Negotiating Trust on the Grid

J. Basney[3], W. Nejdl[1], D. Olmedilla[1], V. Welch[3], and M. Winslett[2]

[1] L3S and University of Hannover, Germany {nejdl,olmedilla}@learninglab.de
[2] University of Illinois, USA winslett@cs.uiuc.edu
[3] National Center for Supercomputer Applications, USA
{jbasney,vwelch}@ncsa.uiuc.edu

Abstract. Grids support dynamically evolving collections of resources and users, usually spanning multiple administrative domains. The dynamic and cross-organizational aspects of Grids introduce challenging management and policy issues for controlling access to Grid resources. In this paper we show how to extend the Grid Security Infrastructure to provide better support for the dynamic and cross-organizational aspects of Grid activities, by adding facilities for dynamic establishment of trust between parties. We present the PeerTrust language for access control policies, which is based on guarded distributed logic programs, and show how to use PeerTrust to model common Grid trust needs.

1 Introduction

Ideally, a Grid environment provides its users with seamless access to every resource that they are authorized to access, enabling transparent sharing of computational resources across organizational boundaries. Ideally, the ingredients for transparent sharing are few and simple: digital *certificates*, i.e., verifiable, unforgeable statements signed by the parties that create them (*authorities*); and an *access control policy* for every resource on the Grid, saying what kinds of certificates from which authorities must be presented to gain access to the resource. As an illustration of potential power of this approach, consider the following motivating example, depicted in fig. 1.

Alice submits an earthquake simulation job to a NEESgrid Linux cluster via the Globus Resource Allocation Manager (GRAM) protocol (see www.globus.org). This requires mutual Grid Security Infrastructure (GSI) authentication between Alice and the Globus Gatekeeper that controls access to the cluster, using SSL/TLS and X.509 certificates. In this case, these certificates are issued by a well-known NEESgrid Certification Authority (NCA), and say that a party with a particular distinguished name ("Alice Smith", "NEESgrid Linux Cluster Gatekeeper") has a particular public key. When Alice submits her X.509 certificate to the Gatekeeper, she pushes a proof that she owns the certificate as well, by signing a challenge with the private key for her X.509 certificate. The Gatekeeper follows a similar procedure to prove its identity to Alice.

Once authentication is complete, the Globus Gatekeeper for the cluster will verify that Alice is authorized to submit the job, by consulting the local grid-mapfile (www.globus.org/security/v1.1/grid-mapfile.html). This file maps the subject mentioned in Alice's X.509 certificate to a local Unix account; in other words, Alice will already have preregistered to use the cluster.

Dagstuhl Seminar Proceedings 05271
Semantic Grid: The Convergence of Technologies
http://drops.dagstuhl.de/opus/volltexte/2005/387

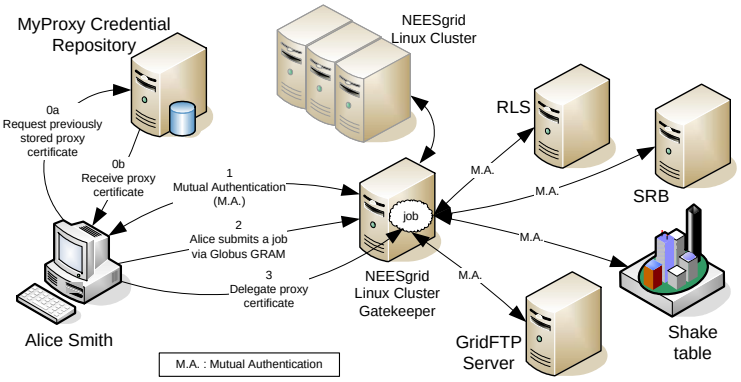

Fig. 1. Job Interaction with Grid Resources

Alice also delegates a GSI proxy certificate [18] to her job. This proxy certificate asserts the identity of the new job, by binding a new distinguished name for the job ("Alice's job") to a new public key. The certificate also says that the entity with distinguished name "Alice Smith" has delegated to the entity with distinguished name "Alice's job" the right to act on her behalf, i.e., to make use of her privileges to access remote resources. Whenever Alice's job needs to gain access to a new resource, it will present this certificate, which it uses to prove that it is acting on Alice's behalf.

After receiving the job description and a delegated credential, the Gatekeeper then passes these to a local job scheduler (e.g., Portable Batch System (PBS), Condor, IBM's Load Leveler, or LSF) for execution. When the local job scheduler starts Alice's job, the job queries a Replica Location Service (RLS) (www.globus.org/rls/) to determine where on the Grid to find its input data. This requires mutual authentication using GSI between Alice's job and RLS. In other words, the job must authenticate to the service by showing its proxy certificate and proving that it is the rightful possessor of the certificate; the job must verify that it is talking to RLS, by having the service prove that it possesses the RLS identity; and RLS must decide that Alice's job is authorized to access RLS.

Once Alice's job has located its input data, the job gets the data via GridFTP from a mass storage system. This requires mutual GSI authentication between the job and the GridFTP server running at the mass store. The job retrieves additional input data from an unknown location using the Storage Resource Broker (SRB) (www.npaci.edu/DICE/SRB), which requires an additional round of mutual GSI authentication.

As it executes, the job combines simulation of an earthquake on a parallel platform with data obtained in real time from scientific instruments—shake tables and wave tanks. Each instrument requires a round of mutual GSI authentication before it will interact with Alice's job. The job sends its output to another remote storage service using GridFTP, requiring an additional round of mutual authentication. By the end of the run, Alice's jobs have performed seven or more rounds of mutual authentication with resources. All except the initial round have used the proxy certificate that Alice gave to her job, with no additional effort required on Alice's part.

In practice, NCA may find it too inconvenient to issue X.509 certificates itself, and may delegate the right to issue these certificates to other entities, e.g., NCA-1 and NCA-2. The resulting hierarchical security space is more scalable and manageable than a flat space. If Alice's X.509 certificate is issued by NCA-1, then when she wants to authenticate herself to a service that requires certificates signed by the NCA root authority, she will need to submit her X.509 certificate issued by NCA-1 as well as NCA-1's delegation certificate issued by NCA. Such chains of delegation certificates are an ubiquitous fact of life in trust management. Similarly, Alice's job may spawn new subjobs as it runs. The new subjobs will each be given their own identity and public/private key pair, along with a proxy certificate that records their identity, public key, and the fact that Alice's job has delegated to the subjob the right to act on Alice's job's behalf. For brevity, we use delegation chains of length 1 in examples when possible, and abbreviated forms of distinguished names.

As a Grid grows larger and as jobs become more complex, seamless authorization and authentication becomes harder to provide. Larger Grids will evolve to become a patchwork of resources provided by peers who zealously guard their autonomy and do not fully trust one another. Owners who provide resources that are part of a large Grid will often not be willing to give up authority to decide who is entitled to access those resources. In the example above, the practice today is for the job to authenticate to all services with its single proxy certificate, delegated to it on initial submission via GRAM. However, on today's Grids one already sees disagreement about which issuers of identity certificates are trusted at different sites, so users end up with multiple X.509 identity certificates issued by different authorities.

An approach to authorization and authentication for large Grids that is based on possession of a large set of identity certificates and/or the existence of local accounts for each user will not provide the scalability, flexibility, and ease of management that a large Grid needs to control access to its sensitive resources. In part, this is because authorization decisions will depend on properties other than identity and resource owners will not have access to a full description of the properties of each possible user. For scientific Grids, such properties may include whether the requesting job is acting on behalf of a project PI, student, or administrator; whether the requester is acting on behalf of a US citizen or non-citizen; and whether the requester is acting on behalf of a member of a particular research project whose membership list is not maintained locally.

A second reason why identity certificates and local accounts cannot be the basis for authorization decisions in a large-scale Grid is that a single job may access resources with many different owners with different trust requirements, and users cannot be expected to obtain and keep track of all the associated certificates and private keys—they need software to automate the process. Nor can resource owners have a local account in place for every potential user or keep track of all the relevant changes in users' qualifications, such as group memberships. Already we are seeing new authorization certificates starting to be available on Grids (PRIMA [12], VOMS [1], CAS [14], and X.509 attribute certificates, which allow authorities to attest to properties other than identity, as mentioned above). The need for software to manage these certificates on users' behalf has been noted, and in response software such as MyProxy [13], a certificate wallet for Grid users, is being developed. However, as yet there is no standard API that users can

call to find out what certificates to submit to gain access to a particular resource, and no way for a particular resource's owner to tell users what certificates to submit. Nor are there ways to establish trust gradually: who shows their certificates first is hard-wired, with no chance to start off by disclosing relatively insensitive certificates before moving on to more private information like citizenship or clearances. There is also no way to say, "I'll show you my US citizenship certificate *if* you prove that you really are Service XYZ first."

Trust negotiation is an approach to authorization and authentication that addresses all the problems described in the previous paragraphs. With trust negotiation, trust is established iteratively and bilaterally by the disclosure of certificates and by requests for certificates; those requests may be in the form of disclosures of access control policies that spell out exactly which certificates are required to gain access to a particular resource. This differs from traditional identity-based access control systems:

1. Trust between two strangers is established based on parties' properties, which are proven through disclosure of digital certificates.
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, accounts, documents and other data, roles in role-based access control systems, certificates, 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 certificate 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 procedure, trust is established incrementally through a sequence of bilateral certificate disclosures.

A trust negotiation is triggered when one party requests to access a resource owned by another party. The goal of the negotiation is to find a sequence of certificates $(C_1, \ldots, C_k)$, such that when certificate C_i is disclosed, its policy has been satisfied by certificates disclosed earlier in the sequence, and such that $C_1, \ldots, C_k$ satisfy the policy of the targeted resource—or to determine that no such sequence exists.

In this paper, we propose the use of automated trust negotiation as a scalable approach to access control in Grid environments. In Section 2 we show how Alice's job can use trust negotiation in the scenario above. Then we introduce the PeerTrust language for expressing policies and guiding negotiations in Section 3 via examples. We conclude with a discussion of related work and a brief look at further research issues.

2 How Alice's Job Uses Trust Negotiation on the Grid

A trust negotiation will take place every time that Alice's job tries to access a new resource on the Grid. For example, when her job tries to access the shake table resource using https, the resulting negotiation may look as follows (fig. 2):

Step 1. Alice's job sets up a TLS session with the shake table access manager, which intercepts all requests sent to the shake table. When the session is in place, all further

communication will be protected from eavesdropping, so the negotiation will be private. Alice's job and the shake table access manager do not discuss their access control requirements until the session is in place. (In a TLS handshake, the server presents its identity certificate chain first, and the client must present its own certificate chain before the server has proved that it owns the leaf certificate in the server's chain; this will not always suit the needs of the two parties. Here, the identity certificates are being used only to establish a private session, and need not reveal any sensitive information about either party, such as their identity in any well-known namespace.)

Step 2. As soon as the session is established, Alice's job initiates a TLS rehandshake, using an extended version of the rehandshake that supports trust negotiation [7]. A TLS rehandshake can be initiated by either party at any point during a TLS session. In this rehandshake, Alice's job sends the shake table a policy, written in RT [11], that must be satisfied before Alice's job is willing to partake of further communication. This policy says that the device that Alice's job is talking to must prove that it really is the shake table, by presenting an X.509 certificate chain for the shake table's identity, with the root certificate in the chain issued by the NEESgrid Certificate Authority or the NPACI CA, and proving that it is the rightful owner of the leaf certificate in the chain. At this point, Alice's job has not yet sent any actual request to the shake table, i.e., the shake table has not yet seen the URL (for a GET request) or the form data (for a POST request), so her intended use of the shake table remains private [7]. Alice's job only sends her policy at this point, i.e., no additional credentials are sent during this step, and no additional authentication is performed.

Step 3. The shake table's access manager is willing to prove its identity to anyone, and the root certificate in its chain is signed by the NEESgrid CA, an issuer that Alice's job listed as acceptable. The shake table already has this certificate chain cached locally, as it uses that chain constantly to prove its identity to potential users. The shake table sends the identity certificate chain to Alice's job and proves that the shake table owns the leaf certificate in the chain, by signing a challenge with the correct private key and pushing it to Alice with the chain.

Step 4. Before communication proceeds further, the shake table access manager wants to know who it is talking to, to establish an audit trail. Still using the TLS rehandshake facility, it asks Alice's job to submit a proxy certificate chain whose root is signed by one of a listed set of issuers that the shake table finds acceptable.

Step 5. Alice's job already has a proxy certificate chain whose root is signed by the NPACI Certificate Authority, an issuer that the shake table finds acceptable. A proxy certificate is a special kind of certificate used to represent the delegation of authority from one entity to another. The job obtained this chain when it was first created, i.e., when Alice logged into a Grid portal and asked her MyProxy server [13] to issue such a credential to her new job. However, Alice's job is only willing to disclose its/her identity to a resource that has already disclosed its own identity (certified by an acceptable issuer). That policy has been satisfied by the previous disclosure from the access manager, so Alice's job submits its proxy certificate chain and proves that it owns the leaf certificate in the chain.

Step 6. The shake table access manager now knows that the job is acting on Alice's behalf. It has never heard of Alice before. Any NPACI user can query the status of the

shake table and view its web cam, so the fact that Alice is a complete stranger does not matter at this point. The rehandshake is completed, and session key resumption is used to carry on the TLS session with the same symmetric key that was established during the initial TLS handshake.

Alice's job sends her actual request to the shake table, within the same TLS session. The access manager intercepts the request, and sees that Alice's job wants to shake the table. It maps this request to the associated access control policy, which says that requests must come from a job acting on behalf of a member of a project group that is allowed to access the table on that day of the week. Further, the table must not already be in use by anyone else. The access manager initiates another TLS rehandshake and forwards this policy to Alice's job.

Step 7. Alice's job does not know whether Alice belongs to an appropriate project. It contacts Alice's MyProxy server and a Community Authorization Service server (discussed later) for help in satisfying the shake table's policy; details of their conversation are omitted here. A separate trust negotiation ensues, because Alice's MyProxy server is only willing to help jobs acting on behalf of its subscribers.

Among others, Alice's BigQuake membership certificate is stored at her MyProxy server. Her MyProxy server looks up the policy for disclosure of Alice's BigQuake membership certificate, and forwards it and the certificate to Alice's job. CAS requires proof of her project membership before it will tell Alice that her BigQuake project can access the shake table on that day.

Step 8. Since the policy for Alice's BigQuake membership certificate says that the certificate can be disclosed to any NEESgrid resource, the policy has been satisfied by Alice's job's earlier negotiation with the shake table access manager, in which the shake table produced a certificate signed by the NEESgrid Certificate Authority. The job forwards Alice's membership certificate to the shake table access manager, using the TLS rehandshake facility. Because the identity/distinguished name (Alice) in the membership certificate exactly matches the distinguished name in the proxy certificate chain that Alice's job presented earlier, Alice's job does not need to do any additional authentication when it presents the certificate.

Step 9. No one is currently using the shake table, so the shake table access manager sees that the policy for shaking the table is satisfied by the certificates that Alice's job has presented so far, and lets Alice's job shake the table.

The example above describes how trust negotiation takes place when using the TrustBuilder prototype for TLS [7], the RT policy language [11], and a certain set of conventions for negotiation. However, trust negotiation facilities can be embedded in any communication protocol; for example, the TrustBuilder project (http://isrl.cs.byu.edu/) has deployments of trust negotiation in TLS [7], SMTP and POP3 [8], ssh and HTTPS. Other common comunication protocols, including GridFTP, IPsec, and SOAP, are targeted for future deployments of TrustBuilder. In this paper, we adopt TrustBuilder as the underlying infrastructure for trust negotiation in common Grid communication protocols. Figure 3 shows the overall architecture used in all deployments of TrustBuilder. The large "security agent" box represents the software for trust negotiation that is embedded in the access manager for a resource. The agent has access to Alice's locally-stored certificates and access control policies, so that it can

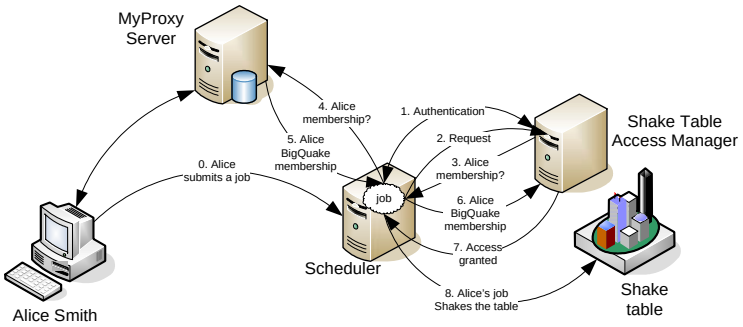

Fig. 2. Negotiation on the Grid

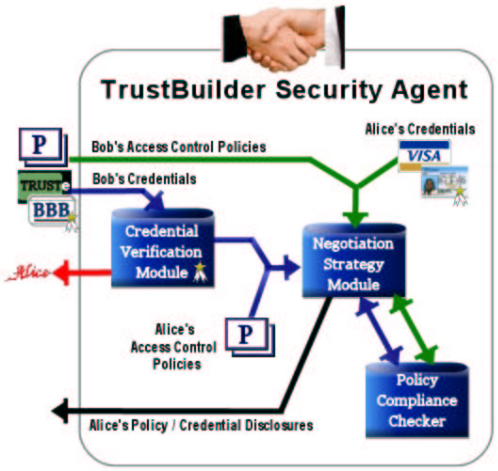

Fig. 3. Architecture of the Trust Negotiation Component of an Access Manager

negotiate on her behalf. For each policy language the agent understands, it has a policy compliance checker, so it can determine whether a policy of Alice's is satisfied by a set of credentials that Bob has sent. The checker also helps the agent to determine whether Alice has sets of credentials that can satisfy a policy of Bob's. Given all the credentials and/or policies that Alice could disclose next, the strategy module determines which to disclose based on Alice's concept of need-to-know and her strategic preferences, e.g., for short negotiations or for negotiations that help her collect information for her marketing database. Once the strategy module decides what to disclose next, the disclosures are bundled into a message type appropriate for the underlying communication protocol (e.g., email for SMTP, rehandshake for TLS) and sent to the other party. The agent's verification module verifies the signatures on all credentials sent to Alice, and checks that their contents have not been tampered with. The modules that understand policy languages do not see the signatures themselves. For further information on how

to embed trust negotiation facilities into common communication protocols, we refer the reader to the sources referenced above.

In principle, any policy language designed for trust negotiation can be used with TrustBuilder, as can any set of conventions for how to carry out a negotiation. In the next section, we use the PeerTrust policy language to describe common kinds of Grid access control policies. Those examples use objective criteria for establishing trust (e.g., group membership), because this is how today's Grids for high-performance computing are set up; but trust criteria based on reputation can also be expressed in access control policies and supported by trust negotiation.

To address some of the shortcomings in promulgating security-related information on Grids, Grid researchers are working to adopt Web Services technologies as fundamental building blocks [4]. This includes the use of SOAP and WS-Security (www.oasis-open.org) as mechanisms to augment TLS and allow for the attachment and transport of arbitrary security-related information, and WSDL (www.w3.org/TR/wsdl) as a standard mechanism for allowing services to expose their functionality and security policies (www.ibm.com/ developerworks/library/ws-policy). Our view is that trust negotiation facilities should be embedded in all common communication protocols, so that authorization needs at every system level can be satisfied in the most natural way. For example, trust negotiation at the TLS level allows a party to establish a certain level of trust *before* disclosing the exact request (e.g., URL, form data, and/or SOAP headers) that it is sending to a server. If trust negotiation is available only at higher levels of software, such as in HTTP or SOAP parameters, then that ability is lost. Conversely, if trust negotiation is available only at low levels of the system, such as TLS, then it may be cumbersome for applications and users to pass all their authorization-related information back and forth at the TLS level, when the use of HTTP/SOAP headers would be much simpler and more natural. Similarly, access control policies can be valuable corporate assets that are not made available for everyone's perusal. Thus a call to WS-Policy to examine a particular policy may trigger a trust negotiation to prove that the requester is authorized to view the policy. We believe that trust negotiation facilities should be available wherever needed, i.e., in *all* popular communications protocols. In the remainder of this paper, we focus on the higher-level aspects of trust negotiation that are the same for its deployment in all communication protocols: the expression of access control policies and reasoning about their contents.

3 The PeerTrust Policy Language

To use automated trust negotiation, we need widely understood languages for specifying access control policies, so that a Grid resource owner and potential user can agree on whether a particular policy is satisfied by a particular set of certificates. Further, such languages must admit efficient determination of whether a policy is satisfied and of which certificates will satisfy a policy. Such languages must also be expressive enough to describe all common Grid access control requirements, including descriptions of delegation agreements and restricted delegations. Ideally, the same languages should also be useful in non-Grid scenarios, such as the Semantic Web.

Logic programs are an obvious substrate for declarative, universally comprehensible policy languages. In fact, recent efforts to develop policy languages suitable for use in trust negotiation have all chosen logic programming as a substrate (RT [11], Cassandra [2], SD3 [17], Bonatti and Samarati [3] with an emerging consensus that constraint logic programs are a very promising choice. PeerTrust also uses a logic programming basis (datalog plus lists, with some new syntactic features described below); for simplicity, we do not use constraints in this paper, because their additional expressivity is not needed for our examples. In general, however, constraints offer an easy way to describe certain kinds of common access control restrictions, e.g., that the shake table can only be accessed by projects whose project numbers start with 46352, or every file in every subdirectory of a particular directory should be accessible. Further, this ease of description comes without a significant compromise in the efficient evaluation of policies. In this short paper, we consider only positive authorizations and policies without negation. We assume that the reader is familiar with the syntax and semantics of simple logic programs.

The motivating examples in the previous sections relied on the use of X.509 certificates for simple proofs of identity. In scenarios such as those, the associated access control policies typically implicitly assume that the party who submits certificates must authenticate to the identity mentioned on the leaf certificate of each submitted chain. In PeerTrust, all authentication requirements are made explicit. This allows PeerTrust policies to make use of more complex certificates that describe relationships between several parties (as opposed to the properties of a single owner), such as a birth certificate. In PeerTrust, a policy can require a party to authenticate to any identity mentioned in a certificate.

3.1 Alice Accesses the Wave Tank

To illustrate the features of PeerTrust, we use the example of Alice's job trying to stir the wave tank. All attempts to access the wave tank are intercepted and filtered by the wave tank access manager, which has an access control policy for each kind of attempted access. The wave tank access manager requires each requester to authenticate itself to the manager and to present a proof that it is acting on behalf of some party (typically a human user). The tank manager relies on a Community Authorization Service with distinguished name "CAS-1" as an authority on which parties belong to which project group, and relies on service CAS-2 as an authority on which project groups are authorized to access the resources it manages. Further, the tank manager expects the requester to provide the proof of group membership. Finally, each project group can only use the wave tank on certain days, and only one job at a time can use the tank.

Access Requests and Queries. In the TrustBuilder/RT shake table example, the communication between parties consisted of disclosures of policies (implicit requests for certificates) and certificates. In the version of PeerTrust described in this paper, messages between parties take the form of logic programming queries and proofs that certain pieces of information are answers to those queries. For example, when Alice's job tries to stir the wave tank, the wave tank access manager intercepts the request and generates the following query to its local PeerTrust system:

”wave tank access manager”:
?- grant(”Alice’s job”, ”stir”, ”wave tank”) $ ”Alice’s job”.

As with PeerTrust statements, this query is prefixed by an identity of the party that possesses it, in this case the wave tank access manager. The manager has only one identity, the distinguished name listed in the X.509 certificate issued by NCA to bind “wave tank access manager” to a particular public key. In contrast, we expect Alice to have many identities, each corresponding to an X.509 certificate created by a different authority (possibly even self-issued). On the Grid today, authorization statements refer to parties by their distinguished names, rather than by their public keys as is done in some other trust management scenarios. Thus our policies also refer to parties by their X.509 distinguished names, although for brevity in this paper we are using abbreviations such as “wave tank access manager” and “Alice’s job” instead of writing out the full distinguished names. Each Grid Certification Authority guarantees that it will bind a particular distinguished name to at most one public key at a given point in time.

‘$ “Alice’s job” ’ has been appended to the end of the query to represent the fact that the query is coming from a party with the identity “Alice’s job”. The identity of the requesting party is appended to each query because the party receiving the query may give different answers depending on who asks the query, or may refuse to answer the query at all. More generally, any literal can have an argument of the general form “$ Requester” appended to it. We require that each Requester be instantiated with an *authenticated* identity, e.g., that it be the authenticated identity that the party who tried to access the resource used to set up the underlying TLS connection; it can be a throw-away identity used only for that purpose and not certified by any well-known authority. For example, Alice’s job might self-issue an identity certificate that she uses only for setting up a secure channel for communication, so that she does not give out any privileged information before trust is established. When using the FTP protocol, Alice’s job might authenticate to the identity “guest” using an email address as her password. In the current example, we assume that Alice’s job sets up the underlying communication channel using the identity “Alice’s job” that was given to it when its parent job spawned it. More generally, whatever underlying protocol is used for communication, we assume that it provides a private and authenticated channel for communication between the parties that it connects.

Policies. The tank manager has a policy for who is allowed to access the wave tank:

```
”wave tank access manager”:
grant(Job, Operation, Resource) $ Job ←
  actingOnBehalfOf(Job, User) |
  member(User, Project) @ ”CAS-1” @ Job,
  grant(Project, Operation, Resource) @ ”CAS-2”,
  accessManager(Resource, ”wave tank access manager”),
  currentDay(Today),
  authorizedDay(Project, Today),
  notAlreadyInUse(Resource, Job).
```

The tank policy says that if seven conditions are met, then the wave tank access manager will allow a particular Job to perform a particular Operation on a particular

Resource. '$ Job' has been appended to the head of the rule to represent the fact that this rule is used only to answer queries generated by the job that trying to access the resource. In other words, if a third party asked the wave tank access manager whether Alice's job could stir the tank, the manager would refuse to answer the query (assuming it had no other relevant rules).

The fourth condition of the tank policy is the simplest; it says that the Resource must be one that the wave tank access manager manages. accessManager is an extensional predicate, i.e., the wave tank access manager stores facts that list all the resources it manages. For our purposes, the relevant stored fact is 'accessManager("wave tank", "wave tank access manager").'

Authorities and Signatures. The ability to reason about statements made by other parties is central to trust negotiation. The second and third conditions of the wave tank policy say that the wave tank access manager considers CAS-1 to be an authority on which groups each user belongs to, and CAS-2 to be an authority on which groups are allowed to access the manager's resources. The literal 'grant("BigQuake", "stir", "wave tank") @ "CAS-2"' is derivable at the wave tank access manager if CAS-2 tells the manager that 'grant("BigQuake", "stir", "wave tank")' is true. In PeerTrust, a party can send queries to other sites[4] as a means of determining what is considered to be true at those sites. In PeerTrust, there may be many formulas that are true at CAS-2 but underivable for the wave tank manager. For example, user Bob may be a member of the BigQuake project according to CAS-1, but the wave tank access manager will probably be unable to derive that fact unless Bob tries to access the wave tank.

When the wave tank manager asks CAS-2 if the BigQuake project can stir the wave tank (later on, we will see how the tank manager knows to ask this particular query), the query is represented at CAS-2 as follows:

"CAS-2":
?- grant("BigQuake", "stir", "wave tank") $ "wave tank access manager".

Again, '$ "wave tank access manager"' at the end of the query represents the fact that the query comes from a party who has authenticated to the identity "wave tank access manager".

We assume that given an identity for a party, e.g., taken from an X.509 certificate from a well-known issuer or from an identity that the party used to set up a communication channel, the system underlying PeerTrust is able to send a message to that party. Thus, for example, the wave tank access manager is able to send a message to CAS-2 containing the query described above. In the version of PeerTrust described in this paper, each query answer takes the form of a *signed proof*, i.e., a derivation tree which includes signatures specifying which party is responsible for which part of the derivation tree. For the moment, let us simply assume that CAS-2 responds to the wave tank access manager's query by saying that the BigQuake project can stir the wave tank:

"wave tank access manager":
grant("BigQuake", "stir", "wave tank") signedBy ["CAS-2"].

[4] We use the words "site" and "identity" interchangeably. A party may have many identities, and chooses a particular identity to use whenever it sends a query.

CAS-2 is required to sign its answer before it forwards it to the wave tank access manager. More generally, in the version of PeerTrust described in this paper, all pieces of the proof bear digital signatures, so they form a nonrepudiable proof that a particular answer was given by the signer. The identity in the "signedBy" term is the distinguished name in the X.509 certificate for the public key used in the signature attached to the answer. The wave tank access manager can cache the answer and use it to handle subsequent access requests from the BigQuake project without having to send messages to CAS-2. Similarly, if the wave tank access manager ever wants to prove to a third party that 'grant("BigQuake", "stir", "wave tank") @ "CAS-2"' is true, the manager can prove it by sending the signed and cached answer from its earlier query to CAS-2. The underlying semantic principle in both cases is that given any signed rule at a site, e.g., 'literal signedBy ["identity"]', it follows that 'literal @ "identity"' is true at the same site. Similarly, given any rule of the form

site:
head $ Requester $\leftarrow$ signedBy [identity] literal$_1$, ..., literal$_n$.

we also have

site:
head $ Requester @ identity $\leftarrow$ literal$_1$ @ identity, ..., literal$_n$ @ identity.

Let us turn our attention to the second condition in the body of the tank policy, 'member(User, Project) @ "CAS-1" @ Job'. For User "Alice Smith", the tank manager will only be able to derive 'member("Alice Smith", project) @ "CAS-1" @ "Alice's job"' for a particular project Alice's job is willing and able to obtain the appropriate project membership certificate and send it to the wave tank access manager. To start the process, the wave tank access manager sends a query '?- member ("Alice Smith", Project) @ "CAS-1"' to Alice's job. Alice's job may already have cached a signed statement from CAS-1 that lists one or more projects that Alice Smith belongs to; if not, Alice's job can ask CAS-1 for such a statement. In the latter case, the query received by CAS-1 has the following form:

"CAS-1":
?- member("Alice Smith", Project) $ "Alice's job".

For the moment, let us assume that CAS-1 eventually responds to Alice's job's query by saying that Alice is a member of exactly one project, BigQuake. Then the received answer has the following form:

"Alice's job":
member("Alice Smith", "BigQuake") signedBy ["CAS-1"].

Alice can then sign the answer herself and send it to the wave tank access manager, producing

"wave tank access manager":
member("Alice Smith", "BigQuake") signedBy ["CAS-1", "Alice's job"].

Returning to the tank policy, we see that if the tank manager has learned all of the information described above (i.e., the relevant authorities have said that Alice Smith is a member of the BigQuake project, which is allowed to stir the tank), then bound versions of the conditions in the body of the policy are all derivable at the manager's site, except for possibly the first, sixth, and seventh. The sixth condition is true at the wave tank access manager if Today is the BigQuake project's day to stir the tank, where Today is instantiated by a call to an external "built-in"predicate currentDay, whose meaning is widely accepted by all parties (e.g., the day of the week at the site making the access control decision). The notAlreadyInUse(Resource, Job) literal also involves an external call. Let us assume that the wave tank access manager's list of facts for the "authorized-Day" predicate indicates that the BigQuake project is allowed to stir the tank today, and no one is currently stirring the tank.

Guards. We have now considered all but the first condition in the body of the tank policy. Notice the "|" symbol that separates this literal from the remainder of the body of the rule. When the "|" symbol is present in the body of a rule, all the body literals to its left form the *guard* of the rule. Guards do not affect the semantics of a rule but do affect its evaluation in practice, by requiring that all the literals in the guard must evaluate to "true" before the remainder of the literals in the rule can be evaluated. In this particular case, the variable instantiations from the head of the policy allow us to instantiate the guard literal as 'actingOnBehalfOf("Alice's job", User)'. This guard checks to see which user has spawned this job, by requiring a proof that a user says that Alice's job is acting on the user's behalf. After a discussion of CAS-1's and CAS-2's policies, we will consider how Alice's job can provide such a proof to the wave tank access manager. Assuming that we get a proof for the case where User is "Alice Smith", and the remaining literals in the body of the policy are all true at the wave tank access manager as discussed above, then the head of the policy rule, 'grant("Alice's job", "stir", "wave tank") $ "Alice's job"' is also true at the wave tank access manager. Thus the manager can allow Alice's job to stir the wave tank.

In the tank policy, guards are used to improve efficiency. A more important use is to limit derivability. Corporate and personal policies often contain sensitive information that should not be disclosed to just any party. The use of guards ensures that queries corresponding to the sensitive parts of a policy will not be sent out until a certain threshold of trust, as defined by the rule's guard, has been established. The simple scientific Grid examples used in this paper do not involve sensitive policies.

Users, Jobs and Delegation. CAS-1 and CAS-2 have their own policies on what kinds of queries they are willing to answer. CAS-1 will accept queries about project groups from users' jobs. CAS-2 will accept queries about groups from resource managers and users' jobs. These policies, and the relevant facts, are as follows:

"CAS-1":
member(User, Group) $ Job ← // queries from users' jobs
  actingOnBehalfOf(Job, User), authenticatesTo(Job, "CAS-1", "NCA") @ Job |
  member(User, Group). member("Alice Smith", "BigQuake").

```
"CAS-2":
grant(Group, Operation, Resource) $ Job ← // queries from users' jobs
  actingOnBehalfOf(Job, User), authenticatesTo(Job, "CAS-2", "NCA") @ Job,
  member(User, Group) @ "CAS-1" @ Job | grant(Group, Operation, Resource).
grant(Group, Operation, Resource) $ Requester ← // queries from resource managers
  authenticatesTo(Requester, "CAS-2", "NCA") @ Requester |
  accessManager(Resource, Requester), grant(Group, Operation, Resource).
accessManager("wave tank", "wave tank access manager").
grant("BigQuake","stir", "wave tank").
```

The wave tank access manager, CAS-1, and CAS-2 all have predicates called member and authenticatesTo. However, each of these predicates is defined locally—there is no a priori global agreement on how the predicates are defined or what their extensions are. For example, CAS-2 might not believe that the wave tank access manager manages the wave tank, even though the wave tank access manager does believe it. That is why both sites' policies check (local versions of) the same predicate. More generally, in trust negotiation we must be able to distinguish between predicates that can be queried by external entities and appear in query answers, and ones that cannot—analogous to the public and private procedures in object-oriented languages. Entities may also have private rules, which are neither shown to nor can directly be applied by other entities. The bodies of public rules can include predicates that are defined partly or entirely in private rules. Public and private rules and predicates are straightforward to design and implement in definite Horn clauses. In this paper we use only public rules, because the support for privacy that private rules provide is not needed in our scientific Grid examples. We assume that all parties are using a shared ontology for the (public) predicates that appear in queries and query answers, so that they are intelligible without an intermediate translation step.

The literals involving predicates actingOnBehalfOf and authenticatesTo in CAS-1's and CAS-2's rules require further explanation. In the examples in this paper, authenticatesTo(Identity, Party, Authority) @ Requester is derivable at Party and at Requester, if Requester has proved to Party that Requester has identity Identity, using an X.509 certificate issued by Authority. As with other predicates, the exact interpretation of the external predicate authenticatesTo can vary from site to site; other possible definitions include demonstrating knowledge of the password associated with a particular login at site Authority or passing a biometric test as defined by Authority. One can also use the authenticatesTo predicate to define more complex authentication-related predicates that include, for example, constraints on the length of the X.509 identity certificate chain and on the identity of the Authority at the root of the chain.

Recall that the third condition in the body of the wave tank policy is satisfied if the wave tank access manager determines that the literal grant("BigQuake", "stir", "wave tank") is true at CAS-2. Suppose that the wave tank manager queries CAS-2 for this information, and authenticates itself to CAS-2 during channel setup using its only identity, "wave tank access manager". The answer to the query depends on the set of variable bindings under which the literal authenticatesTo(Requester, "CAS-2", Authority) @ Requester is derivable at CAS-2. CAS-2 already knows one binding for Requester, because the requester already authenticated to "wave tank access manager" during con-

nection setup. However, this authentication might not have used a certificate issued by NCA, or might have used another kind of authentication not acceptable for the purpose of this particular policy. This can be remedied if needed through the external procedure call associated with authenticatesTo. During the course of this procedure call, Requester can sign a challenge from CAS-2 using the public key associated with another of Requester's distinguished names, and return the signed challenge to CAS-2 along with a copy of the X.509 certificate that binds that key to that distinguished name. The uniqueness of the challenge from CAS-2 prevents Requester from replaying authentications that it has seen earlier from other parties.

Proxy Certificates and ActingOnBehalfOf. On the Grid, Alice uses X.509 proxy certificates [19] to delegate rights to her jobs. Proxy certificates are similar to X.509 End Entity Certificates. The primary difference is that users like Alice issue proxy certificates to their jobs to create short-term (e.g., one-day) delegation of their rights. The short-term nature of proxy certificates allows them to be more lightly protected than the long-term EEC certificates. For example, proxy certificates are often protected with local filesystem permissions as opposed to being encrypted.

Alice can use Grid proxy certificates together with her own local definition of actingOnBehalfOf to control which rights her subjobs have. The wave tank access manager can refer to her definition, and ask the job to provide the proof:

```
"wave tank access manager":
actingOnBehalfOf(Proxy, User) ← actingOnBehalfOf(Proxy, User) @ User @ Proxy.
```

When a party starts a job on the Grid, it gives a signed delegation / proxy certificate to the new job. We represent this in PeerTrust as the literal authorizedJob(Job) signedBy [Party]. Alice has additional rules that say that the subjobs of her jobs are also acting on her behalf. Alice's job can prove to the wave tank access manager that it is acting on Alice's behalf by signing and presenting the following, for example:

```
actingOnBehalfOf(Proxy1, "Alice Smith") ← signedBy ["Alice Smith"]
  authorizedJob(Proxy1, ExpirationDate) @ "Alice Smith",
  currentDay(Today), Today < ExpirationDate.
actingOnBehalfOf(Proxy2, "Alice Smith") ← signedBy ["Alice Smith"]
  actingOnBehalfOf(Proxy1, "Alice Smith"),
  authorizedJob(Proxy2, ExpirationDate) @ Proxy1,
  currentDay(Today), Today < ExpirationDate.
authorizedJob("Alice's proxy","1/1/2005") signedBy ["Alice Smith"].
authorizedJob("Alice's second proxy","1/1/2005") signedBy ["Alice's proxy"].
authorizedJob("Alice's job","1/1/2005") signedBy ["Alice's second proxy"].
```

These signed statements allow the wave tank access manager to derive the fact that Alice Smith would say that Alice's job is acting on her behalf. As actingOnBehalfOf is a local predicate, CAS-1 and CAS-2 can use the definition above, or one of their own. Because Grid proxy certificates expire quickly, the expiration check is important. Checks of certificate validity dates are explicit in PeerTrust (rather than being built into the surrounding environment, like tampering and signature checks) because expired

certificates and certificates not yet valid can be quite useful. For example, to obtain a visa to many countries, one must show a passport that is valid for six months in the future. To argue that student loan repayment should have been deferred, one can show that one was a full-time student during the previous semester.

The example policies in this paper have not needed checks for certificate revocation. In PeerTrust, those checks must be described explicitly in the policy (just like expiration checks) because they are usually quite costly (a round-trip message to a revocation service) and whether they are worth the cost is an application-dependent decision. For example, a university will probably want to check whether a student ID has been revoked (e.g., because the student has withdrawn) before allowing the student to use an athletic facility. But a local hardware store that offers a discount to students for small purchases will not find a student ID revocation check worthwhile.

3.2 Alice Starts a Job Through a Portal

Alice has sent several of her long-term certificates to the NCSA MyProxy certificate repository [13] for safekeeping, and obtained a username and password there that will allow her to access MyProxy when she is away from her office. Now she connects to the internet at the airport, to check on the status of a job she submitted earlier, and sees that it has failed. She tweaks its parameters and is ready to resubmit it to the TeraGrid Linux cluster at SDSC. Her interface for the job submission is the public NPACI HotPage Grid Portal web page (https://hotpage.npaci.edu/), where she types in the name of her certificate repository (NCSA MyProxy) along with her user name and password at that MyProxy server, and the name she has given to her certificate (stored at MyProxy) that she wants her job to be able to use. The NPACI HotPage Portal authenticates itself to NCSA MyProxy, which is only willing to talk to certain clients (portals). Once authenticated, the portal asks MyProxy to issue a proxy credential that it can give to Alice's new job, so the job can prove that it is acting on Alice's behalf. Alice may have many identity credentials at MyProxy, and any of them can be used as the basis for issuing a new proxy credential. To indicate which one to use, Alice has typed in the credential name at the portal's web page. To obtain access to the proxy creation service, MyProxy requires the portal to submit the user name and password that Alice supplied earlier. NCSA MyProxy checks that the password is the right password for that user name, and if so, lets the portal access the proxy credential issuing service. With the new proxy credential in hand, the portal can submit Alice's job to the appropriate platform.

The previous paragraph describes how job submission through a portal works on the Grid today. Now we show how to encode the relevant access control policies in PeerTrust. The portal temporarily caches the information Alice typed in:

NPACI HotPage Portal (private):
requestedProxyServer(”Alice Smith”, ”NCSA MyProxy”).
requestedPassword(”Alice Smith”, ”Alice1234”).
requestedCredential(”Alice Smith”, ”Alice at NCA”).

NPACI HotPage Portal (public):

```
password(User, Password) $ MyProxy ←
  requestedProxyServer(User, MyProxy) @ User @ MyProxy |
  requestedPassword(User, Password).
member(”NPACI HotPage Portal”, ”NCSA MyProxy Permitted Portals”)
   signedBy [”CAS-1”].
```

The facts about Alice are private so that the portal can protect its users' privacy. For example, Alice's password will only be sent to the MyProxy server that she requested. The portal has cached certificates from CAS-1 regarding the portal's own group memberships, because the MyProxy servers will not talk to just any portal [20], and it cannot submit a Grid job without obtaining a proxy certificate for the job. In particular, the MyProxy server has the following policy for access to its proxy issuing service:

```
NCSA MyProxy:
accessProxyIssuingService(UserName, Password, CredentialName) $ Portal ←
  member(Portal, ”NCSA MyProxy Permitted Portals”) @ ”CAS-1” @ Portal,
  authenticatesTo(Portal, ”NCSA MyProxy”, ”NCA”) @ Portal |,
  correctPassword(UserName, Password).
```

Here, correctPassword is a local extensional private predicate. If access is granted, MyProxy's credential issuing service works with the portal to create a new proxy credential and associated key pair, which can be used to satisfy literals of the form 'authorizedJob(”NPACI HotPagePortal”) signedBy [”Alice Smith”]' in PeerTrust policies. The portal can use the new credential chain to submit the job via SDSC Gatekeeper on Alice's behalf.

4 Related Work

The Secure Dynamically Distributed Datalog (SD3) trust management system [9] 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 “Authority” argument, though this argument cannot be nested. “Requester” arguments are not possible either, which is appropriate for SD3's target application of DNS, but restricts SD3's expressiveness too much for our purposes. SD3 does not have the notion of guards, which are needed when the information in policies is to be kept private from certain parties. The newly proposed policy language Cassandra [2] combines many of the features of SD3 and RT [11], and is also close to PeerTrust.

Recent work in the context of the Semantic Web has focussed on how to describe security requirements, leading to the KAoS and Rei policy languages [10, 16]. 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 certificates during an iterative trust negotiation process that can be declaratively expressed and implemented based on guarded distributed logic programs.

Similar to the situated courteous logic programs of [6] 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 [6] 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.

Akenti [15] is a distributed policy-based authorization system designed for Grid environments using attribute certificates and delegated authorization. The Akenti model provides mechanisms for managing authorization policies from multiple stakeholders. After a requester authenticates to a resource, the Akenti policy engine gathers use-conditions for the resource and attribute certificates for the requester, then evaluates whether the requester meets the use-conditions. Akenti assumes attribute certificates are publicly available and does not provide the privacy features of an interative trust negotiation process as presented in this paper.

As discussed earlier, authorization in GSI is commonly based on identity credentials and access control lists (grid-mapfiles). However, recent work applies authorization credentials and authorization policy management techniques to Grid computing. The Community Authorization Service (CAS) [14], Virtual Organization Membership Service (VOMS) [1], and Privilege Management and Authorization (PRIMA) system [12] each embed authorization credentials in GSI proxy credentials, to enable transport of authorization credentials in existing GSI authentication protocols. CAS embeds signed SAML policy assertions in GSI proxy credentials, encoding specific rights such as read and/or write access to files served by a GridFTP server. VOMS embeds attribute certificates in GSI proxy credentials that specify group and virtual organization membership for access to community resources. PRIMA embeds XACML privilege statements in GSI proxy credentials that are compared with XACML resource policies for access control. The trust negotiation techniques described in this paper could be applied to these systems to avoid unnecessary disclosure of attributes and authorization credentials as well as providing mechanisms for gathering the necessary credentials to obtain a requested authorization.

5 Conclusion and Further Work

In this paper, we have used examples to show how to handle common Grid authentication and authorization scenarios using trust negotiation with the PeerTrust policy language. For readers interested in experimenting with PeerTrust, the PeerTrust 1.0 prototype is freely available at http://www.learninglab.de/english/projects/peertrust.html. Like the earlier prototype described in [5], PeerTrust 1.0's outer layer is a signed Java application or applet program, which keeps queues of propositions that are in the process of being proved, parses incoming queries, translates them to the PeerTrust language, and passes them to the inner layer. Its inner layer answers queries by reasoning about PeerTrust policy rules and certificates using Prolog metainterpreters (in MINERVA Prolog, whose Java implementation offers excellent portability), and returns the answers to the outer layer. PeerTrust 1.0 imports RDF metadata to represent policies for access to resources, and uses X.509 certificates and the Java Cryptography Architecture

for signatures. It employs secure socket connections between negotiating parties, and its facilities for communication and access to security related libraries are in Java.

In this paper, we have not had room to discuss PeerTrust's semantics, which are an extension of those of SD3. In PeerTrust, in general there will be many facts that are true but that a party is unable to derive at run time. Further, the facts that a party is able to derive may depend on its previous interactions with its peers. PeerTrust's distinction between truth and run-time derivability opens up many interesting research questions for future work. One would like to be able to analyze a PeerTrust program's safety and availability properties: for a particular resource, which users will be able to find proofs to convince the resource manager to let them access it, and under what conditions? Which classes of users will never be able to access that resource? These classic analysis questions have recently been shown to be tractable for RT and may also be tractable for the more complex searches for proofs that are possible in PeerTrust. Other interesting issues include how to control the evolution of a distributed proof process, including termination guarantees, deadlock, and livelock; support for additional kinds of query answers, for negation, negative authorizations, and conflicting beliefs; and additional support for ease of management of multiple identities. Distributed query processing techniques will be useful in helping parties satisfy policies as efficiently as possible (e.g., it is unwise for Alice to try to prove that she is authorized to access a cluster by trying to get a list of all authorized users and then checking to see if she is on it). The performance of TrustBuilder+PeerTrust in simple Grid trust negotiations is also an interesting question for future work; we anticipate that the additional overhead in GSI at job startup will be dwarfed by the very high cost of key pair generation and XML argument marshalling/demarshalling at startup (approximately 2 seconds on today's platforms).

Acknowledgements We thank the reviewers for their many helpful comments and questions, not all of which we had room to answer in the final version of this paper. Winslett's research was supported by DARPA (N66001-01-1-8908), the National Science Foundation (CCR-0325951, IIS-0331707) and The Regents of the University of California. The research of Nejdl and Olmedilla was partially supported by the projects ELENA (http://www.elena-project.org, IST-2001-37264) and REWERSE (http://rewerse.net, IST-506779). Work on MyProxy is supported by the NSF NMI program, and research on CAS is funded by the Department of Energy (W-31-109-ENG-38).

References

1. R. Alfieri, R. Cecchini, V. Ciaschini, L. dell'Agnello, A. Frohner, A. Gianoli, K. Lőrentey, and F. Spataro. Voms: An authorization system for virtual organizations. In *Proceedings of the 1st European Across Grids Conference*, Santiago de Compostela, Feb. 2003.
2. M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *Policies in Distributed Systems and Networks*, June 2004.
3. P. A. Bonatti and P. Samarati. A uniform framework for regulating service access and information release on the web. *Journal of Computer Security*, 10(3):241–272, 2002.

4. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.
5. 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 *European Semantic Web Symposium*, Heraklion, Greece, May 2004.
6. B. Grosof and T. Poon. SweetDeal: Representing agent contracts with exceptions using XML rules, ontologies, and process descriptions. In *WWW12*, 2003.
7. A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith. Advanced client/server authentication in TLS. In *NDSS*, San Diego, Feb. 2002.
8. A. Hess and K. E. Seamons. An access control model for dynamic client content. In *8th ACM Symposium on Access Control Models and Technologies*, Como, Italy, June 2003.
9. T. Jim. SD3: A Trust Management System With Certified Evaluation. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001.
10. L. Kagal, T. Finin, and A. Joshi. A policy based approach to security for the semantic web. In *International Semantic Web Conference*, Sanibel Island, Oct. 2003.
11. N. Li and J. Mitchell. RT: A Role-based Trust-management Framework. In *DARPA Information Survivability Conference and Exposition (DISCEX)*, Washington, D.C., Apr. 2003.
12. M. Lorch, D. Adams, D. Kafura, M. Koneni, A. Rathi, and S. Shah. The PRIMA system for privilege management, authorization and enforcement in grid environments. In *4th Int. Workshop on Grid Computing - Grid 2003*, Phoenix, AZ, USA, Nov. 2003.
13. J. Novotny, S. Tuecke, and V. Welch. An online credential repository for the grid: MyProxy. In *Symposium on High Performance Distributed Computing*, San Francisco, Aug. 2001.
14. L. Pearlman, C. Kesselman, V. Welch, I. Foster, and S. Tuecke. The community authorization service: Status and future. In *Proceedings of the Conference for Computing in High Energy and Nuclear Physics*, La Jolla, California, USA, Mar. 2003.
15. M. Thompson, W. Johnson, S. Mudumbai, G. Hoo, and K. Hackson. Certificate-based access control for widely distributed resources. In *Usenix Security Symposium*, Aug. 1999.
16. 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 International Semantic Web Conference*, Sanibel Island, Oct. 2003.
17. J. Trevor and D. Suciu. Dynamically distributed query evaluation. In *PODS*, 2001.
18. S. Tuecke, V. Welch, D. Engert, L. Pearlman, and M. Thompson. *Internet X.509 Public Key Infrastructure Proxy Certificate Profile*. http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-10.txt, Dec. 2003.
19. V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist. X.509 proxy certificates for dynamic delegation. In *3rd Annual PKI R&D Workshop*, Apr. 2004.
20. V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *HPDC*, 2003.