Policy-driven Negotiation for Authorization in the Grid

Ionut Constandache,* Daniel Olmedilla,* Frank Siebenlist,† and Wolfgang Nejdl,*
*L3S Research Center and University of Hanover, Germany
{constandache,olmedilla,nejdl}@l3s.de
†Argonne National Laboratory, USA
franks@mcs.anl.gov

***Abstract*— In many Grid services deployments, the clients and servers reside in different administrative domains. Hence, there is a requirement both to discover each other's authorization policy, in order to be able to present the right assertions that allow access, and to reveal as little as possible of the access policy details to unauthorized parties. This paper describes a mechanism where the client and servers are semantically annotated with policies that protect their resources. These annotations specify both constraints and capabilities that are used during a negotiation to reason about and communicate the need to see certain credentials from the other party and to determine whether requested credentials can be obtained and revealed. The result of the negotiation is a state where both parties have satisfied their policy constraints for a subsequent interaction or where such interaction is disallowed by either or both. Furthermore, we present an implementation of a prototype, based on the PEERTRUST policy language and its reasoning engine, that is integrated in the Web services runtime component of the Globus Toolkit. The negotiation process is facilitated through the implementation of WSRF-compliant service interfaces for protocol message exchanges.**

I. INTRODUCTION

Organizations join in collaborations for the benefit of sharing resources for data retrieval, job execution, monitoring, and data storage. Each organization is defined by its own administrative domain, while the overlaying virtual organization (VO) is defined by the collaboration agreement. Grid toolkits provide the middleware for the applications such that the shared resources residing in the different domains can be securely accessed. Such an environment provides users with seamless access to all resources they are authorized to. In current Grid infrastructures, in order to be granted access at each domain, users' jobs have to secure and provide appropriate digital credentials for authentication and authorization. While authentication along with single sign-on can be provided based on client delegation of X.509 Proxy Certificates to the job being submitted, most authorization mechanisms are still identity based. Because of the potentially large number of users and different certification authorities, this situation leads to scalability problems.

To address these problems, we exploit the annotation of Grid services with machine-understandable languages in the context of authentication and authorization by binding rule-based policies to resources. These policies specify access control requirements that must be satisfied by a requester before access is authorized. In addition, we claim that authorization cannot be based on parties' identities alone because such authorization does not scale and hence is not a one-time mechanism but an iterative process, in which the entities involved must be able to negotiate and incrementally increase their level of trust based on other party's properties. The application of negotiation-based authorization mechanisms to Grid environments overcomes current Grid authorization limitations. Policy-driven negotiations provide scalability, advanced access control, and automatic credential fetching. These features enable a complete set of new scenarios in the context of authorization in which user involvement is dramatically reduced (from administrative and development points of view) in favor of automated interactions.

This paper identifies current limitations in Grid authorization, suggests the application of policy languages to protect access to resources, and demonstrates the power and advantages of this approach (based on our preliminary work presented in [1], [2]). In addition, we propose an architecture in which policy languages can be easily integrated into the Globus Toolkit 4.0, allowing advanced access control and automatic credential fetching. We also describe how this architecture has been implemented by using the semantic policy language PEERTRUST for policy specification.

II. GRID-SPECIFIC AUTHENTICATION AND AUTHORIZATION

The Globus Toolkit [3] offers a collection of software that implements different protocols, specifications, standards, and interfaces to meet the requirements identified by the Open Grid Services Architecture (OGSA) [4] to support a Grid infrastructure. Since its inception in the late 1990s, the Globus Toolkit has addressed issues such as resource discovery, remote execution monitoring and management, data movement and replication, and security in service-oriented distributed environments. In this section we provide a short introduction to the Globus Toolkit 4.0 (GT4.0), focusing on its integrated Grid Security Infrastructure (GSI) [5].

GSI provides integrity protection, confidentiality, and authentication for sensitive information passed over the network, as well as the facilities to securely traverse the different organizations that are part of a collaboration. The fundamental security mechanism used by the GSI infrastructure is based on

public key cryptography and the associated PKIX X.509-based Public Key Infrastructure (PKI) [6]. GSI relies on an X.509 identity certificate to bind a public key to a unique name (called Distinguished Name, or DN) identifying the private key holder. A trusted certification authority (CA) guarantees through its signing policy that the two belong together and attests to this by signing the certificate.

GSI supports a delegation mechanism through the use of X.509 Proxy Certificates [7]. The user can delegate his rights to agents, intermediaries, or resources such that they can interact with other resources on the user's behalf. The mechanism consisits of the agent generating a new key pair and the user signing and issuing a proxy certificate that holds the agent's public key. A proxy certificate can be signed by an end entity certificate or by another proxy certificate. During authentication the whole chain of (proxy) certificates is exchanged between the parties, allowing them to obtain the user identity information from the end entity certificate at the root of the chain.[1]

GT4.0 provides an authorization framework for enforcing authorization on both client and service side. On the client side, authorization mechanisms rely on a configured chain of policy decision points (PDPs) to determine whether authorization should be granted or denied for a client invocation. The decision regarding authorization is made based on the conjunction of all PDP decisions; that is, authorization is granted only if all PDPs have returned a permit decision. The PDP's decision logic can be implemented based on the client's DN, the resource accessed, and the operation invoked. On the client side, service authorization options are `self` (identity of the service and client are expected to be the same), `host` (a certain host name is expected of the service accessed), or `identity` (a certain identity of the service is expected where identity refers to the DN present in service certificate).

On the service side, a number of PDP implementations for different authorization options are part of the GT4.0 distribution. These PDPs are configured either at the service level or for the entire GT4.0 container running the service. The options include `self`, `host`, and `identity` as well as `gridmap` (the identity of the client must be mapped to a local user account on the resource in a grid-mapfile), `samlCallout` (a SAML authorization callout to an external OGSA Authorization compliant service), and `userName` (username- and password-based authorization).

III. Example Scenario: Interaction between Jobs and Services in GT4.0

Let us consider the following illustrative scenario depicted in Figure 1. A group of scientists at the Computer Science and Engineering Department of University "Politehnica" of Bucharest (UPB) would like to obtain data regarding oceanic water waves in order to develop signaling instruments for tsunami hazards avoidance. Fortunately, the university and the Navy Institute have an agreement that allows the UPB members to use any of the institute's scientific instruments as long as they are not already in use and UPB has not exceeded its maximum number of monthly allocated hours for the use of institute's resources. Both the university and the Navy Institute support the Globus Toolkit 4.0, which provides the common interoperable middleware layer the scientists need to collect the desired data from a wave tank available at the institute site.

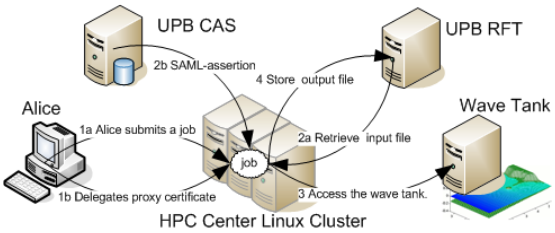

Fig. 1. Simple Grid Scenario

Alice's client program submits a job over the Grid infrastructure to the UPB High-Performance Computing Center Linux cluster. There the job will retrieve the input data needed to setup the wave tank, perform CPU-intensive computations to refine the output data of the experiment, and store the results. First, however, the Linux cluster must authenticate and authorize the submitted job. Alice's client program authenticates using Alice's X.509 End Entity Certificate signed by the UPB Certification Authority trusted at the Linux cluster site. Authorization is also granted as a grid-mapfile entry maps Alice's DN to a local user account permitting the job to start its work with the privileges of the associated local user.

Since the job needs additional resources, a new X509 Proxy Certificate is delegated by Alice to the UPB Linux cluster. Using this additional proxy certificate, the job can authenticate and gain access to other Grid resources. First it contacts the UPB Reliable File Transfer Service (RFT) [8] to retrieve the input data. Authentication and authorization are satisfied based on Alice's DN at the root of the certificate chain.

Alice has already been informed that the Navy Institute requires a proof of her university affiliation, so she has instructed the job to contact the UPB CAS Server and retrieve a statement attesting to her involvement with the university. After the required authentication and identity-based authorization, the job obtains a SAML assertion attesting to Alice's university affiliation. The job-processing application subsequently creates a new X.509 Proxy Certificate embedding the previous obtained assertion. This proxy certificate ensures access at the Navy Institute wave tank site.[2]

When the data has been corrected and refined, the Linux cluster contacts the UPB RFT service; and after another round of authentication and authorization verification, the final data is made available to Alice and her group at the university file server.

[1]The user's identity information is authenticated if that user's end entity certificate is signed by a trusted CA.

[2]We assume that the wave tank is not in use and UPB has not exceeded its allocated hours.

IV. Current Limitations and Assumptions

During the interactions depicted in the previous scenario, authorization decisions were made based on user identity. The user already had an account at the remote locations with a grid-mapfile entry mapping his X.509 certificate's subject's DN to his local account. Another assumption was that the UPB CA was trusted at each location the job needed access to (UPB resources and Navy Institute wave tank). There was no intermediary step for the job and the contacted resources to argue about trusted authorities. The trusted CAs of each site were expected to be known by the client, who was expected to provide an appropriate certificate. In these conditions, mapping client identities to local accounts raises serious scalability problems because of the large number of potential users. Even more unrealistic is the requirement of having a single trusted Certification Authority when the Grid spans multiple organizations each having its own authentication infrastructure.

Managing identity-based access control lists requires site administrators to keep track of all possible clients allowed to request services. This is a difficult task because the user mapping entries are the result of the user's associations with certain organizations and projects. Once such relations cease to exist, the resource administrators have to remove all these entries to ensure that access is no longer allowed. On the other hand, as soon as new users are required to deploy the Grid infrastructure, new mapping entries have to be entered in all relevant access lists.

Keeping track of all the different credentials, in relation to the resource and the domain where they are needed, is a burden on the client side as well, because of the difficulty in predicting the job's needs prior to its execution. When the job is expected to present a required and missing credential, the user may not be available. Alternatively, providing the job with all the user's credentials might imply disclosing sensitive information, a situation that the user may not be comfortable with. The issue for the user or job is to provide just the right credentials to the Grid service it tries to access—no more, no less.

In all the previously described interactions, the authorization decision was a one-step process, without the client asking about the resource access requirements and without any negotiation for the granting of authorization. The client was simply expected to be aware of and be able to satisfy the service requirements. This approach seems infeasible for large-scale Grid deployments, where each of the computational resources may be under a different authority and where entities with no previous interactions may have to communicate.

Some of the deployed Grids base their authorization on assertions issued by infrastructure services established at the virtual organization level (e.g., CAS). Grid service providers may feel uncomfortable relying only on remote assertions, especially in the case of a large Grid environment. As a result, some service providers may use, publish, and enforce policies in the local domain such that the administrators keep fine-grained access control over the resource they provide.[3]

We argue in the following sections that when Grid deployments support the specification, advertisement, and enforcement of service-level access control policies, together with capabilities for the automatic fetching of credentials, large-scale collection of resources can indeed be supported. Such a mechanism will also enable dynamic negotiation for authorization and access granting based on the properties of both parties.

V. Rule-based Policy Languages

Rule-based policy languages have been used extensively in the context of security policies, trust management, and business rules [9]. The reason researchers have typically selected rule-based languages to express policies is not arbitrary. Their semantics is closer to the way humans think and is especially useful for access control protection where one specifies *what* conditions are to be fulfilled by the requester, without specifying *how*. Furthermore, rule-based policies provide self-described statements that can be exchanged, shared, and reused among parties and that allow one to reason over them, hence enabling interoperability.

Security in distributed environments such as the World Wide Web, peer-to-peer systems, or Grids was traditionally built under the assumption that service providers and consumers are known to each other. In common scenarios, before allowing access to (possibly) sensitive resources, trust relations are established by having clients create an account, store a profile, or requiring administrators to add identities to some kind of access control lists.

Alternatively, through semantic annotations, rule-oriented access control policies, and trust negotiation [10], entities can automatically build a trust relationship, through which they will feel comfortable sharing selected private information. The trust negotiation process evolves through disclosure of credentials and requests for credentials in an iterative and bilateral process. The requests for credentials are in fact authorization policies that have to be satisfied. The trust level reached through a negotiation between one client and a resource is in fact an authorization decision about whether to allow access to the resource. This approach distinguishes itself from identity-based access control because trust is established based on properties of previously unknown parties .

Trust negotiation is triggered when one party requests access to a resource owned by another party. The goal of a trust negotiation is to find a sequence of credentials $C_1, \cdots, C_k, R$, where R is the resource where access is attempted, 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.

[3]Even if authorization statements are made by third parties at the VO level, the authorization decision may involve local information and state (e.g., the wave tank should not be in use, and the UPB use of instruments should not have exceeded a certain limit).

The PEERTRUST language [11], [12] for expressing access control policies is based on definite Horn clauses (the basis for logic programs [13]), namely, 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 (check [11], [12] for more information) and 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, suppose that an online library requires a student id, issued by University of Hanover, before it allows access to the available online books. One can think of this as a case of the library *delegating evaluation* of the query "Is the requester a student?" to the University of Hanover (UniHann). To express delegation of evaluation to another party, we extend each literal lit_i with an additional *Issuer* argument,

lit_i @ Issuer

where *Issuer* specifies the party who is responsible for evaluating lit_i or has the authority to evaluate lit_i. For example, the library's policy may be expressed as

```
Library:
allowAccess(Book,X) ←
    student(X) @ 'UniHann'.
```

where X represents the requester whose quality of student should be evaluated by 'UniHann'. 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, it is unlikely that UniHann will answer all the requests directly from the library, so a more practical approach is to ask the requester to evaluate the query himself, that is, to disclose his student id:

```
Library:
allowAccess(Book,X) ←
    student(X) @ 'UniHann' @ X.
```

The library 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 @ Issuer $ Requester

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. Therefore, the final shape of the rule of the online library would look like the following:

```
Library:
allowAccess(Book) $ Req ←
    student(Req) @ 'UniHann' @ Req.
```

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 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 (credentials), and it may use these rules in its proofs in certain situations. For example, Alice can use a rule that was defined by UniHann to prove that she is really a student:

```
Alice:
student('Alice') @ 'UniHann'
    signedBy ['UniHann'].
```

In this example, the "signedBy" term indicates that the rule has UniHann's digital signature on it. A signed rule has an additional argument that says who issued the rule.

Guards: To guarantee a certain evaluation order for the literals in the body of a rule, 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. For example, if the library offers a special price for students interested in a paperback copy of a book, the policy could be expressed as

```
Library:
order(Book) $ Req ←
    student(Req) @ Institution @ Requester |
    applyDiscount(Book).
```

The requester would have first to prove he is a student. Then the library would apply a discount for the requested book.

VI. DISTRIBUTED POLICY-BASED NEGOTIATION FOR GRID SERVICES AUTHORIZATION

This section addresses some of the current Grid Security Infrastructure limitations by describing how PEERTRUST policies[4] can be integrated to accommodate a large, loosely coupled Grid environment.

In our initial example (see Figure 1) we illustrated how different resources may be used in a Grid environment based on the Globus Toolkit 4.0. The fact that resources are often expensive in terms of cost, maintenance, and management normally results in more complex access policies than the already established conditions of two previously acquainted partners (in our example, UPB and Navy Institute). Jobs may use discovery and scheduling services to locate the best available resources, which make it even more difficult to predict what service instances will be used.

Our solution proposes, on one hand, that each Grid service advertises its authorization requirements through access control policies and, on the other hand, that each client specifies his credential disclosure policies and is able to query for resource's access policies. When such policies are advertised, services and clients can negotiate, increasing incrementally their trust relationship based on satisfying the other party's

[4]Although we use the PEERTRUST language for policy specification in our examples and implementation, all the ideas presented in this paper are extensible to any other policy language with *delegation of authority* and *negotiation capabilities* [14].

policies. We will change our initial scenario in order to accommodate the new capabilities highlighted above.

Alice's research group needs the data as soon as possible, so Alice's client program queries the National Monitoring and Discovery Hierarchical Service (NMDHS) for the best Linux cluster in terms of memory size and number of available processors. Alice program will send a request of the form `queryingAllowed()` to the NMDHS. The service policy requires any client to have a credential signed by one of the recognized state institutions (check the `id` policy below), so Alice's client provides Alice X.509 End Entity Certificate issued by the UPB CA. Since the credential is not protected by any policy, Alice's client program discloses it, authorization is fulfilled, and the requested information is delivered.

NMDHS:
queryingAllowed() $ Req ←
    validCredential(Req).
validCredential(Req) ←
    id(Req,'UPB CA') @ 'UPB CA' @ Req.
validCredential(Req) ←
    id(Req,'Navy Ins. CA') @ 'Navy Ins. CA' @ Req.
...

The best available Linux cluster belongs to the Research Center for Aeronautical Sciences (RCAS), so Alice's client program initiates a new negotiation process for submitting Alice's job. The RCAS Linux cluster policy restricts access to jobs acting on behalf of members of projects listed in the Ministry of Education (MinEdu) database.

RCAS Cluster:
submit(Job) $ Req ←
    actingOnBehalfOf(User,Job),
    member(User,Project) @ 'MinEdu CAS' @ Req.

Alice's client program demonstrates that the job is acting on Alice's behalf by providing Alice's certificate, but it does not yet have a credential proving that Alice is a member of a listed project. To solve this situation, Alice's client forwards a query to the Ministry of Education CAS server (MinEdu CAS) and retrieves an assertion attesting that Alice participates in a project regarding signaling instrumentation. This credential is disclosed to the RCAS Linux cluster, and job submission is granted. Since the job needs to contact other resources, a proxy credential gets delegated on behalf of Alice, to the RCAS Linux Cluster.

When Alice's job starts its execution, it has to retrieve the input data from the UPB Reliable File Transfer Service (UPB RFT). UPB RFT policies require the job to prove that it is acting on behalf of a UPB staff member.

UPB RFT Service:
retrieve(File) $ Req ←
    member(Req,'Staff') @ 'UPB CAS' @ Req |
    check(Rights,Req,File) @ 'UPB CAS'.
store(File) $ Req ←
    member(Req,'Staff') @ 'UPB CAS' @ Req.

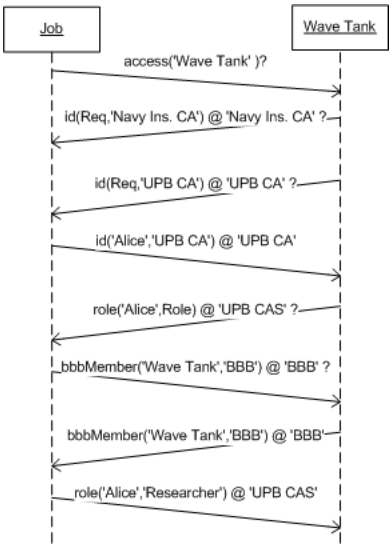

Fig. 2. Job Negotiations

Therefore, using its delegated credential, the job authenticates to the UPB CAS, demonstrates it is acting on Alice's behalf, and retrieves a credential attesting that Alice is part of the university staff. Once this credential is released to the UPB RFT service, the service checks by itself, with the same UPB CAS if the required file can be accessed by Alice. UPB RFT acquires an assertion identifying the input file as belonging to Alice, and in this way the file retrieval operation is allowed.

Now the job contacts the Navy Institute wave tank (with a query `access('Wave Tank'))` and discovers that it has to provide a certificate signed by either UPB CA or Navy Institute CA (see Figure 2). By disclosing Alice's delegated certificate, the job demonstrates it is acting on behalf of an entity certified by UPB CA. Further, the wave tank informs Alice's job that it should reveal Alice's role at the university.

Wave Tank:
access(Resource) $ Req ←
    checkUser(Req,CA), |
    notInUse(Resource),
    timeLimitNotExceeded(CA).
checkUser(Req,'Navy Ins. CA') ←
    id(Req,'Navy Ins. CA') @ 'Navy Ins. CA' @ Req.
checkUser(Req,'UPB CA') ←
    id(Req,'UPB CA') @ 'UPB CA' @ Req, |
    role(Req,Role) @ 'UPB CAS' @ Req |
    Role = 'Researcher'.
bbbMember('Wave Tank','BBB') @ 'BBB'
    signedBy ['BBB'].

In this case, Alice had provided a policy to the job, stating that Alice's roles are revealed only to entities that have proved their liability by disclosing a Better Business Bureau (BBB) membership credential.

Job:
role('Alice',Role) @ 'UPB CAS' $ Req ←

member(Req,'BBB') @ 'BBB'.

The wave tank has no policy protecting its BBB credential; therefore, after its disclosure, the job contacts UPB CAS to retrieve Alice's roles. Since Alice is a researcher and assuming that the other local policies are fulfilled, the job is granted access to the wave tank.

The refined and corrected data generated by the Linux cluster is saved by using the UPB RFT Service, which allows file storage if the action is requested by a university staff member (see the RFT policy above). The job has cached this credential (during a previous interaction) and does not need to ask for it again from the UPB CAS. By revealing this proof of Alice's university staff membership, the job is allowed to use the UPB RFT for storing its output files.

This scenario shows that the job has been submitted with only one credential and dynamically negotiated authorization at each resource accessed. This negotiation involved learning from resources themselves which credentials are needed (specified in policies) and where to retrieve them from. Resources were shared with no implied previous interactions and with no further involvement of administrators other than to set policies for access control and credential disclosure.

VII. Architecture and Implementation

One of our main design goals was direct integration with Grid services paradigms. To achieve this goal, we devised an extension that is easily pluggable into any GT4.0-compliant Grid service or client. In this section, we present the technical details of our implementation (see also Figure 3). Because this implementation was developed for the GT4.0 Java Container, it is entirely written in Java and therefore requires Java-based Grid services and clients for a straightforward integration.

As we have seen in our discussions regarding the PeerTrust capabilities, a service might allow different possibilities for granting access, and the client may choose which one to follow, according to his own credential protection policies.[5] Because of the GT4.0 limitation in the PDPs decision composition (no disjunction supported), we have developed an Interceptor PDP, which is part of the *Client Call Interceptor*(see Figure 3) and is responsible only for protection against unauthorized calls. We have used the PeerTrust language and implemented its logic outside this custom PDP to permit policy combinations as both conjunctions and disjunctions. Our custom PDP intercepts client operation invocations and allows or denies the calls according to the existence of a previous successfully completed negotiation. The Interceptor PDP checks at the *Negotiation Module* whether a successful negotiation for the operation requested has been previously completed. If the answer is positive, the PDP grants the Grid service operation invocation. If the answer is negative, the PDP throws a negotiation exception to inform the client that a trust negotiation must take place that should be completed successfully before access is granted. In the latter case, the client Negotiation Module must start a negotiation with the homologous module on the Grid service (with a query of the form `request('OperationName')`). This triggers the negotiation process, in which policies and credentials are exchanged. The entire negotiation process is automated and requires no explicit user intervention.[6]

The *Negotiation Module* is responsible for the negotiation management. It handles the communication with other parties using *Grid connectors* (implemented with the Globus Toolkit 4.0) and interfaces with the PeerTrust *Module* (described later in this section). Whenever a negotiation succeeds, the Negotiation Module caches the identity of the client and the operation granted, in order to avoid the same negotiation in (presumably) successive calls. Ideally, authorization should be invalidated once the shortest-lived credential (between those disclosed) has expired; but since we plan to experiment with a wide variety of credential formats, each requiring an independent parsing, the current implementation relies only on the expiration time of the client proxy certificate (the one used by the client for authentication).

On the service side, the Grid Connectors are composed of two modules: the *Negotiation Provider* and the *Negotiation Topic*. These modules are plugged into the Grid service, enhancing its functionality with policy-based negotiation capabilities. The client can use the additional Grid service's operations, implemented in the *Negotiation Provider Module*, for pushing policy requirements and credentials to the server side.

Negotiations are essentially asynchronous. A request might, for instance, trigger a new policy negotiation with a third party and so on, a situation that makes it difficult to predict when the request will finish its evaluation. Moreover, a client might want to continue its usual execution without having to wait for the answer from the server (e.g., in order to start in parallel several negotiations with different entities) and might simply want to be informed asynchronously when each negotiation is finished.[7]

Asynchronous client notifications can be implemented through GT 4.0 support for WS-Base Notification [15] and WS-Topics [16], which standardize asynchronous communications between consumers and providers. We associate each negotiation (triggered by the client request for a service operation) with a topic of interest (*Negotiation Topic*) through which messages can be delivered to the client side. The client subscribes to this topic via a subscribe operation exposed by the notification producer (in our case the Grid service). On the client side, the notification consumer (in Figure 3 the *Notification Listener*) exposes a notify function that the Grid service uses to inform the client about its own requirements in terms of authorization.

A Grid service describes its supported operations and their

[5]For instance, the client might choose the one where less sensitive information is revealed or might choose a service that is locally available.

[6]Although we refer most of the time to fully automated negotiations, it would of course be possible to monitor and require confirmation from the user for all or some steps of the negotiation.

[7]Note that a server might act as a client if it starts a new negotiation with a third party.

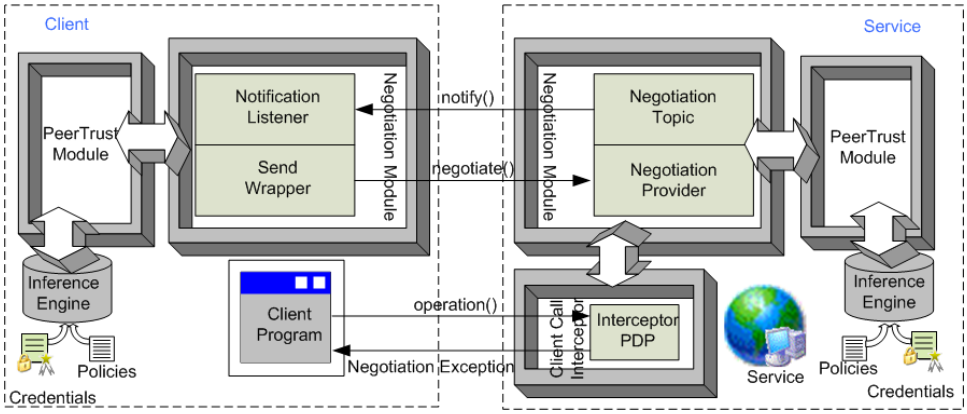

Fig. 3. High-Level Architecture

associated parameters through a so-called WSDL file. We have defined a special WSDL file that describes all the interfaces needed to perform our negotiations. Any Grid service can use our negotiation capabilities by including our definitions in its WSDL description. Doing so allows a Grid service to expose additional operations that are used to push messages to the Grid service during the trust negotiation process. These two operations are implemented externally to the application-specific Grid service code in our Negotiation Provider Module. This module is plugged into a Grid service through the specification of a "provider" (our defined TrustNegotiation-Provider) in the Grid service deployment descriptor (WSDD file). Providers are used to confer extra functionalities to Grid services. GT4.0 support for notifications is also plugged into a Grid service by means of two providers (delivered with GT4.0) configured in the same deployment descriptor. We emphasize that in order to confer the negotiation capabilities to any Grid service we require small modifications to its descriptors (configuration) files, but no modification to the application code itself.

For the client side, we have developed an API with a jar file to facilitate easy integration of trust negotiation capabilities with the client code. Through the API, the client needs to set the Grid service address and must implement an interface(*Send Wrapper*) to communicate with the service with which trust is negotiated. The client can also use a provided class for the automatic fetching of a SAML assertion from a Community Authorization Server.

So far we have described how the policy-based negotiations can be integrated into GT4.0, but we have not described how the reasoning is performed. Queries for enforcement of policies are sent to the PEERTRUST *Module*, which controls the access to the *Inference Engine*. The *Inference Engine* is built on the Minerva Prolog engine[8] and reasons on policies and credentials configured locally or retrieved during the negotiations undertaken. The Inference Engine answers to the PEERTRUST module queries indicating whether a request conforms to local policies (therefore allowing or disallowing the disclosure of credentials requested) or whether extra conditions must first be satisfied by the requester. The evaluation of a query allows one to include a proof in any answer returned to other parties. The proof contains the policies and credentials (possibly third-party credentials) required to attest the other party's request.

Because our implementation uses an abstract representation of credentials and policies, it can use different kinds of formats (certificates, signed assertions, signed RDF statements, etc). Currently we support X.509 v3 certificates, which can carry holder attributes in the extensions. In addition, we have integrated the use of proxy certificates that carry SAML assertions retrieved from a Community Authorization Service.

VIII. CONCLUSIONS, AND RELATED AND FURTHER WORK

Our main contribution to the Grid authorization solution space is the presented negotiation process in which entities, distributed over the Grid, engage each other for access granting. During this process, entities express their access requirements and enumerate the required credentials with their associated third-party issuers. Since credentials are retrieved dynamically during the negotiations, the access policies fulfillment is established at runtime. Furthermore, services and clients are true peers in the negotiation, since both are able to protect their local resources (e.g., credentials and provided services) and to formulate requests for the required credentials from the other party.

One of the current trends in Grid deployment is to move from identity-based to attribute-based authorization or role-based access control (RBAC). The main reasons for this shift are the scalability issues associated with identity-based policies and the ease of administration through abstraction mechanisms as groups and roles. Projects such as VOMS [17], PERMIS [18], PRIMA [19], and XPOLA [20] have proposed solutions for attribute-based access control. GridShib [21], a recently started project with the goal of integrating Shibboleth [22] and the Globus Toolkit, aims to transparently retrieve attribute assertions from a Shibboleth service for the requester. The Community Authorization Service set at the VO level

[8]Minerva Prolog (`http://www.ifcomputer.com/MINERVA/`) provides a java-based prolog engine that allows for an easy integration.

is an authorization service where the client can ask for the access rights needed to invoke a request at a remote service. Despite the increased scalability, however, all these authorization solutions rely on fixed configurations of trust roots, on expected knowledge of required credentials, and on expected knowledge from which such credentials can be retrieved. Their deployment is therefore inflexible, and policy changes are expensive. In addition, most authorization schemes are server policy centered, and no policy is enforced on the service itself by the client.

It is encouraging to see that some of the requirements for policy advertisement and capability matching that we have identified and addressed in our negotiation framework are recognized in emerging standards and specifications such as XACML [23], SAML [24], WS-Agreement [25], and WS-Policy [26]. We are following their development closely and hope to leverage in our own negotiation protocols work some of the policy primitives that may result from these efforts.

In the near future, we plan more experiments to study the performance cost induced by our policy negotiation approach. Although we know in advance that our negotiation requires more round-trips and is therefore more expensive than conventional approaches, it will be a trade-off with the time and efforts needed for hard-coded configurations and for out-of-band policy information exchange. We will also investigate the most simple scenario where only a two-step negotiation would be required to query resource access policies and their subsequent fulfillment, which would give us a lower-limit performance mark. In addition, declarative policies may create loops when its distributed evaluation take place; we plan to integrate existing techniques for loop detection [27] and the avoidance of deadlocks [28].

A further area of research is the possible utilization of our traceable negotiation process for accounting purposes, such as billing and audit. Our infrastructure already supports the recording of the negotiation steps at each resource involved. The iterative process of requests for and disclosures of credentials may be extended to include the negotiation for pricing and exchange of payment until an agreement is reached.

Lastly, we plan the integration of a more expressive language than PEERTRUST (e.g., PROTUNE [29]) into our framework to allow, for example, the execution of arbitrary operations when certain policies are satisfied (e.g., logging information).

ACKNOWLEDGMENTS

This work was partially funded by the European Commission and by the Swiss State Secretariat for Education and Research within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`), and by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract W-31-109-Eng-38.

REFERENCES

[1] J. Basney, W. Nejdl, D. Olmedilla, V. Welch, and M. Winslett, "Negotiating Trust on the Grid," in *2nd WWW Workshop on Semantics in P2P and Grid Computing*, New York, May 2004.

[2] I. Constandache, W. Nejdl, and D. Olmedilla, "Policy Based Dynamic Negotiation for Grid Services Authorization," in *Proceedings of the Semantic Web and Policy Workshop*, November 2005, pp. 55–67.

[3] "Globus Toolkit," http://www.globus.org.

[4] "The Open Grid Services Architecture OGSA, version 1.0," Global Grid Forum.

[5] "Grid Security Infrastructure," http://www.globus.org/security/overview.html.

[6] R. Housley, T. Polk, W. Ford, and D. Solo, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, RFC 3280, 2002.

[7] 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.

[8] W. Allcock, I. Foster, and R. Madduri, "Reliable Data Transport: A Critical Service for the Grid," in *Building Service Based Grids Workshop, Global Grid Forum 11*, 2004.

[9] G. Antoniou, M. Baldoni, P. A. Bonatti, W. Nejdl, and D. Olmedilla, "Rule-based Policy Specification," in *Decentralized Data Management Security*, T. Yu and S. Jajodia, Eds. Springer, 2006.

[10] W. H. Winsborough, K. E. Seamons, and V. E. Jones, "Automated Trust Negotiation," DARPA Information Survivability Conference and Exposition. IEEE Press, Jan. 2000.

[11] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. E. Seamons, and M. Winslett, "No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web," in *1st European Semantic Web Symposium (ESWS 2004)*, LNCS, vol. 3053. Heraklion, Crete, Greece: Springer, May 2004, pp. 342–356.

[12] "PeerTrust Project," http://www.L3S.de/peertrust/.

[13] J. W. Lloyd, *Foundations of Logic Programming*, 2nd ed. Springer, 1987.

[14] K. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu, "Requirements for Policy Languages for Trust Negotiation," in *IEEE POLICY'02*, 2002, p. 68.

[15] "Web Services Base Notification," http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf.

[16] "Web Services Topics," http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf.

[17] 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.

[18] D. Chadwick and O. Otenko, "The PERMIS X.509 Role Based Privilege Management Infrastructure," in *7th ACM Symposium on Access Control Models and Technologies*, 2002.

[19] 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 *Proceedings of the 4th Int. Workshop on Grid Computing*, Phoenix, AZ, Nov. 2003.

[20] L. Fang, D. Gannon, and F. Siebenlist, "XPOLA - An Extensible Capability-based Authorization Infrastructure for Grids," in *4th Annual PKI R&D Workshop*, 2005.

[21] V. Welch, T. Barton, K. Keahey, and F. Siebenlist, "Attributes, Anonymity, and Access: Shibboleth and Globus Integration to Facilitate Grid Collaboration," in *4th Annual PKI R&D Workshop*, 2005.

[22] "Shibboleth Project, Internet2," http://shibboleth.internet2.edu.

[23] "OASIS eXtensible Access Control Markup Language (XACML) TC," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

[24] "OASIS Security Services (SAML) TC," http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security.

[25] "Grid Resource Allocation Agreement Protocol WG," https://forge.gridforum.org/projects/graap-wg.

[26] "Web Services Policy Framework," http://www-128.ibm.com/developerworks/webservices/library/specification/ws-polfram/.

[27] M. Alves, C. V. Damásio, D. Olmedilla, and W. Nejdl, "A Distributed Tabling Algorithm for Rule Based Policy Systems," in *7th IEEE POLICY 2006*, London, Ontario, Canada, June 2006.

[28] N. Li, W. Du, and D. Boneh, "Oblivious Signature-Based Envelope," in *PODC '03: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*. New York: ACM Press, 2003, pp. 182–189.

[29] P. A. Bonatti and D. Olmedilla, "Driving and Monitoring Provisional Trust Negotiation with Metapolicies," in *6th IEEE POLICY 2005*, Stockholm, Sweden, June 2005, pp. 14–23.