

# Trust Negotiation for Semantic Web Services

Daniel Olmedilla<sup>1</sup>, Rubén Lara<sup>2</sup>, Axel Polleres<sup>2</sup>, and Holger Lausen<sup>3</sup>

<sup>1</sup> L3S and University of Hanover, Germany

olmedilla@l3s.de

<sup>2</sup> DERI Innsbruck, Austria

{ruben.lara,axel.polleres}@deri.at

<sup>3</sup> DERI Galway, Ireland

holger.lausen@deri.ie

**Abstract.** Semantic Web Services enable the dynamic discovery of services based on a formal, explicit specification of the requester needs. The actual Web Services that will be used to satisfy the requester's goal are selected at run-time and, therefore, they are not known beforehand. As a consequence, determining whether the selected services can be trusted becomes an essential issue. In this paper, we propose the use of the Peertrust language to decide if trust can be established between the requester and the service provider. We add modelling elements to the Web Service Modeling Ontology (WSMO) in order to include trust information in the description of Semantic Web Services. In this scenario, we discuss different registry architectures and their implications for the matchmaking process. In addition, we present a matching algorithm for the trust policies introduced.

## 1 Introduction

Semantic Web Services [12] aim at providing automatic support for discovery, composition and execution of Web Services by means their explicit semantic annotation, overcoming the limitations of current Web Service technologies. One of the features of Semantic Web Services is that the functionality they provide may depend on the invocation of other services that are dynamically located and, therefore, their characteristics are not completely known at design time. In such a dynamic and open environment, where the interacting parties can be determined at run-time, trust becomes an essential issue. As Semantic Web Services provide P2P interactions between services, trust establishment mechanisms based on a simple client/server approach, in which the requester has to register and/or unconditionally disclose his (maybe private) information to the provider in order to gain access to the service [4], are not appropriate. However, some mechanisms must be in place to determine if trust between the requester and the provider can be reached. Policy languages appear as a solution to bring trust to Semantic Web Services. A policy is a rule that specifies in which conditions a resource (or another policy) might be disclosed to a requester. Related work in [8] uses a trusted matchmaker where services must register providing not only the services description but also the policies associated to that service. A user/agent includes its policy together with its request and the matchmaker filters the available services according to

the requester's functional goals together with the requester and service compatibility according to their policies. That match is performed using goals and authorization policies from the requester and the service providers. However, we believe that there are two types of policies at any entity: sensitive (and therefore the owner will not disclose them) and non-sensitive (which might be made public). A centralized matchmaker assumes that all the parties involved will disclose their policies to it. If parties do not fulfill this requirement, the matchmaker results will not be accurate. In addition, delegation becomes important when more than one entity is involved while taking a decision. For example, suppose Alice wants to buy book at Uni-Book store. Uni-Book offers a discount to any student registered at any university in the region of Lower Saxony (e.g, Hanover University). Alice is a student and she has her student id card but Uni-Book might want to verify that she did not withdraw after she registered. Therefore, Uni-Book delegates its authorization decision to Hanover Registrar (the entity in charge of student registration at Hanover University). Centralized approaches assume that services (and policies) of Hannover Registrar must be available at the registry together with Uni-Book services in order to allow delegation. Furthermore, access control is not longer a one-shot, unilateral affair found in traditional distributed systems or recent proposals for access control on the Semantic Web [6, 22]. The distributed and open nature of the Web requires trust to be established iteratively through a negotiation process where the level of trust increases in each successful iteration. This iterative process has not been taken into account in previous work on semantic web services.

In this paper, we propose an architecture based on a distributed registry and a match-making process which provides a solution to the limitations or assumptions described above. We use the PeerTrust language [4] which provides access control through Trust Negotiation to determine whether the establishment of an appropriate trust level between the requester and the provider is possible at discovery time.

Section 2 presents different possible registry architectures that influence what information is made available to the matchmaker and under which assumptions. Section 3 briefly describes trust negotiation and the Peertrust policy language. The inclusion of information disclosure policies in the modeling of Semantic Web Services is discussed in Section 4. Section 5 presents our implementation of the algorithm for the matching of trust policies. Finally, conclusions and future work are presented in Section 6.

## 2 Registry architectures

The use of Matchmakers together with service registries has been proposed in order to allow users/agents to find services that fulfill their goals [11, 17]. Service descriptions and matchmakers do not usually take into account trust policies during the process of identifying matching services. However, many useless service invocations (because e.g. access is denied to him) that do not lead to the user's expected results can be avoided by considering trust policies during the matchmaking process. In this section we will only consider issues purely related to the algorithm of determining if trust can be reached between the requester and the provider. Existing security architecture proposals for semantic web services involve the use of a matchmaker where both the requester and the service provider policies must be available [8]. While this approach has several advan-

tages, it is also built on some assumptions that should be reviewed. We believe that service providers will not disclose sensitive policies to a third entity (and loose control over them) and therefore, this would reduce the accuracy of the matchmaker, which can only make use of non-sensitive policies. Even if we assume that the matchmaker is trusted, many companies would not provide their policies (e.g, a resource protected by a policy requiring an employee id from Microsoft or IBM might suggest a secret project between both companies [23]). Consequently, we believe that delegation and negotiation will play an important role on trust and security for Semantic Web Services. In the delegation process (act of delegating a decision to another entity) two entities are involved: delegator (the entity that delegates the decision) and the delegatee (the entity that receives the delegation and takes the decision). Delegation in a centralized matchmaker might not be possible if delegatee's policies are not available in the matchmaker. In this section we describe different possible matchmaking architectures according to where client and server policies are stored (e.g. locally or 3rd party) and where the matchmaking process is done (client side, server side, trusted matchmaker) and we discuss their advantages and drawbacks.

## 2.1 Centralized matchmaking

Typically, service providers must register in a centralized registry/directory (e.g., UDDI) where they describe the properties of their services. A potential requester try to find the appropriate service by looking it up in that registry. If a service that matches its goals is found, it retrieves the complete information of that service and invokes it.

**Trusted matchmaker** The scenario presented above directly suggests to have our trust matchmaker together with the goal matchmaker at the registry and therefore both tasks might be done at the same time (as depicted in figure 1). This approach is easy to implement and the fastest (algorithm's computation is performed locally at the registry and only matching services are retrieved) but it has some disadvantages. First of all, the matchmaker must be fully trusted because requester and service providers must provide their policies (which may include confidential information) in order to find the matches for the request. This first assumption might be a problem for many users or providers who do not feel comfortable losing control of their policies. A solution is to distinguish between non-sensitive policies (e.g., a book seller might want or at least does not mind to publish that it gives a discount to students) and sensitive policies (e.g., in the policy "in order to access Bob's health record the requester must be an employee of the Psychology department of a Hospital" someone could infer that Bob has some psychological problems) and only provide non-sensitive policies to the registry. While this solution gives more flexibility to users and providers, it also reduces the accuracy of the matching algorithm. Now some private policies are missing and therefore some possible matches will not be selected (reduction of recall) and many matches will be selected although they will not be usable (reduction of precision). The second big disadvantage is related to delegation. An entity might delegate decisions to other entities (e.g, a client gets the status of preferred client if he is already a client of our company's partners). A centralized matchmaker would then need to have the policies of all the entities which

could be involved in the process (the company's partners) or to have mechanisms to retrieve automatically such information. A possible mechanism could be to expect all the company's partners to publish a service that provide access to their policies. Although this seems to be difficult, if we reached to have such a service at each delegatee entity, this service must be as well protected with some policies in order to not allow anyone to retrieve those policies. A list of allowed matchmakers might be provided or a policy language (e.g., Peertrust) could protect them. In any case, delegation might become a time consuming task and decrease the performance of the algorithm. Batch processes or caching might be some possibilities to minimize this problem.

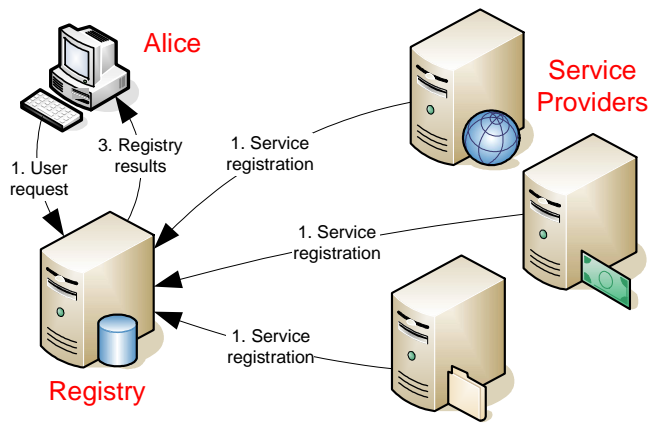


Fig. 1. Centralized Registry and Matchmaker

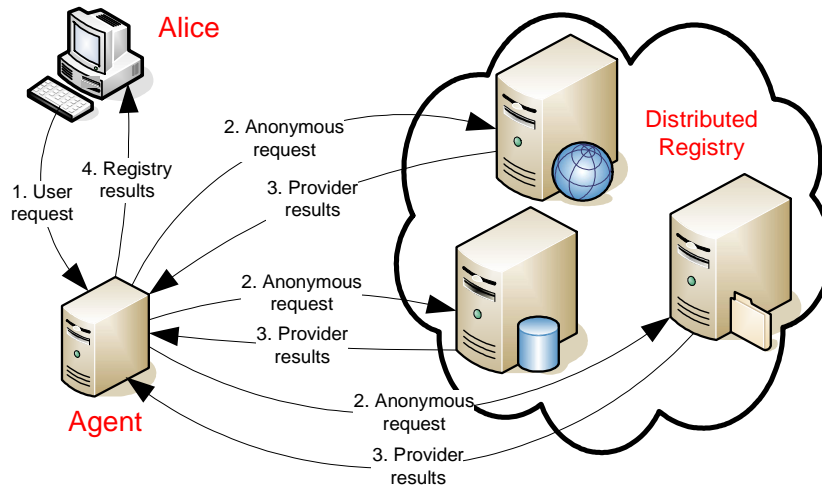
**Local client's matchmaker** A different approach is to have the matchmaker locally at the client side. It must retrieve all services information (and policies) from the registry and run the trust policies matchmaking algorithm locally. While this approach allows the use of all the client's private policies and certificates/credentials, it still has the disadvantage that providers would disclose only their public policies reducing the accuracy of the algorithm. Furthermore, this approach seems not to be scalable to registries with a high number of services. Service descriptions and policies must be retrieved from the registry to the user computer with the corresponding network overload, and this approach requires a client machine powerful enough to run the algorithm in a reasonable time.

## 2.2 Distributed matchmaking

Above we have described how a centralized and trusted registry might store the policies from users and service providers and the implications for the trust policies matchmaking process. It turned out that many disadvantages appear when relying on such an architecture. In this section we propose an alternative architecture where service providers

do not need to register at a specific registry and, most important, they do not need to provide their policies to any third entity (trusted or not).

In [21] such an architecture is described where centralized registries are replaced by a Peer-to-Peer network. Whenever a new service provider wants to offer its services, it must just join the network. This approach allows service providers not to lose control over the descriptions of their services and, in our context, not to disclose private information within their policies to other entities. We propose to follow such an architecture and allow different agent to provide users access to the service descriptions from the providers. A user might then send a query together with his policies to an agent he trusts in<sup>4</sup>. The agent sends the same query to the network. This query is distributed to the peers on the network and each peer on the network applies a matching algorithm. Whenever a peer has matches, it sends them back to the agent which joins the results and give them to the user. This architecture is depicted in Figure 2.



**Fig. 2.** Distributed Registry and Matchmaker

In this context there are several issues that must be solved. The user might not want that each provider knows about his policies. The agent plays an important role here. It is not only a mediator between the user and the distributed registry, but it is also in charge of making the query anonymous so that the providers do not know the real owner of the query they receive (making the policies anonymous too). A second problem is that we moved the matching algorithm to the service provider side. The question that then arises is how to proceed if the provider returns matches that are not real matches. Although the possibility of a provider faking the match results does exist, the provider would not

<sup>4</sup> Here it is important to note that different groups of users might use different trusted agents (e.g., the university might set up an agent for its students and professors while a company could use a different one)

benefit from that behaviour, because those false matches will lead to failed invocations. With false matches a service provider only obtains extra, unnecessary and unsuccessful invocations.

The advantages of this architecture are summarized as follows:

- *Distributed registry service.* A distributed registry service allows service providers to keep control over the description of their services as well as the policies associated to them. Additionally, we gain the nice properties of distributed environments like e.g. no single point of failure and better scalability .
- *Distributed matchmaking.* The matching algorithm might be computationally expensive. In a distributed architecture the computation time is shared by the different providers improving performance and scalability.
- *Privacy kept on service provider policies.* We believe that it is not realistic to ask service providers to disclose their (maybe very sensitive) policies to a centralized registry (even if it is trusted). In a distributed approach, servers keep those policies locally and private.

### 3 Peertrust and Trust Negotiation

In the previous sections we have mentioned policies and their importance to improve the matchmaking process. Although a policy language (REI,[7]) was already used in [8], in this paper we use the Peertrust language instead because it is especially designed to enable delegation and trust negotiation. WS-Policy [2] cannot be directly used for our purposes, as it does not describe trust policies but a general framework to describe policies for Web Services. Describing Peertrust policies in this framework will be considered in the future.

Peertrust consists on a set of guarded distributed logic programs. The Peertrust language [14] is a policy language based on Definite Horn clauses with some extensions to provide e.g. delegation. Rules are of the form

$$lit_0 \leftarrow lit_1, \dots, lit_n$$

In the remainder of this section, we concentrate on the syntactic features that are unique to the PeerTrust language and we will consider only positive authorizations.

In our previous example, Alice is a student and she wants to buy a book at Uni-Book. Uni-Book has a policy where it states which requirements any buyer must fulfill. The policy looks:

```
discount(BookTitle) $ Buyer ←  
  studentId(Buyer) @ University @ Buyer,  
  validUniversity(University),  
  studentId(Buyer) @ University.
```

Uni-Book's policies could be much more complex, but this simple policy will help us to introduce the syntax of the Peertrust language. Three conditions must be fulfilled before any buyer gets a discount at Uni-Book (represented in the body of the policy). Starting with the simplest one, the second condition checks if a university is a valid

university in order to get the discount (if it belongs to the region of Lower Saxony). The following is a list of Lower Saxony universities which are valid universities in our context.

```
validUniversity("Hanover University").
```

```
...
```

```
validUniversity("Bremen University").
```

In the third condition, *@ Issuer* represents a delegation process on Issuer. In this case, Uni-Book delegates on the university the proving of the buyer's student status (if she is still a student or if she registered and afterwards withdrew). In addition, the *Issuer* argument can be a nested term containing a sequence of issuers, which are evaluated starting at the outermost layer. In the first condition, two "@" are nested, which means that when Uni-Book receives the request from Alice, it asks her to prove that she is a student at a university and this proof must contain a digital credential signed by the university. In our case, University of Hanover issued a digital credential to Alice when she registered (like credentials in real life where the university issues a student card to registered students). As Alice already possesses this credential, she sends it to Uni-Book. If she had not had it, she should have had to send a request to "Hanover University" (which in this case would be *studentId("Alice") @ "Hanover University"*) to get such a credential.

On the head of the policy above, a symbol "\$" appears. This *\$ Requester* represents the party that sent us the query to allow parties to include the party that sent the query into the policy. The *Requester* argument can be nested, too, in which case it expresses a chain of requesters, with the most recent requester in the outermost layer of the nested term.

Summarizing, when Alice requests Uni-Book for a discount on a book, Uni-Book asks Alice to prove that she is a student in a university. Alice is a student at Hanover University so she discloses her credential (signed by University of Hanover) to Uni-Book. Uni-Book checks that the university is a valid university and sends a request to Hanover University in order to check if Alice is still a student there (she has not withdrawn after her registration). If Hanover University answers that Alice is still a student, she will get the discount.

Using the *Issuer* and *Requester* arguments, we can delegate evaluation of literals to other parties and also express interactions and the corresponding negotiation process between parties. In this paper we will not use other features of the Peertrust language like *local rules and signed rules, guards and public and private predicates*. For more details, we refer to [14] for a detailed description.

Continuing with our example, Uni-Book requires Alice to prove that she is a student at a university. However, Alice is not willing to disclose her student id card to anyone who requests it. Contrary, she will disclose her credential only to entities that are members of the "Better Business Bureau". Therefore she has the following policy:

```
student('Alice') $ Requester ←  
  member(Requester) @ 'Better Business Bureau' @ Requester.
```

Describing policies on both sides (Alice and Uni-Pro) allows a negotiation process where at each iteration trust is increased. After Uni-Book proves Alice that it belongs

to the Better Business Bureau, Alice knows enough to disclose her student card what makes the negotiation succeed and, therefore, Alice gets the discount. After several iterations (where also other entities like Hannover University might be involved) the level of trust is enough to perform the transaction.

## 4 Application to Semantic Web Services

A Web Service provider can act as requester for other services in order to provide its declared functionality. In the general case, a provider will specify subgoals that have to be accomplished in order to achieve its overall functionality. These subgoals are defined in the orchestration of the service, which in addition describes related issues such as the control flow and data flow among the subgoals. These subgoals have to be resolved at run-time, and actual Web Services that fulfill the defined subgoals have to be located. In general, the actual Web Services that will be used to provide the functionality required by the requester can be located at run-time based on a formal, explicit definition of the requester requirements. Therefore, an essential aspect to determine what services are applicable to fulfill the requester's goal is to be able to decide which candidate services can be trusted.

Consequently, trust information must be part of the description of Semantic Web Services, and this information has to be exploited during the discovery process in order to determine matching services.

We use the Web Service Modelling Ontology (WSMO)-Standard v0.3 [18] as the modelling means to describe Semantic Web Services and we situate the information disclosure policies into the appropriate modelling elements in order to exploit it during the discovery process [9]. Our main reasons to choose WSMO instead of other proposals such as OWL-S [16], IRS-II [13] or METEOR-S [20] are: 1) It allows the use of arbitrary logical expressions in the description of the service functionality, thus providing more complete descriptions than the other approaches, and 2) It uses logic programming (F-Logic [10]) to describe the logical expressions used in the description of the service, which makes possible, in the future, the alignment of the trust policies described in the Peertrust language and the functionality descriptions in WSMO.

In WSMO-Standard, goals describe the objectives that a client may have, while capabilities describe the functionality of the service. A requester describes his goal by specifying its postconditions (the state of the information space that is desired) and effects (the state of the world that is desired). Capabilities also describe postconditions of the service (the information results of its execution) and effects (the state of the world after its execution). With this information, the discovery process can match the requester's goal against the available service capabilities and determine what services provide the required functionality.

However, the service capability also needs to describe what information the Web Service requires to provide its service (preconditions) and its assumptions. Therefore, the preconditions is where the policies described in the previous sections come into play. The Web Service will state in its preconditions what information the requester has to disclose (including credentials) to gain access to the service. Credentials can be described using the ontology described in [3] and included in the preconditions of

the service. In addition to enumerate all the information items that the service requires for its execution, we add the Peertrust expressions that describe the exact policies the service employs. Since in WSMO-Standard preconditions are described via axiom definitions, we need to extend this description to add the relevant policies. Figure 3 depicts our modelling in F-Logic [10]. A precondition includes any number of axiom definitions and any number of policies (encoded as strings that represent a Peertrust formula). Notice that we use F-Logic, as it is the language used in WSMO-Standard to describe preconditions.

```
precondition [
  axiom ==> axiomDefinition,
  policies ==> string
]
```

**Fig. 3.** Definition of precondition

By modelling preconditions in this way in the service capability, we capture both the information that the service requires from the requester and what policies apply to gain access to the service. Therefore, we are modelling not only the functional description of the service in terms of preconditions, assumptions, postconditions and effects but also what are the policies that describe what information the requester must disclose in order to be trusted by the provider. As discussed in Section 2, the provider might not want to make these policies available. For this reason, we propose to follow the distributed architecture described in Section 2.2, where the policies are kept private at the provider side. In addition, we have to model at requester's side the information disclosure policies of such requester i.e. what information he is willing to disclose and under what conditions.

The description of the requester's information disclosure policies is modelled in F-Logic in figure 4. A policy contains a set of information items, for which the actual data (an ontology instance) to be disclosed and the disclosure policy (a Peertrust formula) are specified. Information items that are unconditionally disclosed will have an empty Peertrust formula. Notice that the data disclosed can be an instance of any ontology concept, including a credential.

```
infoDisclosurePolicy [
  infoItems ==> infoItem [
    data => ontologyInstance,
    peerTrustExpression => string
  ]
]
```

**Fig. 4.** Definition of requester's information disclosure policies

Having such information disclosure policies described at the requester side and the preconditions at the provider side, all the declarative elements needed to determine if the trust establishment is possible are in place. We know the conditions the requester must fulfill to be trusted by the provider (described using Peertrust in the preconditions definition), and what information the requester will disclose and under what conditions. Using the respective Peertrust policies and the matching algorithm described in section 5 we can determine if trust can be reached.

It is important to notice that what we determine is whether trust can be reached between the requester and the provider based on published policies. The actual interchange of messages (credentials) to really establish trust at invocation time, and the modelling of the service choreography [19] for that interchange is out of the scope of this paper.

After modelling the elements above, a relevant issue is determining how the matchmaker can access the information described by the requester and the provider. In the distributed architecture proposed in section 2.2, the description of the services is kept on the provider, and the (anonymous) request is sent by the user agent to the peers. In this approach, the matchmaking process is performed at every provider and the results returned to the user agent. Therefore, the provider policies will be available to the matchmaker, as the matchmaking process will take place on the provider's side. However, not only the requester's goal but also its information disclosure policies or the subset relevant for this goal i.e. the information that the requester is willing to disclose (under certain conditions) to achieve the goal has to be submitted to the peers by the user agent, as these policies are necessary to determine whether trust can be reached between the parties.

An obvious drawback of this approach is that the requester might be willing to disclose a big set of information that is not sensitive, and submitting this information to all the peers creates an information overload. Therefore, we propose an alternative solution in which the matchmaker requests to the user agent that submitted the query the information it needs to satisfy the service requirements. The user agent will have access to the requester information disclosure policies, and will send back to the matchmaker the relevant information together with the relevant (anonymous) policies. To do so, the user agent will expose a Web Service that receives the requests from the matchmakers and send back the appropriate policies. This service will only be accessible to providers that are part of the P2P network of providers, that are assumed to be trusted.

## 5 Algorithm implementation

In this section we present our implementation of an algorithm that performs the matching of trust policies. Each of the service providers has this algorithm and it runs it locally whenever a new request (query hereafter) from an agent arrives.

We limit ourselves to the evaluation of the policies described in previous sections i.e. the matching of the Peertrust policies described for the requester and the provider. For more details about matching services with requests we refer the reader to [9] and [11].

Guarded distributed logic programs can be evaluated in many different ways. This flexibility is important, as different registries, and even different service providers

within the same registry, may prefer different approaches to evaluation. As we provide explicit delegation in our policies the service provider might include other entities (peers) to delegate decisions. We will present a simple evaluation algorithm for PeerTrust that is based on the cooperative distributed computation of a proof tree, with all peers employing the same evaluation algorithm. The algorithm assumes that each peer uses a queue to keep track of all active proof trees and the expandable subqueries in each of these trees. The proof trees contain all information needed, including used rules, credentials and variable instantiations. Peers communicate with one another by sending queries and query answers to each other.

The following sketch of the algorithm uses EITHER:/OR: to express a non-deterministic choice between several possible branches of the algorithm.

```

Let TreeList denote the structure with all active proof trees
Set TreeList := []
Let Tree denote the structure holding Query$Requester and Proof
    both of which may still contain uninstantiated variables
Loop
  EITHER:
    Receive Tree: a query to answer / a goal to prove
    Add_New_Tree(Tree, TreeList)
  OR:
    Receive Answer(Tree)
    Add_Answers(Answer(Tree), TreeList)
  OR:
    Receive Failure(Tree) from peer
    Send Failure(Tree) to Requester
    Remove_Tree(Failure(Tree), TreeList)
  OR:
    Process_a_Tree(TreeList)
end Loop

```

At each step a peer can receive a new query from another peer, receive answers, learn that there are no answers for a query it previously sent to a peer, or selects one of its active trees for processing . If this tree is already complete, the answers can be returned to the peers who requested this evaluation. If the tree contains subqueries which still have to be evaluated, the peer selects one of them and tries to evaluate it.

```

Process_a_Tree(TreeList)
Let NewTrees denote the new proof trees
Set NewTrees := []
Select_Tree(Tree, TreeList, RestOfTreeList)
IF all subqueries in Tree are already evaluated
THEN
  Send (Answer(Tree)) to Requester
  TreeList := RestTreeList
ELSE
  Select_Subquery (SubQuery, Tree)
  IF SubQuery can be evaluated locally
  THEN

```

```

    Loop while new local rules are found
      Expand SubQuery into its subgoals
      Update_Tree(Tree,NewSubgoals)
      Add_Tree(Tree,NewTrees)
    End loop
  ELSE //if it is a goal with an "@ Issuer" suffix,
    // indicating remote evaluation
    IF peer has Signed_Rule(SubQuery)
      Loop while new signed rules are found
        Expand SubQuery into its subgoals
        Update_Tree(Tree,NewSubgoals)
        Add_Tree(Tree,NewTrees)
      End loop
    ELSE
      Send Request(SubQuery) to Issuer
      Update_Status(Tree, waiting)
    END IF
  END IF
  IF no local or remote expansion for SubQuery was possible
    Send (Failure(Tree)) to Requester
  ELSE
    Add_New_Trees
      (NewTrees,RestTreeList,NewTreeList)
  END IF
  TreeList := NewTreeList
END IF

```

Expansion of subqueries is done either locally (using the peer's rules and signed rules) or by sending the subquery to a remote peer (in case of delegation). Many queries per proof can be active (i.e., awaiting answers and being processed) at any time. Each new query from a remote peer starts a new proof tree while answers from remote peers are "plugged into" existing proof trees. An example of a query expansion in a proof tree is depicted in figure 5, where a tree is expanded into two and then three trees. Each tree structure contains at least root and leaves, plus any additional information from the proof, including credentials, that we want to keep and/or return to the requester. If one proof tree for the original query is completed, then the negotiation is over and the requester obtains access to the desired resource.

This algorithm can be extended and improved in many different ways. For example, it can be made more efficient and effective at run time by generalizing the definition of a query, allowing iteration through a set of query answers, allowing intensional query answers, support for caching of query answers, and prioritization of rules a la [5]. Alternatively, the algorithm can be revamped in ways that will allow different peers to choose different evaluation algorithms, along the lines of [23], or to provide provable guarantees of completeness and termination, as offered by the algorithms of [23]. No matter what revisions are made, however, at its heart any evaluation algorithm will be working to construct a certified proof tree.

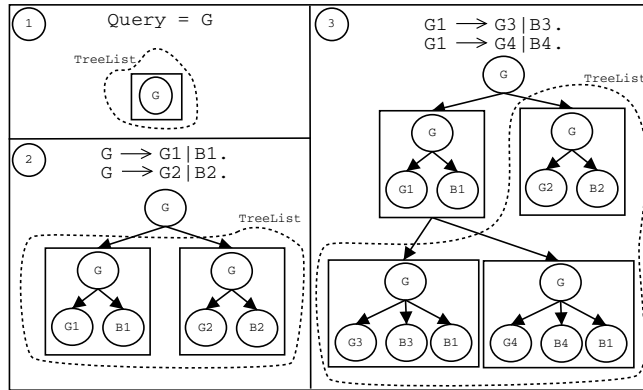


Fig. 5. Resource access control scenario

## 6 Conclusions and future work

Semantic Web Services bring dynamism to current Web Service technologies, and the actual services that will be employed to satisfy a goal are determined at run time. Therefore, requesters interact with services that they do not know beforehand, and they have to determine if they can trust such services. In this context, modelling trust information in Semantic Web Services becomes necessary. In this paper, we include trust policies in WSMO-Standard, together with the information disclosure policies of the requester, using the Peertrust language. Peertrust provides the means to perform trust negotiation and delegation. As the matchmaker needs to have access to the requester and provider policies, in order to match not only the requester functional requirements but also trust information, the architecture of the registry and matchmaker becomes a relevant issue. We have proposed a distributed registry and matchmaker architecture that allows the service providers to keep their policies private, thus not forcing them to disclose sensitive information. It also improves the efficiency and scalability of the solution. We have also implemented an algorithm that matches the requester and provider Peertrust policies to determine if trust between them can be established. Future work includes the integration of this algorithm with the functional matching algorithm (matching of the requester goal and the service capability) in our P2P network (Edutella, [15]). We are also studying the possibility of extending our work to Web Services on Grid environments. Some previous work on using Peertrust on Grid environments can be found in [1]. At this point, we use strings to model the Peertrust formulas in the description of the service and in the description of the requester information disclosure policies. However, Peertrust expressions can be modelled directly as F-Logic formulas, extending their semantics to include Peertrust features such as delegation. As part of the integration of the functional matching and the trust matching algorithms, we plan to model Peertrust expressions as F-Logic formulas. We will also investigate other possibilities to better integrate policies in WSMO. Finally, we plan to refine the approach we propose to give the provider access to use the requester's information disclosure policies, in which we assumed the providers requesting access to these policies is trusted.

**Acknowledgments** This research is partially funded by the projects ELENA (<http://www.elena-project.org>, IST-2001-37264), REVERSE (<http://reverse.net>, IST-506779), Knowledge Web (<http://knowledgeweb.semanticweb.org/>, FP6-507482), DIP (<http://dip.semanticweb.org/>, FP6-507683) and SWWS (<http://swws.semanticweb.org/>, IST-2001-37134).

## References

1. J. Basney, W. Nejdl, D. Olmedilla, V. Welch, and M. Winslett. Negotiating trust on the grid. In *Proc. of 2nd Workshop on Semantics in P2P and Grid Computing*, New York, 2004, May 2004.
2. D. Box, F. Curbera, M. Hondo, C. Kaler, D. Langworthy, A. Nadalin, N. Nagarathnam, M. Nottingham, C. von Riegen, and J. Shewchuk. Web services policy framework (ws-policy). <http://www-106.ibm.com/developerworks/library/ws-polfram/>, May 2003.
3. G. Denker, L. Kagal, T. Finin, M. Paolucci, and K. Sycara. Security for daml web services: Annotation and matchmaking. In *Proceedings of the 2nd International Semantic Web Conference*, Sanibel Island, Florida, USA, Oct. 2003.
4. R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett. No registration needed: How to use declarative policies and negotiation to access sensitive resources on the semantic web. In *Proc. of the 1st European Semantic Web Symposium*, Heraklion, Greece, May 2004.
5. B. Grosz. Representing e-business rules for the semantic web: Situated courteous logic programs in RuleML. In *Proceedings of the Workshop on Information Technologies and Systems (WITS)*, New Orleans, LA, USA, Dec. 2001.
6. L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *Proceedings of the 2nd International Semantic Web Conference*, Sanibel Island, Florida, USA, Oct. 2003.
7. L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, Lake Como, Italy, June 2003.
8. L. Kagal, M. Paoucci, N. Srinivasan, G. Denker, T. Finin, and K. Sycara. Authorization and privacy for semantic web services. In *AAAI 2004 Spring Symposium on Semantic Web Services*, Stanford University, Mar. 2004.
9. U. Keller, R. Lara, A. Polleres, and H. Lausen. Inferencing support for semantic web services: Proof obligations. <http://www.wsmo.org/2004/d5/d5.1/v0.1/>, Apr. 2004. WSML working draft.
10. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.
11. L. Li and I. Horrocks. A software framework for matchmaking based on semantic web technology. In *Proceedings of the 12th International Conference on the World Wide Web*, Budapest, Hungary, May 2003.
12. S. McIlraith, T. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems, Special Issue on the Semantic Web*, 16(2):46/53, March/April 2001.
13. E. Motta, J. Domingue, L. Cabral, and M. Gaspari. Irs-ii: A framework and infrastructure for semantic web services. In *2nd International Semantic Web Conference (ISWC2003)*. Springer Verlag, October 2003.
14. W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: automated trust negotiation for peers on the semantic web. Technical Report, Oct. 2003.
15. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. Edutella: A P2P networking infrastructure based on RDF. In *Proceedings of the 11th International World Wide Web Conference (WWW2002)*, Hawaii, USA, June 2002.

16. OWL-S services coalition. OWL-S: semantic markup for web services. <http://www.daml.org/services/owl-s/1.0/owl-s.pdf>, November 2003.
17. M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In I. Horrocks and J. Handler, editors, *1st Int. Semantic Web Conference (ISWC)*, pages 333–347. Springer Verlag, 2002.
18. D. Roman, H. Lausen, and U. Keller. Web service modeling ontology - standard. <http://www.wsmo.org/2004/d2/v0.3/>, Mar. 2004. WSMO working draft.
19. D. Roman, L. Vasiliu, C. Bussler, and M. Stollberg. Choreography in wsmo. <http://www.wsmo.org/2004/d14/v0.1/>, Apr. 2004. WSMO working draft.
20. K. Sivashanmugam, K. Verma, A. Sheth, and J. Miller. Adding semantics to web services standards. In *1st International Conference on Web Services (ICWS'03)*, pages 395–401, June 2003.
21. U. Thaden, W. Siberski, and W. Nejdl. A semantic web based peer-to-peer service registry network. Technical report, Learning Lab Lower Saxony, 2003.
22. G. Tonti, J. M. Bradshaw, R. Jeffers, R. Montanari, N. Suri, and A. Uszok. Semantic web languages for policy representation and reasoning: A comparison of KAoS, Rei and Ponder. In *Proceedings of the 2nd International Semantic Web Conference*, Sanibel Island, Florida, USA, Oct. 2003.
23. T. Yu, M. Winslett, and K. Seamons. Supporting Structured Credentials and Sensitive Policies through Interoperable Strategies in Automated Trust Negotiation. *ACM Transactions on Information and System Security*, 6(1), Feb. 2003.