No Registration Needed:
How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web

Rita Gavriloaie[1], Wolfgang Nejdl[1], Daniel Olmedilla[1], Kent E. Seamons[2], and Marianne Winslett[3]

[1] L3S and University of Hannover, Germany
{gavriloaie,nejdl,olmedilla}@learninglab.de
[2] Department of Computer Science, Brigham Young University, USA
seamons@cs.byu.edu
[3] Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA
winslett@cs.uiuc.edu

Abstract. Gaining access to sensitive resources on the Web usually involves an explicit registration step, where the client has to provide a predetermined set of information to the server. The registration process yields a login/password combination, a cookie, or something similar that can be used to access the sensitive resources. In this paper we show how an explicit registration step can be avoided on the Semantic Web by using appropriate semantic annotations, rule-oriented access control policies, and automated trust negotiation. After presenting the PeerTrust language for policies and trust negotiation, we describe our implementation of implicit registration and authentication that runs under the Java-based MINERVA Prolog engine. The implementation includes a PeerTrust policy applet and evaluator, facilities to import local metadata, policies and credentials, and secure communication channels between all parties.

1 Introduction

In traditional distributed environments, service providers and requesters are usually known to each other. Often shared information in the environment tells which parties can provide what kind of services and which parties are entitled to make use of those services. Thus, trust between parties is a straightforward matter. Even if on some occasions there is a trust issue, as in traditional client-server systems, the question is whether the server should trust the client, and not vice versa. In this case, trust establishment is often handled by unidirectional access control methods, most typically by having the client log in as a preregistered user. This approach has several disadvantages.

1. Such practices are usually offered in a "take-it-or-leave-it" fashion, i.e., the clients either *unconditionally* disclose their information to the server in the preregistration phase, or cannot access the service at all. The clients cannot apply their own access control policies for their personal information, and decide accordingly whether the server is trustworthy enough so that sensitive information can be disclosed.

2. When the clients fill out the online registration form, it is very hard for the server to verify whether the provided information is valid.

3. Often, much of the information required in online registration forms does not seem directly related to the service that clients want to access. However, since the clients have to unconditionally provide the information, there is no way that they can discuss their misgivings with the server.

A notable example of the latter shortcoming is that traditional access control methods usually require the customer to reveal her exact identity, which, in many cases, is irrelevant to the service the customer requests. For example, suppose that Alice visits the current ACM on-line store. To find out the price of a one-year membership in ACM SIGMOD, Alice must go through a lengthy registration process to obtain an "ACM Web Account" and use that account for all her interactions with the ACM Store. Even if Alice decides to join ACM SIGMOD, she should not have to provide all of the personal information that is requested on the ACM Web Account registration form, such as her fax number. These kinds of registration requests are sufficiently annoying and intrusive that services have sprung up to automatically fill in such forms with false information and keep track of the resulting account names and passwords (see, e.g., http://www.sixcube.com/autofiller or http://www.roboform.com).

Similar access control needs have arisen in the context of the EU/IST ELENA project [13], whose participants include e-learning and e-training companies, learning technology providers, and universities and research institutes (http://www.elena-project.org/). Any potential student or learning provider should be able to connect to the ELENA network and access an appropriate subset of its resources. For example, suppose that E-Learn Associates manages a 1-credit-unit Spanish course in the ELENA network, and Alice wishes to take the course. If the course is accessible free of charge to all police officers who live in and work for the state of California, Alice can show E-Learn her digital police badge to prove that she is a state police officer, as well as her California driver's license, to gain access to the course at no charge.

However, Alice may not feel comfortable showing her police badge to just anyone; she knows that there are Web sites on the west coast that publish the names, home addresses, and home phone numbers of police officers. Because of that, Alice may only be willing to show her badge to companies that belong to the Better Business Bureau OnLine. But with this additional policy, access control is no longer the one-shot, unilateral affair found in traditional distributed systems or recent proposals for access control on the Semantic Web [9, 14]: to see an appropriate subset of Alice's digital credentials, E-Learn will have to show that it satisfies the access control policies for each of them; and in the process of demonstrating that it satisfies those policies, it may have to disclose additional credentials of its own, but only after Alice demonstrates that she satisfies the access control policies for each of *them*; and so on.

In this paper, we build upon the previous work on policy-based access control for the Semantic Web by showing how to use *automated trust negotiation* in the PeerTrust approach to access control. We start by introducing the concepts behind trust negotiation in section 2. We then introduce guarded distributed logic programs to express and implement trust negotiation in a distributed environment, in section 3. Section 4 describes the PeerTrust execution environment and an application scenario implemented

for this paper, and section 5 provides additional information about the PeerTrust implementation. The PeerTrust implementation uses the Java-based MINERVA Prolog engine for reasoning and uses Java applications and applets for everything else, including negotiation capabilities both on the server and the client side, means for secure communication, and facilities to import local RDF metadata, policies, and credentials. The paper concludes with a short discussion of related approaches and further work.

2   Trust Negotiation

Digital credentials (or simply *credentials*) make it feasible to manage trust establishment efficiently and bidirectionally on the Internet. Digital credentials are the on-line counterparts of paper credentials that people use in their daily life, such as a driver's license. By showing appropriate credentials to each other, a service requester and provider can both prove their qualifications. In detail, a credential is a digitally signed assertion by the *credential issuer* about the properties of one or more entities. The credential issuer uses its private key to sign the credential, which describes one or more attributes and/or relationships of the entities. The public key of the issuer can be used to verify that the credential was actually issued by the issuer, making the credential verifiable and unforgeable. The signed credential can also include the public keys of the entities referred to in the credential. This allows an entity to use her private key to prove that she is one of the entities referred to in the credential, by signing a challenge [12]. Digital credentials can be implemented in many ways, including X.509 [7] certificates, anonymous credentials [1, 2], and signed XML statements [3].

When credentials do not contain sensitive information, the trust establishment procedure is very simple. For example, if Alice will show her police badge and driver's license to anyone, then she (more precisely, a software agent acting on her behalf) first checks E-Learn's policy for its Spanish course, which is always available. Then she presents her police badge and license along with the retrieval request. However, in many situations, especially in the context of e-business, credentials themselves contain sensitive information. For example, as discussed earlier, Alice may only be willing to show her police badge to members of the BBB. To deal with such scenarios, a more complex procedure needs to be adopted to establish trust through negotiation.

In the PeerTrust approach to automated trust establishment, trust is established gradually by disclosing credentials and requests for credentials, an iterative process known as *trust negotiation*. This differs from traditional identity-based access control:

1. Trust between two strangers is established based on parties' properties, which are proven through disclosure of digital credentials.
2. Every party can define access control policies to control outsiders' access to their sensitive resources. These resources can include services accessible over the Internet, documents and other data, roles in role-based access control systems, credentials, policies, and capabilities in capability-based systems.
3. In the approaches to trust negotiation developed so far, two parties establish trust directly without involving trusted third parties, other than credential issuers. Since both parties have access control policies, trust negotiation is appropriate for deployment in a peer-to-peer architecture, where a client and server are treated equally.

Instead of a one-shot authorization and authentication, trust is established incrementally through a sequence of bilateral credential disclosures.

A trust negotiation is triggered when one party requests to access a resource owned by another party. The goal of a trust negotiation is to find a sequence of credentials $(C_1, \ldots, C_k, R)$, where R is the resource to which access was originally requested, such that when credential C_i is disclosed, its access control policy has been satisfied by credentials disclosed earlier in the sequence—or to determine that no such credential disclosure sequence exists. For example, when Alice registers for the free Spanish course:

Step 1 Alice requests to access E-Learn's Spanish course at no charge.

Step 2 E-Learn asks Alice to show a police badge issued by the California State Police to prove that she is a police officer, and her driver's license to prove that she is living in the state of California.

Step 3 Alice is willing to disclose her driver's license to anyone, so she sends it to E-Learn. However, she considers her police badge to contain sensitive information. She tells E-Learn that in order to see her police badge, E-Learn must prove that it belongs to the Better Business Bureau.

Step 4 Fortunately, E-Learn does have a Better Business Bureau membership card. The card contains no sensitive information, so E-Learn discloses it to Alice.

Step 5 Alice now believes that she can trust E-Learn and discloses her police badge to E-Learn.

Step 6 After checking the badge and driver's license for validity (i.e., that each credential's contents has not been altered since it was signed, and that it was signed by the right party) and that Alice owns them, E-Learn gives Alice the free registration for this course.

We can view each resource in this example as an item on the Semantic Web, with its salient properties stored as RDF properties. For example, the amount of credit associated with the course can be represented by the RDF property (written as a binary predicate) "creditUnits(spanish101, 1)". Each resource is automatically associated with the appropriate access control policies by means of these properties, as discussed later.

3 PeerTrust Guarded Distributed Logic Programs

In this section, we describe the syntax of the PeerTrust language. The semantics of the language is an extension of that of SD3 [15] to allow the set of all PeerTrust programs to be viewed as a single global logic program. We refer the interested reader to [15, 11] for semantic details.

Definite Horn clauses are the basis for logic programs [10], which have been used as the basis for the rule layer of the Semantic Web and specified in the RuleML effort ([4, 5]). PeerTrust's language for expressing access control policies is also based on definite Horn clauses, i.e., rules of the form

$$lit_0 \leftarrow lit_1, \ldots, lit_n$$

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

References to Other Peers The ability to reason about statements made by other parties is central to trust negotiation. For example, in section 2, E-Learn wants to see a statement from Alice's employer that says that she is a police officer. One can think of this as a case of E-Learn *delegating evaluation* of the query "Is Alice a police officer?" to the California State Police (CSP). Once CSP receives the query, the manner in which CSP handles it may depend on who asked the query. Thus CSP needs a way to specify which party made each request that it receives. To express delegation of evaluation to another party, we extend each literal lit_i with an additional *Issuer* argument,

$$lit_i \ @ \ \text{Issuer}$$

where *Issuer* specifies the party who is responsible for evaluating lit_i or has the authority to evaluate lit_i. For example, E-Learn's policy for free courses might mention policeOfficer(X) @ csp. If that literal evaluates to true, this means that CSP states that Alice is a California police officer. For example, if all California state police officers can enroll in Spanish 101 at no charge, we have:

eLearn:
freeEnroll(spanish101, X) ← policeOfficer(X) @ csp.

This policy says that the CSP is responsible for certifying the employment status of a given person. For clarity, we prefix each rule by the party in whose knowledge base it is included.

The *Issuer* argument can be a nested term containing a sequence of issuers, which are evaluated starting at the outermost layer. For example, CSP is unlikely to be willing to answer E-Learn's query about whether Alice is a police officer. A more practical approach is to ask Alice to evaluate the query herself, i.e., to disclose her police badge:

eLearn:
policeOfficer(X) @ csp ←
 policeOfficer(X) @ csp @ X.

CSP can refer to the party who asked a particular query by including a *Requester* argument in literals, so that we now have literals of the form

$$lit_i \ @ \ \text{Issuer} \ \$ \ \text{Requester}$$

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. 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. For instance, extending and generalizing the previous example, consider E-Learn Associates' policy for free Spanish courses for California police officers:

eLearn:
freeEnroll(Course, Requester) $ Requester ←
 policeOfficer(Requester) @ csp @ Requester,
 rdfType(Course, "http:.../elena#Course"),
 dcLanguage(Course, "es"),
 creditUnits(Course, X),
 X < = 1.

If Alice provides appropriate identification, then the policy for the free enrollment service is satisfied, and E-Learn will allow her to access the service through a mechanism not shown here. In this example, the mechanism can transfer control directly to the enrollment service. For some services, the mechanism may instead give Alice a non-transferable token that she can use to access the service repeatedly without having to negotiate trust again until the token expires. The mechanism can also implement other security-related measures, such as creating an audit trail for the enrollment. When the policy for a negotiation-related resource such as a credential becomes satisfied, the run-time system may choose to include the resource directly in a message sent during the negotiation, as discussed later.

Local Rules and Signed Rules Each party defines the set of access control policies that apply to its resources, in the form of a set of definite Horn clause rules that may refer to the RDF properties of those resources. These and any other rules that the party defines on its own are its *local* rules. A party may also have copies of rules defined by other parties, and it may use these rules in its proofs in certain situations. For example, Alice can use a rule (with an empty body in this case) that was defined by CSP to prove that she is really a police officer:

alice:
policeOfficer(alice) @ csp
    signedBy [csp].

In this example, the "signedBy" term indicates that the rule has CSP's digital signature on it. This is very important, as E-Learn is not going to take Alice's word that she is a police officer; she must present a statement signed by the police to convince E-Learn. A signed rule has an additional argument that says who issued the rule. The cryptographic signature itself is not included in the logic program, because signatures are very large and are not needed by this part of the negotiation software. The signature is used to verify that the issuer really did issue the rule. We assume that when a party receives a signed rule from another party, the signature is verified before the rule is passed to the GDLP evaluation engine. Similarly, when one party sends a signed rule to another party, the actual signed rule must be sent, and not just the logic programmatic representation of the signed rule.

More complex signed rules often represent delegations of authority. For example, the California Highway Patrol (CHP) can use a signed rule to prove that it is entitled to answer queries about California state police officers, using its database of CHP officers:

chp:
policeOfficer(X) @ csp ←
    signedBy [csp]
 policeOfficer(X) @ chp.

If Alice's police badge is signed by the CHP, then she should cache a copy of the rule given above and submit both the rule and the police badge when E-Learn asks her to prove that she is a California state police officer.

Guards To guarantee that all relevant policies are satisfied before access is given to a resource, we must sometimes specify a partial evaluation order for the literals in the body of a rule. Similar to approaches to parallel logic programming such as Guarded Horn Logic [16], we split the body's literals into a sequence of sets, divided by the symbol "|". All but the last set are *guards*, and all the literals in one set must evaluate to true before any literals in the next set are evaluated. In particular, (a subset of) the guards in a policy rule can be viewed as a query that describes the set of resources for which this policy is applicable. The query is expressed in terms of the RDF characteristics of the resources to which the policy applies. In this paper, access will be granted to a resource if a user can satisfy any one of the policy rules applicable to that resource. (This scheme can be extended to allow negative authorizations and conflict resolution.) For example, if Alice will show her police membership credentials only to members of the Better Business Bureau, she can express that policy as a guard that is inside her police credential but outside the scope of CSP's signature. This is expressed syntactically by adding these additional local guards between ← and "signedBy".

alice:
policeOfficer(alice) @ csp
  ← member(Requester) @ bbb @ Requester
    | signedBy [csp].

Both local and signed rules can include guards. PeerTrust guards express the evaluation precedence of literals within a rule, and any rule whose guard literals evaluate to true can be used for continued evaluation. In contrast, guards in parallel logic programming systems usually introduce a deterministic choice of one single rule to evaluate next.

Public and Private Predicates In trust negotiation, we must be able to distinguish between predicates that can be queried by external parties, and ones that cannot—analogous to the public and private procedures in object-oriented languages. For example, authentication and time-of-day predicates are private. Parties may also have private rules, which are neither shown to nor can directly be called by other parties. Public and private rules and predicates are straightforward to design and implement in definite Horn clauses. The examples in this paper include only public rules.

4 Execution Environment and High-Level Algorithms

PeerTrust's current implementation is targeted at scenarios that arise in the ELENA project, where clients want to access protected Web resources residing in a distributed repository (see Fig. 1). In Fig. 1, Alice and E-Learn obtain trust negotiation software signed by a source that they trust (PeerTrust Inc.) and distributed by PeerTrust Inc. or another site, either as a Java application or an applet. After Alice requests the Spanish course from E-Learn's web front end, she enters into a trust negotiation with E-Learn's negotiation server. The negotiation servers may also act as servers for the major resources they protect (the Learning Management Servers (LMS) in Fig. 1), or may be separate entities, as in our figure. Additional parties can participate in the negotiation,
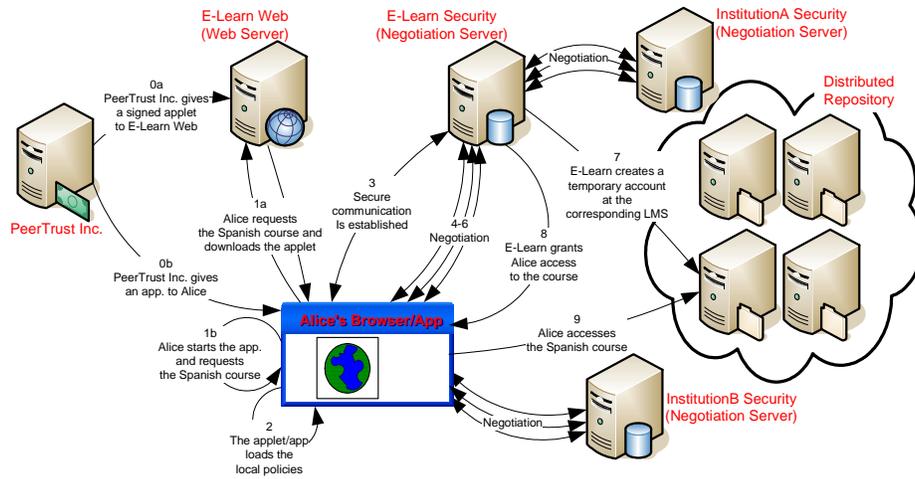
Fig. 1. Resource access control scenario

if necessary, symbolized in our figure by the InstitutionA and InstitutionB servers. If access to the course is granted, E-Learn sets up a temporary account for Alice at the course provider's site, and redirects her original request there. The temporary account is invisible to Alice.

Parties possess credentials that they have either obtained directly from credential issuers or from on-line repositories of issued credentials. We expect that credential issuers will also give parties default, tailorable policies to protect the credentials they have issued. For example, when Alice receives a new credit card, it is to the advantage of the credit card company to help Alice guard against theft or misuse of her card, by giving her a bootstrap policy to protect the card. Issuers can also direct novice users of digital credentials to trustworthy organizations that provide trust negotiation software, so that the users can actually make use of their new credentials. This software should be signed by the organization that produces it, so that users can be sure that the software that they download and run has not been tampered with, and is actually the software produced by the organization. In the PeerTrust 1.0 demo, a trusted party offers trust negotiation software as a signed Java application or applet that any other party can download and use. Other parties offer credentials and policies that can be obtained at run time and (for demo purposes only) assign public and private keys to users new to the demo.

When a client asks to access a resource, the negotiation server must determine what policies are applicable to that resource. As in [9], we do not want to assign policies explicitly to each resource, due to scalability and manageability concerns. Instead, in the context of the Semantic Web, we implicitly assign policies to each resource based on the RDF metadata associated with that resource. These metadata are imported into the negotiation server as facts in the logic program. Such facts are mentioned in the bodies of each policy rule, to define the scope of the policy; rule bodies may include additional limitations such as time-of-day restrictions to further restrict the scope of the

policy. If desired, the rule bodies can be precompiled or indexed so that the policies relevant to a particular resource can be found quickly.

In our example, E-Learn might use the following course metadata:

```
<rdf:Description about="http://www.learninglab.de/.../spanishCourse/">
<rdf:type resource="http://www.learninglab.de/.../elena#Course"/>
<dc:language>es</dc:language>
<dc:title>Curso de español</dc:title>
<dc:description>
Dirigido a estudiantes sin conocimiento previo del lenguaje. Se estudiarán
las estructuras gramaticales básicas y vocabulario de uso común.
</dc:description>
<elena:creditUnits>1</elena:creditUnits>
<dc:creator>Daniel Olmedilla</dc:creator>
<dcq:created>
<dcq:W3CDTF><rdf:value>2003-12-15</rdf:value></dcq:W3CDTF>
</dcq:created>
```

In the following policy, E-Learn specifies that the E-Learn employee who is the author of a resource can modify that resource.

eLearn:
modify(Course, Requester) $ Requester ←
  employee(Requester) @ eLearn @ Requester,
  dcCreator(Course, Requester).

The RDF description of the resource may match the bodies of several policy rules, automatically making several policies applicable to the resource. As mentioned earlier, for an access control model based on positive authorizations, the client will be given access if s/he can satisfy any of the relevant policies. If no policies are relevant, then the client does not gain access. If a negotiation succeeds in showing that a client is entitled to access the requested resource, then the negotiation server redirects the client's browser to the correct location for the desired service, as described later.

Because PeerTrust represents policies and credentials as guarded distributed logic programs, the trust negotiation process consists of evaluating an initial logic programming query ("Can this user access this resource?") over a physically distributed logic program (a set of credentials and policies). Because the program is distributed, the help of several parties may be needed to answer a query. From the perspective of a single party, the evaluation process consists of evaluating the query locally to the maximum extent possible, and sending requests to other parties for help in evaluating the aspects of the query that cannot be resolved locally. The "@ issuer" arguments in the logic program tell each party who to ask for help. (When an issuer argument is uninstantiated, one may need to ask a broker for help in finding the relevant parties.)

As with any logic program, a PeerTrust query can be evaluated in many different ways. This flexibility is important, as different parties may prefer different approaches to evaluation. For PeerTrust 1.0, we chose to adopt a *cooperative* model of evaluation. In other words, PeerTrust 1.0 assumes that all parties are using identical trust negotiation software. The constructs used in the first-generation implementation will form the substrate of PeerTrust 2.0, which will adopt an *adversarial* paradigm that gives parties more autonomy in their choice of trust negotiation software and provides additional

guarantees of security against attacks. PeerTrust 2.0 will also address the resolution of deadlocks and livelocks, which are not considered in PeerTrust 1.0.

The evaluation algorithm for PeerTrust 1.0 is based on the cooperative distributed computation of a proof tree that, if fully instantiated, will prove that the client is entitled to access the desired resource. Each party 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 to show that a client is entitled to access a resource, including the policy rules that have been used, the instantiations of variables in those rules, and the credentials themselves. When a party needs help from another party in expanding a proof tree, it sends a traditional logic programming query to the other party. The response to that query includes both the traditional logic programming query answers, as well as proof trees showing that each of the answers to the query really is an answer to the query. (We will refer to this collection of information simply as the "query answer" in this paper.) To provide privacy during negotiation and avoid certain kinds of attacks, queries and query answers must be sent over secure communication channels.
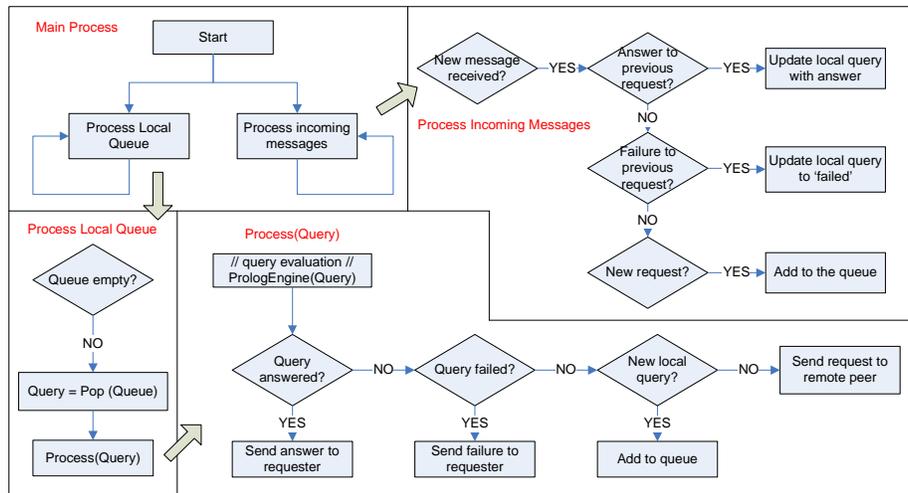

Fig. 2. PeerTrust 1.0 Evaluation Algorithm

The evaluation algorithm used in PeerTrust 1.0 is described in [11]. Briefly, the evaluation algorithm evaluates a logic program using the usual Prolog evaluation strategy, depth-first and left-to-right. PeerTrust 2.0 will include more elaborate evaluation strategies, and, using those, will also allow the different parties to choose from different negotiation strategies as discussed in Yu et al. [18].

Figure 2 shows the high-level modules in the current implementation of PeerTrust, which is identical across all parties. Each party has a queue that contains the queries that it needs to process. Each party has two threads. The first thread reads from the queue and processes one query at a time. Processing involves local query evaluation

(using a Prolog inference engine), sending out the answers to newly evaluated queries, and sending new queries to other parties, all depending on whether there are any new subqueries to be processed locally or remotely. Answered queries are deleted from the queue. If we send a query to another party, the query at the requester side is set to status "waiting" (other queries can be processed in the meantime). If the query does not return an answer, this is also forwarded to the requester. The second thread listens for new queries or answers sent by other parties to previously posed queries. It processes these messages and updates the queue accordingly. If a new query is received, we add it to the queue. If an answer is received, the query status (previously "waiting") is updated so the server can continue with its processing (evaluate one of the remaining subgoals).

5 Implementation

PeerTrust Application and Applet The outer layer of PeerTrust's negotiation software is a signed Java application or applet program[4], which implements the queues described in the previous section, parses incoming queries, translates them to the PeerTrust language, and passes them to the inner layer. The inner layer answers queries by reasoning about PeerTrust policy rules and credentials, and returns the answers to the outer layer. The current implementation of the inner layer of PeerTrust uses MINERVA Prolog[5], which provides a Prolog compiler and engine implemented in Java; this offers excellent portability and simplifies software installation.

The first implementation of PeerTrust [11] simulated distributed parties within one Prolog process. The current version, PeerTrust 1.0, is fully distributed, employs secure socket connections between all negotiating parties, and uses Prolog metainterpreters to evaluate queries over sets of policies and credentials. PeerTrust's facilities for communication between parties and access to security related libraries are written in Java. The Prolog meta interpreters, Java code, examples, and traces of their runs on different kinds of examples, as well as a demo application, are available at http://www.learninglab.de/english/projects/peertrust.html.

To import RDF metadata, we modified an open source RDF parser written in SWI-Prolog[6] to work with MINERVA Prolog. The modified parser extracts RDF predicates from XML and transforms them into the internal PeerTrust format for rules and facts.

Public Key Infrastructures and Authentication PeerTrust 1.0 uses Public Key Infrastructure (PKI), specifically X.509 [7], as the framework for how digital credentials are created, distributed, stored and revoked. With an X.509-based approach to trust negotiation, function calls are made at run time to verify the contents of each received credential by determining whether the credential really was signed by its supposed issuer. This may involve looking up the public key of the issuer at a trusted third party. Further, when a policy specifies that the requester must be a specific person mentioned

---

[4] The signature guarantees that the software has not been tampered with.

[5] See http://www.ifcomputer.co.jp/MINERVA/home_en.html.

[6] See http://www.swi.psy.uva.nl/projects/SWI-Prolog/packages/sgml/online.html for the original parser.

in a credential (for example, that Alice must be the police officer whose name appears on the police badge that she has disclosed), then an external function call to an authentication library is needed so that Alice can prove that she is that person, generally by interactively proving that she possesses the private key associated with the public key that in turn is associated with that particular field of the credential.

PeerTrust 1.0 relies on the TLS protocol and its authentication facilities (as implemented in the Java Secure Socket Extension package, JSSE[7]), and uses appropriate signed rules to relate the constants used in the logic programs, e.g., "alice," to Alice's public key. For conciseness, the policies that we present in this paper do not include these details, and are written as though Alice (and every other party) has a single global identity, "alice," and Alice has already convinced E-Learn that she is in fact "alice."

Importing and Using Digitally Signed Rules To guarantee that signed rules are issued by specific parties and that no one tampered with the rule, PeerTrust 1.0 uses digital signatures as defined by the Java Cryptography Architecture and implemented in the java.security.Signature class. Rules are signed with the private key of the issuer, and can be verified with the corresponding public key. PeerTrust 1.0's signature verification is based on MD5 hashes of the policy rules (stored in Prolog plain text format) and RSA signatures.

Authentication When a party submits a credential, in order to verify that it is a party mentioned in the credential, we have to obtain sufficient evidence that the submitter possesses the private key associated with a public key that appears in the credential. Earlier work in trust negotiation proposed an extension to TLS to conduct the negotiation as part of the TLS handshake [6]. The extension leverages the credential ownership mechanisms of TLS to verify credential ownership during trust negotiation.

Another approach to trust negotiation is to establish a standard SSL/TLS session and then conduct the trust negotiation over the secure connection, which is the approach we take in our current prototype. When credential ownership must be verified, there are several possible approaches. If one party disclosed a certificate during the SSL handshake, then any credentials subsequently disclosed during trust negotiation that contain the same public key can rely on the SSL handshake as adequate authentication. Otherwise a proof-of-identity protocol must be introduced over TLS. The protocol must be carefully designed to thwart man-in-the-middle or reflection attacks [12]. PeerTrust 2.0 will support authentication by borrowing from the proof-of-ownership techniques supported in TrustBuilder, the first trust negotiation prototype to support these security features. In TrustBuilder, the negotiation participants push a proof of ownership along with the credentials they disclose.

6 Pulling It All Together: Alice Signs Up to Learn Spanish

The interaction diagram in Figure 3 shows the negotiation details that were sketched in Figure 1, for the case of Alice enrolling in Spanish 101.

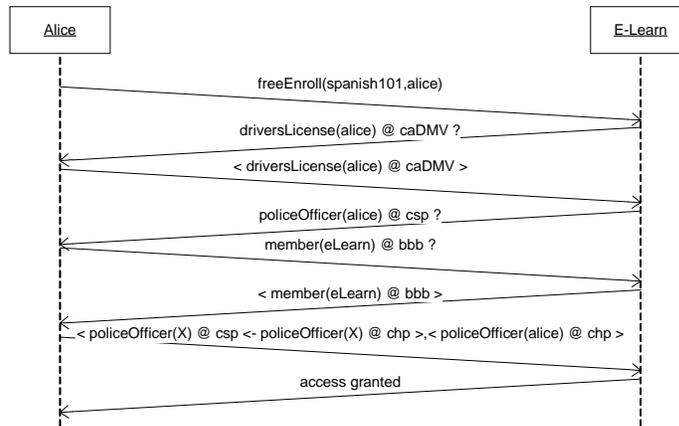[7] JSSE can be found at http://java.sun.com/products/jsse/.

Fig. 3. Negotiation between Alice and E-Learn

After Alice requests free access to the Spanish course offered by E-Learn Associates, E-Learn responds by asking for Alice's driver's license. Alice responds by sending back the appropriate signed proof (shown without its signature in Fig. 3), i.e.,

driversLicense(alice) @ caDMV
  signedBy [caDMV].

As the next step, E-Learn asks Alice to prove that she is a California state police officer. Alice can prove this by disclosing her police officer credential signed by CHP plus the rule signed by CSP that says that all CHP officers are CSP officers. The latter rule is freely disclosable, but as shown earlier, Alice's guard on her CHP credential restricts its disclosure to BBB members only. So, rather than disclosing her proof, Alice asks E-Learn about its membership status at the Better Business Bureau. After E-Learn produces a proof of membership signed by the BBB, Alice discloses her CHP credential and CSP's signed rule, and then E-Learn allows Alice to access the Spanish course.

7 Related Work

The Secure Dynamically Distributed Datalog (SD3) trust management system [8] is closely related to PeerTrust. SD3 allows users to specify high level security policies through a policy language. The detailed policy evaluation and certificate verification is handled by SD3. Since the policy language in SD3 is an extension of Datalog, security policies are a set of assumptions and inference rules. SD3 literals include a "site" argument similar to our "Issuer" argument, though this argument cannot be nested, and "Requester" arguments are not possible either, which is appropriate for SD3's target application of domain name service, but restricts SD3's expressiveness too much for our purposes. It also does not have the notion of guards.

The P3P standard [17] focuses on negotiating the disclosure of a user's sensitive private information based on the privacy practices of the server. Trust negotiation generalizes this by basing resource disclosure on any properties of interest that can be

represented in credentials. The work on trust negotiation focuses on certified properties of the credential holder, while P3P is based on data submitted by the client that are claims the client makes about itself. Support for both kinds of information is warranted.

Yu et al. [18] have investigated issues relating to autonomy and privacy during trust negotiation. The work on autonomy focuses on allowing each party in a negotiation maximal freedom in choosing what to disclose, from among all possible safe disclosures. Their approach is to predefine a large set of negotiation *strategies*, each of which chooses the set of disclosures in a different way, and prove that each pair of strategies in the set has the property that if Alice and E-Learn independently pick any two strategies from the set, then their negotiation is guaranteed to establish trust if there is any safe sequence of disclosures that leads to the disclosure of the target resource. Yu et al.'s approach to protecting sensitive information in policies is UniPro; PeerTrust supports UniPro's approach to policy protection, through PeerTrust's use of guards in rules.

Recent work in the context of the Semantic Web has focussed on how to describe security requirements in this environment, leading to the KAoS and Rei policy languages [9, 14]. KAoS and Rei investigate the use of ontologies for modeling speech acts, objects, and access types necessary for specifying security policies on the Semantic Web. PeerTrust complements these approaches by targeting trust establishment between strangers and the dynamic exchange of credentials during an iterative trust negotiation process that can be declaratively expressed and implemented based on GDLPs.

Similar to the situated courteous logic programs of [5, 4] that describe agent contracts and business rules, PeerTrust builds upon a logic programming foundation to declaratively represent policy rules and iterative trust establishment. The extensions described in [5, 4] are orthogonal to the ones described in this paper; an interesting addition to PeerTrust's guarded distributed logic programs would be the notion of prioritized rules to explicitly express preferences between different policy rules.

8 Conclusion and Further Work

This paper has tackled the problem of explicit registration needed for accessing protected resources on the Web, where the client has to provide a predetermined set of information to the server. We showed how explicit registration can be avoided on the Semantic Web by placing appropriate semantic annotations on resources that need protection, writing rule-oriented access control policies, and providing facilities for automated trust negotiation. Our PeerTrust language for expressing access control policies is based on guarded distributed logic programs. The Java/Prolog-based PeerTrust 1.0 prototype provides trust negotiation capabilities for servers and clients, with facilities to import and reason about access control policies, digital credentials, and metadata about local resources requiring protection.

PeerTrust 2.0 will include import/export of PeerTrust policies in RuleML format and the use of XML digital signatures (based on the Apache XML Security Suite), as well as interfaces to protect general resources such as Semantic Web services.

References

1. S. Brands. *Rethinking Public Key Infrastructures and Digital Certificates: Building in Privacy*. The MIT Press, 2000.
2. J. Camenisch and E. Herreweghen. Design and Implementation of the *Idemix* Anonymous Credential System. In *ACM Conference on Computer and Communication Security*, Washington D.C., Nov. 2002.
3. D. Eastlake, J. Reagle, and D. Solo. Xml-signature syntax and processing. W3C Recommendation, Feb. 2002.
4. B. Grosof. 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.
5. B. Grosof and T. Poon. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *Proceedings of the 12th World Wide Web Conference*, Budapest, Hungary, May 2003.
6. A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. Seamons, and B. Smith. Advanced Client/Server Authentication in TLS. In *Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2002.
7. International Telecommunication Union. *Rec. X.509 - Information Technology - Open Systems Interconnection - The Directory: Authentication Framework*, Aug. 1997.
8. T. Jim. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
9. 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.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd edition edition, 1987.
11. W. Nejdl, D. Olmedilla, and M. Winslett. PeerTrust: automated trust negotiation for peers on the semantic web. Technical Report, Oct. 2003.
12. B. Schneier. *Applied Cryptography, second edition*. John Wiley and Sons. Inc., 1996.
13. B. Simon, Z. Miklós, W. Nejdl, M. Sintek, and J. Salvachua. Smart space for learning: A mediation infrastructure for learning services. In *Proceedings of the Twelfth International Conference on World Wide Web*, Budapest, Hungary, May 2003.
14. 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.
15. J. Trevor and D. Suciu. Dynamically distributed query evaluation. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, Santa Barbara, CA, USA, May 2001.
16. K. Ueda. Guarded horn clauses. In *Logic Programming '85, Proceedings of the 4th Conference*, LNCS 221, pages 168–179, 1986.
17. W3C, http://www.w3.org/TR/WD-P3P/Overview.html. *Platform for Privacy Preferences (P3P) Specification*.
18. 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.