Model is presented in section 6.2.

Since, to the requester, the Broker is a (representative of) the provider, the Process Model of the Broker should contain the crucial elements of the Process Model of the provider. However, since the Broker is unaware of the provider until it has discovered and selected the provider based on a requester's query, the Broker is faced with a challenge: it must publish a Process Model that depends on the provider's Process Model, but the provider is not known until the requester reveals its query. On the other hand, the requester cannot query (interact with) the Broker until the Broker publishes its Process Model. The result is a paradoxical situation in which the Broker cannot reveal its Process Model until it receives the query of the requester, but cannot receive the query from the requester until it publishes its Process Model.

Essentially, the Broker paradox results from an inflexibility of the OWL-S specification of service invocation, which requires the specification of the Process Model before the interaction, and it does not allow any means to modify the Process Model during the interaction.[1]

Extending OWL-S

The solution of the Broker's Paradox that we propose requires an extension of the specification of the OWL-S Process Model to allow the flexibility to dynamically modify an agent's Process Model during the interaction. As a result, the Broker can provide an initial, provider-neutral, Process Model to the requester, and then modify it

[1] The current industry proposed standards have the same inflexibility, since the Web services Description must be specified once and for all with no provisions for on-the-fly loading or modification.

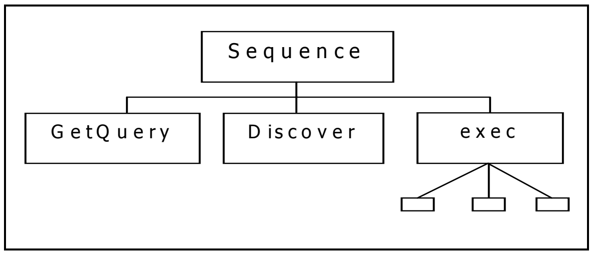

Figure-2. Broker's Process Model

consistently with the requirements of the Process Model of the provider. This results in the New Process Model, which the requester uses in its interactions with the Broker.

To implement this solution, we propose to extend the OWL-S Model Processing language by adding a new statement, that we call *exec.* The *exec* statement takes as input a Process Model and executes it. Therefore, the Broker can compile a new Process Model, return it as an output of one of its processes, and then use the *exec* to turn the new Process Model into executable code that specifies the Broker's new interaction protocol.

The provider-neutral Process Model of the Broker is shown in Figure 2. It shows that the Broker performs a sequence of three operations. The first operation is `GetQuery` in which the Broker gets the query from the requester. The second operation is `Discover` in which the Broker uses its discovery capabilities to find the best provider. The result of the Discover process is a new Process Model that depends on the provider found. Finally, the Broker performs the *exec* operation which passes control to a new Process Model. This change of control is shown in the figure by the three small rectangles that display processes that will be run as a consequence of the *exec*.

The use of the *exec* solves the Broker's Paradox by removing the inflexibility of the OWL-S Process Model. The *exec* operation allows the separation of service discovery from service invocation and interaction. First the discovery is completed, then the interaction, which depends on the discovered provider, is initiated through the *exec*.

One important question that is left unanswered is whether there is a clever way to use OWL and OWL-S that does not require the extension of the language that we propose. Unfortunately, such an extension does not exist, because neither OWL nor OWL-S provides a way to transform a term into a predicate of the logic, which is the essential step that is performed by the *exec*.

Formal Semantics of exec

Intuitively, the semantics of the *exec* operation is to execute the processes that it contains as arguments. In other words, the state transformation produced by *exec*(P) is equivalent to the state transformation produced by the direct execution of P. This intuition is captured by the axiomatic semantics of *exec*, described in Table 2, which is a natural extension of the axiomatic execution semantics of OWL-S shown in Table 1.

The execution of an *exec* statement is shown in Table 2. This rule specifies that the execution of *exec(P)* in the state (∏,φ) should produces the same results that are produced by the execution of P in the same state in the state (∏,φ). This definition allows us to transform the specification of a process P into the execution of the process, which is exactly what we are seeking with the definition of *exec*.

$$\text{exec(P)} \quad \frac{\Pi,(E[P],\varphi)\rightarrow\Pi',(E[P'],\varphi')}{\Pi,E[\text{exec}(P)],\varphi)\rightarrow \Pi',(E[P'],\varphi')}$$

Table-2. The execution semantics of the *exec* statement

Broker Implementation

We have implemented a prototype of a Broker that makes use of OWL-S with the *exec* extension described above to mediate between agents and Web services. We based our implementation of the Broker on the OWL-S Virtual Machine (OWL-S VM) [24], which is a generic OWL-S processor that allows Web services and agents to interact on the basis of the OWL-S description of the Web service and OWL ontologies. In the implementation of the Broker, we extended the OWL-S VM to include the semantics of the *exec*. Furthermore, we developed the reasoning that allows the Broker to perform discovery and to mediate the interaction between the provider and the requester.

Broker-based discovery

The Broker expects from the requester a query in OWL-QL format [12], where the predicate corresponds to a property in the ontology, the terms in the query are either variables, or instances that are consistent with the semantic type requirements of the predicate.

The discovery process takes as input the query of the requester and generates as output the advertisement of a provider (if any is known to the Broker) that can answer the query. The discovery process has three steps. First the Broker abstracts from the query to the capabilities that are required to answer that query, thus constructing a service request. Second, the Broker finds appropriate providers by matching the capabilities required to solve the query (the service request) with the capability advertisements of providers. Third, the Broker uses similarity of the match of the service request and the returned advertisements as well as other parameters in the returned Service Profiles to select the most appropriate provider. The matching of the service request against the advertised capabilities was implemented using the OWL-S matching engine reported in [22] and [23].

The automatic abstraction from the requester's query to a service request is, to our knowledge, an unexplored problem. The abstraction process must respect the constraints of the OWL-S discovery process, namely generation of an OWL-S service profile with the appropriate required service inputs and outputs that (1) reflected the semantic content of the query and (2) reflected the requirements of the generated service request.

```
1. set V = set of variables in the query
2. set T= set of instantiated terms in the query
3. set I= abstraction of each term in T to its immediate
   class
4. use predicate definition in the ontology to abstract
         variables in V to their class
5. set O= abstraction of each variable in V to its class
6. generate a service request  with input I and outputs O
```

Figure 3: The abstraction algorithm

The instantiation algorithm follows the 6 steps listed in Figure 3. In the steps 1 and 2, terms from the query are extracted distinguishing between variables and instantiated terms. In step 3, the set of inputs of the service request is derived by abstracting the instantiated terms to their immediate class. For instance, if one term were Pittsburgh, it would be abstracted to City (assuming the presence of a location ontology). Step 4 is needed to handle variables. In OWL-QL variables are of class Variable, but there is no constraint on the type that they have to assume. We use the definition of the predicate in the ontology to constrain the type of the values of the variable to the most restrictive class of values that they can be assigned to. In step 5, we use the abstraction in step 4 to generate the set of outputs O. Finally, in step 6, the service request is generated by specifying the inputs and the outputs[1].

```
1. KB= knowledge from query
2. I= input of process
3. for i∈I
4.     select k from KB with the same semantic type of I
5.     if  k exists
6.         remove i from I
```

Figure-4. Algorithm for pruning redundant information

Broker-based mediation

After the Broker has selected a provider, it must mediate between the provider and the requester. The mediation process depends on the Process Model of the provider which specifies what information is required and when. In theory, the Broker may just present to the requester the Process Model of the provider and limit mediation to message forwarding. But this solution is very inefficient, since it ignores the information that the requester already provided to the Broker. For example, the requester may ask the Broker to book a trip to Pittsburgh. The Broker may find a Travel Web service that asks for departure and arrival location. The task of the Broker is to recognize that arrival location information has already been specified so the Broker needs to ask the requester for the departure location only.

The algorithm for pruning redundant information is shown in Figure 4. First, the Broker records the information provided by the query in a KB (step 1), and the inputs of the process (step 2). Next for each input i, the Broker looks in the KB for information that it can use in place of i. If any is found, i is removed from the inputs of a process.

Broker-based Interaction

The architecture of the Broker is shown in Figure 5. To interact with the provider and the requester the Broker instantiates two ports: a server port for interaction with the requester (since the Broker acts as a provider vis a vis the requester) and a client port for interaction with the provider (since the Broker acts as a client vis a vis the provider). The functionalities of the server port are described using OWL-S. Specifically, the Broker exposes to the requester its Process Model, Grounding and WSDL specification. The client (requester) uses these descriptions to instantiate an OWL-S Virtual Machine to interact with the Broker. Since the provider-neutral Process Model exposed by the Broker makes use of the exec extension described in section 5, the OWL-S Virtual Machine used by the requester also includes an implementation of the axioms for exec that we presented in section 5.1. The client port is also implemented as an OWL-S Virtual Machine that uses the Process Model, Grounding and WSDL description of the provider to interact with it.

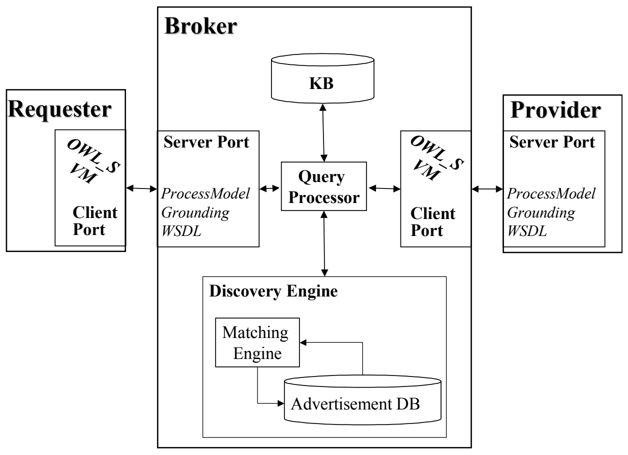

Figure-5. Broker's Architecture

The reasoning of the Broker happens in the *Query Processor* (see Figure 5) that is responsible for the

[1] Inputs and outputs are the most important information for matching; if the query includes additional information, this could also be abstracted. Currently, we did not concern ourselves with this issue.

translation of the messages between the two parties and for the implementation of the algorithms in Figures 3 and 4. Specifically, the Query Processor stores information received from the query in a *Knowledge Base* to be used as needed during the execution. Furthermore, the Query Processor interacts with the *Discovery Engine,* which provides the storage and matching of capabilities*,* when it receives a capability advertisement and when it needs to find a provider that can answer the query of the requester.

Conclusion

Despite the wide use of Brokers in different aspects of distributed systems, and despite the many uses Brokers can have in the discovery and mediation of Web Services, no detailed analysis of what tasks a Broker should carry on has been proposed. One contribution of this paper is to provide such as analysis. In the course of this analysis, a few challenges were uncovered, and solutions for these challenges were presented.

The first of the challenges is the "Broker's paradox", namely that the Broker cannot publish a Process Model that is based on a yet unknown provider before it receives a request query but the requester cannot send a query until it knows the Broker's process Model. This paradox arises from the OWL-S (and WSDL among others) Web Service interaction specification that is based on the declarative specification of a process model that guides the requester and provider interaction. To address the Broker paradox, we extended the OWL-S Process Modeling language with an exec operation that allows the dynamic modification of the Broker's Process Model during its execution to include Process Models of dynamically discovered new parties. We provide a formal semantics for the exec operator that is grounded in the formal execution semantics of OWL-S, and we show how it can be used as a basis for the use of OWL-S to represent the interactions of more than two parties.

A second set of challenges derives from the management of the mediation between the provider and the requester. To address these challenges, we developed a method for abstracting from a service query to a service request. We proposed an algorithm to address this issue. Furthermore, we provided an algorithm for the Broker to make efficient use of the knowledge provided by the requester during the interaction with the provider.

Crucially, the issues emerging with the mediation between the provider and the requester are not unique to Web services Brokering, rather they comes up in web services composition as well. In the context of Web service composition, a planner may issue a goal that it wants to subcontract. The task of the Web service is first to abstract from the specific goal to a capability description of a provider that can solve the goal, then use its current knowledge, and the goal, to interact with the provider. In current research, we are looking to integrate our work in the context of Brokering to automated composition.

Acknowledgements

We would like to thank Khalid El-Arini for his contribution to early development of this work. This work has been was funded by the Defense Advanced Research Projects Agency as part of the DARPA Agent Markup Language (DAML) program under Air Force Research Laboratory contract F30601-00-2-0592 to Carnegie Mellon University.

References

[1] Ankolekar, A, Huch, F, and Sycara, K. "Concurrent Execution Semantics for DAML-S with Subtypes." In *The First International Semantic Web Conference*, 2002.

[2] Benatallah, B, Hacid, M, Rey, C, and Toumani F. "Towards Semantic Reasoning for Web Services Discovery", *In Proc. of the International Semantic Web Conference (ISWC'03)*, Springer Verlag, Sanibel Island, Florida, USA Oct 2003.

[3] Berners-Lee, T, Hendler, J, and Lassila, O. "The semantic web" *Scientific American*, 284(5):34--43, 2001.

[4] Booth, D., Haas, H., McCabe F., Newcomer, E., Champion, M., Ferris, C., Orchard. D. "Web Services Architecture, W3C Working Draft 8 August 2003", http://www.w3.org/TR/2003/WD-ws-arch-20030808/

[5] Christensen, E, Curbera, F, Meredith, G, and Weerawarana, S.: *Web Services Description Language*: http://www.w3.org/TR/2001/NOTE-wsdl-20010315 2001.

[6] Chen, H, Finin, T, and Joshi, A. "Semantic Web in the Context Broker Architecture", In *Proceedings of the IEEE Conference on Pervasive Computing and Communications (PerCom),* Orlando, March, 2004.

[7] DAML Joint Committee, "DAML Query Language (DQL) Abstract Specification", August 2002, http://www.daml.org/2002/08/dql/dql

[8] Dean, M, Schreiber, G, Bechhofer, S, van Harmelen, F, Hendler, J, Horrocks, I, McGuinness, D. L., Patel-Schneider P. F. and Stein, L. A. "OWL Web Ontology Language Reference", *W3C Candidate Recommendation* 18 August 2003 http://www.w3.org/TR/owl-ref/

[9] Decker, K, Sycara, K, and Williamson, M. "Matchmaking and Brokering." In Proceedings of *the Second International Conference on Multi-Agent Systems* (ICMAS-96), The AAAI Press, 1996

[10] Di Noia, T, Di Sciascio, E, Donini, F, and Mongiello, M. "A system for principled matchmaking in an electronic marketplace**.**" In *Proceedings of the twelfth international conference on World Wide Web*. ACM Press, 2003.

[11] Faisst, W. "Information Technology as an Enabler of Virtual Enterprises: A Life-CycleOriented Description." In *Proceedings of the European Conference on Virtual Enterprises and Networked Solutions*, Paderborn, Germany, April 1997

[12] Fikes, R., Hayes, P., and Horrocks, I. "OWL-QL - A Language for Deductive Query Answering on the Semantic Web." *Technical Report Knowledge Systems Laboratory, Stanford University, Stanford, CA*, KSL-03-14, 2003.

[13] Foundation for Intelligent Physical Agents (FIPA). "*FIPA Communicative Act Library Specification.*" www.fipa.org/specs/fipa00037/SC00037J.html

[14] Jennings, N. R, Faratin, P, Norman, T. J, O'Brien, P. and Odgers, B. "Autonomous Agents for Business Process Management" *Int. Journal of Applied Artificial Intelligence* 14 (2) 145-189, 2000.

[15] Li, L, and Horrocks, I. "E-commerce: A software framework for matchmaking based on semantic web technology." In *Proceedings of the twelfth international conference on World Wide Web*, pages 331-339. ACM Press, 2003.

[16] Lu, J, Mylopoulos, J. "XIB: eXtensible Information Broker." *International Journal on Artificial Intelligence Tools*, Vol. 11, No. 1, March 2002.

[17] Drew McDermott. "Estimated-Regression Planning for Interactions with Web Services**."** In *Proceedings of the AI Planning Systems Conference*, 2002.

[18] Martin, D. L., Cheyer, A. J, and Moran, D. B. "The Open Agent Architecture: A Framework for Building Distributed Software Systems". *Applied Artificial Intelligence*, vol. 13, no. 1-2, pp. 91-128, January-March 1999

[19] Mitra, N. "SOAP Version 1.2 Part0: Primer" *W3C Recommandation* 24 June 2003. url: www.w3c.org/TR/2003/REC-soap12-part0-20030624

[20] Motta, E, Domingue, J, Cabral, L, and Gaspari, M. "IRS-II: A Framework and Infrastructure for Semantic Web Services" *In Proc. of the International Semantic Web Conference (ISWC'03)*, Springer Verlag, Sanibel Island, Florida, USA Oct 2003.

[21] The OWL Services Coalition: *Semantic Markup for Web Services (OWL-S)*: http://www.daml.org/services/owl-s/1.0/

[22] Paolucci, M,, Takahiro Kawamura, Payne, T. R, Sycara, K.; "*Semantic Matching of Web Services Capabilities*" In *In Proc. of the International Semantic Web Conference (ISWC'02)*, Springer Verlag, Sardegna, Italy, June 2002.

[23] Paolucci, M, Sycara, K., and Kawamura, T. "Delivering Semantic Web Services." In *Proceedings of the 12Th international conference on World Wide Web*. ACM Press, 2003.

[24] Paolucci, M,, Ankolekar, A, Srinivasan, N, and Sycara, K., "The DAML-S Virtual Machine," In *Proceedings of the Second International Semantic Web Conference (ISWC)*, 2003, Sandial Island, Fl, USA, October 2003, pp 290-305.

[25] UDDI.org "*UDDI Technical White Paper*", 2000, http://www.uddi.org/whitepapers.html

[26] Wong, H. C, and Sycara, K. "A Taxonomy of Middle-agents for the Internet." In *Proceedings of the fifth International Conference on Multia-Agent Sysytesms* (ICMAS'2000), 2000