Globus Toolkit Java Authorization Framework

Rachana Ananthakrishnan (ranantha@mcs.anl.gov)
Tim Freeman (tfreeman@mcs.anl.gov)
Frank Siebenlist (franks@mcs.anl.gov)

February 21, 2007
Updated: October 4, 2007

1. Motivation

Attribute-based authorization, where the authorization decision is not evaluated merely based on the identity of the entity, but also based on attributes asserted by third parties, is increasingly becoming an important requirement. This implies that there is a need for a framework that can collect and communicate these attributes to policy decision points in a standardized way.

Further, the current mechanism for delegation of rights in the Globus Toolkit uses X509 proxy certificates, where all intermediates are empowered to impersonate the original requestor, and the access control policy evaluation only considers that original requester. There is a requirement, however, for a more sophisticated authorization framework that can process restricted delegation and can take the identity of intermediates into account. Rights can be restricted through for example SAML/XACML expressions or external PDPs that allow for fine-grained assertions of rights and attributes, and therefore provide an alternative to the current "blank-check" delegation of rights.

Moreover, the deployment of disparate policy languages and evaluation mechanisms drive the need for a pluggable and generic framework that allows for customized and mechanism-specific policy information points and policy evaluation points.

2. Framework Overview

This new authorization framework facilitates the enforcement of attribute-based authorization and allows for fine-grained delegation of rights. The standardized interfaces for plugging-in policy information points (PIPs) and policy decision points (PDPs) encapsulate all mechanism-specific attribute collection and policy evaluation details.

The attribute-processing module of the framework collects the attributes from the various policy information points and represents those attributes in a canonicalized format for use by the subsequent PDPs.

The policy evaluation module of the framework facilitates the plug-in of of policy decision providers to process restricted delegation and assertions in different formats and evaluation of disparate authorization policies. It also provides a mechanism to search for presence of a valid delegation chain from the requestor to the resource owner, to ascertain access rights. This implies that attributes of intermediates are used in the evaluation process and delegation to these can be restricted.

The attribute processing model and support for fine-grained delegation are key features of this framework and an overview of those are provided below.

2.1 Attribute Processing

An attribute is a piece of information to identify or describe an entity. Our framework collects and processes the attributes and groups them together for the different entities including the requestor, intermediates, and any identity asserting attributes or policies. In the most simple case, the attributes in the requesting context are about the subject requesting access, the resource the access is being requested on, the action that is being requested and the environment. Non-request attributes can be about any subject, resource or action that maybe used in constructing delegation chains from the requestor to the resource owner, or about those entities asserting rights or attributes.

The canonicalized attribute format used for this framework is based on the XACML specification. It defines an attribute to have an attribute identifier, a data type and a set of values. The attribute also has an issuer, who asserts the attribute and optionally has a validity time interval. For example, an attribute could be the group membership of group "Administrator" asserted by "TeraGrid Attribute Authority".

An entity is defined as a collection of various attributes. Attributes that uniquely identify an entity are special kind of attributes, and they are classified as "identity attributes". For example, the X509 Distinguished Name issued by a Certificate Authority is an identity attribute, whereas group membership would be a non-identity attribute. Hence an entity is defined as a collection of identity and non-identity attributes, referred to as entity attributes.

Two sets of entity attributes can be compared to see if the attributes describe the same entity. Two entity attributes are said to be equal if at-least one of the identity attributes match. For example, if one set of entity had X509 Distinguished Name as identity attribute and it matched the X509 Distinguished Name of the other entity, then the attributes can be merged into a single set of entity attributes.

Two attributes are said to be equal if they have the same identifier, data type, issuer and at-least one of their value matches.

For example, if Entity set 1 had the following identity attributes:

AttributeId: Id1
AttributeType: X509 Distinguished Name
Value: O=bar, CN=foo
Issuer: Issuer1

Entity set 2 had the following identity attributes:

AttributeId: Id1
AttributeType: X509 DN
Value: O=bar, CN=foo
Issuer: Issuer1

AttributeId: Id2
AttributeType: Kerberos Token

Value: Some token value
Issuer: Issuer2

Entity Set 3 had the following identity attributes:

AttributeId: Id1
AttributeType: X509 DN
Value: O=bar, CN=different
Issuer: Issuer1

AttributeId: Id2
AttributeType: Kerberos Token
Value: Some other token value
Issuer: Issuer2

In this case, Entity Set 1 and Entity Set 2 can be merged since one of the identity attributes is equal. This allows for merging the attribute collected by various policy information points and results in a rich collection of attributes about the entities. Note that Entity Set 3 cannot be merged with the other two since none of the identity attributes match.

2.2 Delegation of Rights

The framework includes a provider that searches for a single permit delegation chain, from the resource owner to the requestor. The provider first uses all the PIPs that are configured to collect attributes. It then attempts to construct a delegation chain from the resource owner to the requestor. If one such chain exists, a permit decision is returned. If no such chain is found, a deny decision is returned.

For example, say Alice owns resource "gridmap-file.txt".

Alice has policy that states:
    Bob can administrate read access for "gridmap-file.txt"
    Carol can read "gridmap-file.txt"
Entity Bob has policy that says
    Entity Emma read "gridmap-file.txt"
Entity Carol has policy that says
    Entity Deb read "gridmap-file.txt"
    Entity Emma read "gridmap-file.txt"

Now, say Deb attempts to read "gridmap-file.txt". An attempt to construct decision chain is made:

- Only Carol has policy about Deb's access
- There is no policy about Carol administering the resource
- Hence a deny decision is retuned.

Now, Emma attempts to read "gridmap-file.txt"

- ♦ Carol and Bob have policy about Emma's access. So an attempt is made to see if at-least one decision chain can be constructed.
- ♦ There is no policy about Carol administering the resource
- ♦ Now using Bob, Alice has policy that Bob can administer the resource
- ♦ Alice owns the resource
- ♦ So we have a decision chain (Alice -> admin -> Bob -> access -> Emma). Hence a permit decision is returned.

In the above scenario the requestor is Deb or Emma and the resource owner is Alice. But other entities were used to construct the delegation chain and hence the attribute processing framework needs to collect attributes about any entity of interest. Typically these would be the request entities and the PDP decision issuing entities.

Refer to *Delegation Chain Scenarios* section for discussion on constructing delegation chains.

3. Framework Components

The module provides an authorization engine that includes interfaces for PDP/PIP, an attribute processing framework, a configuration framework and a few providers with implementation of some combining algorithms.

The framework consists of a set of interceptors, configuration parameters for them and a combining algorithm. There are three types of interceptors in the framework:

- Policy Information Point (PIP): Each PIP is an information collection point and collects attributes about various entities of interest. Each PIP should group together attributes about same entities and return a list of entities.

- Bootstrap PIP: These are PIPs that collect information only about the request i.e. the peer subject, requested action and resource. Typically these PIPs are run first in the authorization evaluation process, to build the request context.

- Policy Decision Point: Each PDP evaluates the request using some configured policy and returns a decision on whether a said subject can perform a said action on a said resource. A PDP can render two kinds of decision: (a) whether a subject can perform an action on a resource (b) whether a subject can administer a resource to allow other subjects to perform some action on it. These are referred to as access decision and admin decision respectively.

Each authorization configuration consists of a list of Bootstrap PIPs, a list of PIPs and a list of PDPs. The collection of these is referred to as Authorization Chain

A combining algorithm determines how the decisions returned by each PDP is combined to determine whether the operation is allowed or not. It also determines how and when the PIPs should be invoked. The framework defines an interface for combining algorithms and provides two sample implementations.

An authorization chain, with combining algorithm, is used to create an instance of an Authorization Engine. The engine uses the combining algorithm to drive the attribute processing and decision making process.

4. Interfaces

4.1 Attributes and Collections

The module provides an attribute processing framework that defines an attribute using the *Attribute* class. Each attribute is uniquely identified by an attribute identifier, data type and a boolean to indicate if it is an identity attribute. This set is represented using the *AttributeIdentifier* class. Other than the identifier an attribute has an optional issuer, validity and set of values. The framework does not enforce that the values are of the data type specified.

An entity is defined as a collection of identity attributes with at-least one attribute, an optional collection of non-identity attributes and a set of native attributes. Identity attribute collection is represented using *IdentityAttributeCollection* class, non-identity attribute collection is represented using *AttributeCollection* class and an entity using *EntityAttributes* class.

Two attributes are said to be equal if they have the same attribute identifier, data type, issuer and at least one of the values match. Two attribute collections are deemed to be equal if at least one of the attributes in the collection is equal. Equal attributes and collections can be merged. Merging two attributes implies that the value set from one is added to the other. Merging two collections involves merging equal attributes and adding other unique attributes to the collection.

4.2 Policy Information Points

The framework defines an interface for PIPs as follows:

```
public interface PIP {

    public NonRequestEntities collectAttributes(RequestEntities requestEntities)
          throws AttributeException;
}
```

RequestEntities consists of four entities: the requestor, action, resource and environment. If the PIP collects attributes about requestor, action, resource or environment, it should be merged with the relevant entity object with in the *RequestEntities* object. PIPs may collect attributes about non-request entities. In such a case, an *EntityAttributes* object should be created for each and returned as a part of the *NonRequestEntities* collection. The *NonRequestEntities* object consists of three lists – a list of non-request resource entities, non-request action entities and non-request subject entities.

A bootstrap PIP is used to collect attributes only about the request entities and is a specialization of the PIP interface. It does not return any attributes, but only populates the *RequestEntities*.

```
public interface BootstrapPIP extends PIPInterceptor {

   public void collectRequestAttributes(RequestEntities requestEntities)
      throws AttributeException;
}
```

Note: The *PIPInterceptor* interface is explained in later sections

Implementing PIPs section provides details on writing a PIP module.

4.3 Policy Decision Points

The framework defines an interface for PDPs as follows:

```
public interface PDP {

   public Decision canAccess(RequestEntities requestEntities,
                             NonRequestEntities nonReq)
       throws AuthorizationException;

   public Decision canAdminister(RequestEntities requestEntities,
                                 NonRequestEntities nonReq)
                     throws AuthorizationException;
}
```

The interface has two methods: the *canAccess* method is used to decide whether the requestor can access the resource and the *canAdministor* method is used to decide whether the requestor can administer the resource. The request attributes and other subject/resource/action attributes are passed as parameter to both the methods. A *Decision* object that contains the requestor, issuer of decision and validity should be returned.

Implementing PDPs section provides more details on writing a PDP module.

4.4 Authorization Engine

An authorization engine instance consists of a list of PIPs, a chain of PDPs and a combining algorithm that defines how the PIPs should be processed and how the PDP decisions should be combined to get an ultimate decision.

An interface AuthorizationEngineSpi defined that allows for initializing the engine and invoking the processing.

```
public interface AuthorizationEngineSpi extends Serializable {

        public void engineInitialize(String chainName,
                                  AuthorizationConfig authzConfig,
                                  ChainConfig chainConfig)
               throws InitializeException;

         public Decision engineAuthorize(RequestEntities reqEntities,
                                  EntityAttributes resourceOwner)
                  throws AuthorizationException;

         public void engineClose() throws CloseException;

         public ChainConfig getChainConfig();
}
```

The *engineAuthorize* method should contain the combining algorithm and the *engineInitialize* should be used to initialize all interceptors with configuration information. Configuration interfaces are explained in the next section.

4.5 Configuration Interfaces and Classes

4.5.1 Engine Initialization

The engine needs information on the interceptors to use and a scope/prefix for each interceptor to access parameters. The *InterceptorConfig* class is used to store such information for each interceptor. It stores the fully qualified name of the interceptor class name and the prefix for the interceptor. The A*uthorizationConfig* class is used to store such information for all the interceptors that is it has an array of *InterceptorConfig* for each interceptor, *BootstrapPIP, PIP and PDP.*

4.5.2 Interceptor Initialization

Each interceptor might require configuration and parameter information. To allow configuration information to interceptors, all PDPs and PIPs can extend from the following interface:

```
public interface Interceptor extends Serializable {

   public void initialize(String chainName, String prefix, ChainConfig config)
         throws InitializeException;

   public void close() throws CloseException;
}
```

The initialize method is used to push configuration information and the close method is used when the processing is complete to destroy any resources. The *PDPInterceptor* and *PIPInterceptor* interface extend the basic PDP and PIP interface to include the *Interceptor* interface.

The parameters of the initialize method are explained below:

4.5.2.1 Configuration and Parameters

To support a uniform interface to store and access configuration information, the following interface is used:

```
public interface ChainConfig extends Serializable {

   public Object getProperty(String prefix, String property);

   public void setProperty(String prefix, String property, Object value);
}
```

Each property is identified by a scoped name and the value can be any Java object. The actual implementation might store the parameters in any way, for example a map in memory or a database. An authorization engine is initialized with a single object implementing this interface. The scope helps differentiates parameters with the same name for different interceptors.

4.5.2.2 Chain Name

This is a string that is used to uniquely identify the authorization engine this interceptor instance is a part of.

4.5.2.3 Prefix

This is a string that is used as the scope to identify parameters in the configuration object.

5. Providers and Implementation

5.1 Abstract Engine

An implementation of some of the common methods in the *AuthorizationEngineSpi* interface is provided in *AbstractEngine* class. This class is abstract and the *engineAuthorize* method, which implements the PDP combining algorithm needs to be completed by extending classes. The following functionality is provided

- *engineInitialize*: This is a public method required by the interface and invokes initialize all the configured PIP and PDP, in the order it is configured. Any interceptors configured with the same scope and FQDN, only a single instance of the interceptor is created. The same instance is used for processing each occurrence of the interceptor on the chain.
- *engineClose*: This is a public method required by the interface and invokes close on all the configured PIP and PDP, in the order it is configured
- *getChainConfig*: This is a public method required by the interface and returns the configuration object maintained by the engine.

- *collectAttributes:* This method is defined in this class as a protected method and can be overridden by extending implementation. In the base class it invokes *collectAttributes* on all the configured PIPs, in the order they were configured. Moreover, the code checks if the collected attribute is equal to the one in the list and merges if appropriate.

5.2 Combining Algorithms

5.2.1 First Applicable Algorithm

This implementation extends the *AbstractEngine* class and hence the attribute processing is same as the base class. The authorization algorithm invokes each PDP in the order it is configured and the first PERMIT or DENY decision is returned as the decision.

5.2.2 Permit over-ride with Delegation

.
This implementation extends the *AbstractEngine* class and hence the attribute processing is same as the base class. The authorization algorithm attempts to construct a permit decision chain from the resource owner to the requestor. It returns a permit if one such decision chain can be found and deny otherwise.

5.2.2.1 The first PDP is queried with requestor as subject to get a decision on whether the subject has access rights to perform the requested action on the requested resource

5.2.2.2 If the PDP returns a permit, the decision is returned by provider

5.2.2.3 If a decision other than permit is obtained from first PDP, rests of the PDPs are posed the same query, in the order they are configured.

5.2.2.4 If a permit is returned from any PDP and the decision issuer is resource owner a permit is returned.

5.2.2.5 If a permit is returned from any PDP, but decision issuer is not resource owner, then the algorithm is repeated with following query: does the decision issuer have the administrator rights to allow requested action to be performed on requested resource.

5.2.2.6 If by following steps 1 to 5, a decision chain can be constructed from resource owner to requestor, a permit is returned. If not, a deny decision is returned.

The implementation of the above algorithm also keeps track of decision issuers from whom a decision chain to owner could not be constructed. This helps abort pursuing chains that do not lead to owner early.

6. Implementing PIPs

All PIPs should implement the *PIP* or *PIPInterceptor* interface, depending on whether the configuration interface is being used. If the PIP is meant to collect attributes only about the request context, then the PIP can implement the *BootstrapPIP* interface.

The *collectAttributes* method should be populated with code that pulls down attributes. In general, a PIP should process an attribute as follows:

- The attribute should be identified as a subject, environment, resource or action attribute
- It should then be classified as a request context attribute or non-request attribute
- If it is a request attribute,
  - It should be merged with the appropriate attribute entity (subject, environment, resource or action)
- If it is a non-request attribute
  - It should be compared with the entities in the appropriate non-request list (subject, action or resource) to see it is equal to an existing entity. If so, it should be merged.
  - If it is not equal to an existing entity, it should be added to the list.
- The non-request attributes should be returned to the framework.
- If any error occurs in the processing, an *AttributeException* can be thrown depending on the fatality of the error.

Some useful code snippets are discussed below. In this example we assume an attribute for the operation that is invoked. It is an identity attribute since it uniquely identifies the action that is being requested and the value is a string.

```
URI ACTION_URI =  new URI("urn:globus:4.0:container:operation-name");
URI STRING_DATATYPE_URI =
            new URI("http://www.w3.org/2001/XMLSchema#string");
AttributeIdentifier actionIden =
         new AttributeIdentifier(ACTION_URI, STRING_DATATYPE_URI, true):
Attribute operationAttribute =
         new Attribute(actionIden, issuer, Calendar.getInstance(), null);
 operationAttribute.addAttributeValue("{http://foo.bar}add");
 IdentityAttributeCollection attrCol = new IdentityAttributeCollection();
 attrCol.add(operationAttribute);
```

If this attribute is about the request action, then it needs to be merged with the *RequestEntities* object passed as parameter to the method.

```
 EntityAttributes actionEntity = requestAttrs.getAction();
  if (actionEntity == null) {
         actionEntity = new EntityAttributes(attrCol);
         requestAttrs.setAction(actionEntity);
  } else {
         actionEntity.addIdentityAttributes(attrCol);
```

```
   }
```

If the attribute is not about the request action, but is an action attribute, it should be returned as a part of the action attribute list in the *NonRequestEntities* object. The combining algorithm, if extending from *AbstractEngine* will merge the entities from the list with action attributes collected from other PIPs.

```
List actionList = new Vector();
actionList.add(actionEntity);
NonRequestEntities nonReq = new NonRequestEntities(null, null, actionList);
return nonReq;
```

Similar API can be used to process subject, resource and environment attributes.

7. Implementing PDPs

All PDPs should implement the *PDP* interface or the *PDPInterceptor* interface, depending on whether the configuration interface is used.

The *canAccess* and *canAdmin* methods should be used to evaluate the request against the policy as appropriate. A PDP should return one of the following decisions:

- Permit: If the operation on the resource by the subject is not allowed.
- Deny: If the operation on the resource by subject is not allowed
- Not applicable : If some policy information is unavailable, for example a PDP that evaluates SAML Authorization Assertion, would return not applicable if no assertion was collected about the requestor.
- Indeterminate : If some policy configuration is missing/incomplete and a decision cannot be made about the request
- If some fatal exception occurs the *AuthorizationException* should be thrown.

Each decision issued by the PDP should contain a decision issuer (represented as an *EntityAttribute*) and validity. Some useful code snippets are given below.

In this example, let us assume that if the requesting entity is not null, a permit is returned.

```
EntityAttributes issuerEntity = // construct authorization decision issuer.
EntityAttributes reqEntity = reqAttr.getRequestor();
// Evalute request
// Assume decision is valid for an hour
Calendar now = Calendar.getInstance():
Calendar later = Calendar.getInstance();
later.add(Calendar.HOUR, 1);
if (reqEntity == null) {
        return new Decision(issuerEntity, reqEntity,
                Decision.DENY, now, later, exp);
} else {
        return new Decision(issuerEntity, reqEntity,
                Decision.PERMIT, now, later, exp);

}
```

The following code retrieves all action attributes in the request

```
EntityAttributes actionEntity = reqAttr.getAction();
IdentityAttributes identityAttr = actionEntity.getIdentityAttributes();
Iterator iteraror = identityAttr.getCollection().iterator():
```

The following code gets all attributes with said identifier, which is the attribute id, data type and identity/non/identity flag. Since multiple issuers can issue attributes with same identifier, it returns a collection

```
URI ACTION_URI =  new URI("urn:globus:4.0:container:operation-name");
URI STRING_DATATYPE_URI =
        new URI("http://www.w3.org/2001/XMLSchema#string");
AttributeIdentifier actionIden =
        new AttributeIdentifier(ACTION_URI, STRING_DATATYPE_URI, true):
Iterator iterator = identityAttr.getAttributes(actionIden);
```

To get an attribute issued by a specific entity and its value:

```
Attribute actionAtt = identityAttr.getAttributes(actionIden, issuerEntity);
Set values = actionAttr.getValueSet()
```

8. Implementing Combining Algorithms

Combining algorithms should implement the *AuthorizationEngineSpi* interface. Typically the combining algorithm needs to do the following:

- Create an instance of *NonRequestEntities*. This instance is used to maintain a merged list that is passed to the PDPs.
- Process PIPs in some said order
- Process the non-request attributes returned by each PIP
    - For each list (subject, action and resource) in the returned non-request attributes, check if the entity already exists in the instance NonRequestEntities.
    - If it does, merge appropriately
    - If not, add to the instance list
- Use the request attributes and non-request attributes to invoke PDPs as required to return an ultimate decision.
- Propagate errors from the individual PDPs either as part of decision or as an exception.

9. Using Configuration Interfaces

The configuration interfaces can be used to set up the authorization engine with information about the PIPs and PDPs to use. Other interfaces also allow to push parameter information to the PIPs and PDPs. themselves.

Below is a sample code snippet to show configuring authorization engine with PDP and PIP information.

```
InterceptorConfig bootstrap1 =
        new InterceptorConfig("scope1", BootstrapPIP1.class.getName());
InterceptorConfig bootstrap2 =
        new InterceptorConfig("scope2", BootstrapPIP2.class.getName());
InterceptorConfig bootstraps = new InterceptorConfig[] { bootstrap1, bootstrap2 };

InterceptorConfig pip1 =
        new InterceptorConfig("pip1", PIP1.class.getName());
InterceptorConfig pip2 =
        new InterceptorConfig("pip2", PIP2.class.getName());
InterceptorConfig[] pips = new InterceptorConfig[] { pip1, pip2 };

InterceptorConfig pdp1 =
        new InterceptorConfig("pdp1", PDP1.class.getName());
InterceptorConfig pdp2 =
        new InterceptorConfig("pdp2", PDP2.class.getName());
InterceprorConfig[] pdp = new Interceptor[] { pdp1, pdp2 };

AuthorizationConfig config =  new AuthorizationConfig(bootstraps, pdps, pdp);
```

Parameter information to each PDP/PIP (interceptor) can be pushed using the *ChainConfig* interface. An implementation of ChainConfig can use any backend to store the parameter information. For example, a simple implementation could be an in memory store or a flat file. So to set information for a specific interceptor, the scope and parameter name can be used. For example, for PDP1, the following can be used to set parameter.

```
chainConfigImpl.setProperty("pdp1", "examplePropertyName", valueObject)
```

The engine can be initialized with the *AuthorizationConfig* object and the *ChainConfig* object. The engine itself pushes the *ChainConfig* information to all the interceptors. The interceptors can use the prefix and property name to retrieve parameter information. For example, for PDP1, the following can be used to get the parameter.

```
Object parameter = chainConfigImpl.getProperty("pdp1", "examplePropertyName");
```

10. Building and Installing

The source code depends only on Apache Commons Logging and the test code depends on JUnit.

- ♦ Untar the code base:
  - o tar xvf authzFramework.tar.gz
- ♦ cd authzFramework
- ♦ mkdir engine/source/lib
- ♦ mkdir engine/test/lib
- ♦ Add logging jars
  - o cp log4j-1.2.13.jar engine/source/lib
  - o cp commons-loggin-1.1.jar engine/source/lib
- ♦ Add Junit jars
  - o cp junit.jar engine/test/lib
- ♦ Create distribution
  - o cd engine/source
  - o ant dist
- ♦ The distribution is in engine/dist directory and all jars in that directory will be needed for the engine to work.
- ♦ To test the engine code
  - o cd engine/test
  - o ant test
  - o Note: The test depends on the distribution created in previous step

11. Future Enhancements

This section provides an overview on some of the enhancements that can be made to this framework. There are no specific plans in place for implementing these nor has any order of priority been established.

11.1 PIP processing enhancements

- ♦ Support for each PIP to be run on a separate thread. A synchronized wait thread waits for the threads to finish. A time limit can be set, such that any PIP that takes longer than the specified ceiling time to finish will be terminated. Attribute collapsing will need to be coordinated, likely by callbacks to the synchronized wait thread).

- ♦ The attributes collected can be persisted so that they can be re-used in subsequent runs. Based on the context as key, the attributes stored for a given authorization chain can be retrieved. If any one of the attributes has expired, the PIP list needs to be processed again. This feature can be sped up some more, by caching these attributes in memory. Support for client requesting a client flush will be important, especially in cases where the client would like a new role.

- Support for collecting attributes about an entity in the context of values of attributes for other entities. This would require the framework to cache and process attributes in the context of the entities. For example, a subject is a VO admin only if accessing a particular resource, but is a VO user for other resources.

- PIP invocation on demand, rather than in order provided. This would be for PIPs to be processed only when cached attributes expire or for PDPs callback to PIPs. This would require the framework to store and process metadata about the attributes, including the PIP that was used to collect them.

11.2 PDP processing enhancements

- Each decision rendered by an authorization chain, given a collection of attributes, can be persisted for use in subsequent calls. So prior to processing the PDP chain, the store can be looked up and if a valid decision exists, it can be used. This can be further enhanced by caching some of the decisions, so a look up to some persistent store is saved.

12. Appendix

12.1 Delegation Chain Scenarios

12.1.1 Scenario 1

If we have an x509 EE cert for Rachana that is used for the authentication of her request, then after the x509 path validation, we end up with two attributes:

attr1 = Attribute (name="x509SubjectDN", value="CN=Rachana", identity=true)

attr2= Attribute(name=publicKey, value="#$%$^&$&Rachana&*^*&#", identity=true)

We "know" that both of these attributes are identity attributes for the entity, and we know they refer to the name entity, so we can add them to the same set:

entity1 = Set(a1, a2)

Furthermore, we "know" that this entity was associated with the "requester" of the service invocation, so we keep this special relationship by assigning:

theRequester = entity1

We also "know" what the request is in the form of a porttype, resource, action, such that we can fill-in

attr3 = Attribute(name="portType" value="fileTransferPortType")

```
attr4 = Attribute(  name="resourceId",  value="#%%@resrcId$$%#",  identity=true)

entity2 = Set(attr3, attr4)

theResource = entity2

attr5 = Attribute(  name="operation"  value="uri:...read")

entity3 = Set(attr5)

theAction = entity3
```

Now we are ready to ask the question whether the resource is allowed to invoke the operation on the ws-resource. We do that through a function where we pass a list of all the attribute sets, and where we separately pass the references of the subject/resource/action entities:

```
decisionResult = PDP.canAccess( List(entity1, entity2, entity3), theRequester,
                                theResource, theAction)
```

If this would be a call-out, then you would pass the list of entities by value, and the subject/resource/action as references to one of the passed sets.

12.1.2 Scenario 2

Building on Scenario 1, suppose that an attribute assertion was pushed with the request or that an attribute service was configured to be called, and that the asserted attribute was a "group" with value "anl" and asserted through an signature signed by someone with an x509 EE cert with subject name "tim", then we would have the following in addition to the requestor, action and resource attribute collected from the X509 EE Certificate used to authenticate the request.

First we create a new entity for Tim, because we couldn't find any existing entity set that matched the following two identity attribute values:

```
attr6 = Attribute(name="x509SubjectDN",value="CN=Tim", identity=true)

attr7= Attribute(name=publicKey, value="#$%$^&$&Tim&*^*&#", identity=true)

entity4 = Set(attr6, attr7)
```

Then we can use this new entity in the new attribute as the issuer. The subject of the attribute assertion had CN=Rachana, so we can find that entity1 has a matching identity attribute with the same name/value, and we can subsequently add the new attribute to that set:

```
attr8 = Attribute(name = "group", value = "anl", issuer = entity4
entity1 = entity1.add(attr8)
```

and finally the PDP function call that would have Tim's entity added to the list, while Rachana's additional attribute was added to here entity's attribute set:

```
decisionResult = PDP.canAccess( List(entity1, entity2, entity3, entity4),
                    theRequester,  theResource, theAction)
```

So even we don't know whether the new attribute will be used, or if the new entity will be used inside of the PDP function, we pass all we have.

12.1.3 Scenario 3

Building on Scenario 2, assume that Rachana also pushes an authorization assertion signed by Frank, plus another attribute assertion for Frank, signed by Tim.

Then the authorization assertion's signature will be validated, and the issuer will be known through its x509 cert, which in this case will yield yet another new entity:

```
attr9 = Attribute(name="x509SubjectDN", value="CN=Frank", identity=true)

attr10= Attribute(name=publicKey, value="#$%$^&$&Frank&*^*&#", identity=true)

entity5 = Set(attr9, attr10)
```

The new attribute assertion will have a subject that will match Frank, such that it should be added to the associated entity5, plus the issuer can be matched to Tim's entity4:

```
attr11 = Attribute(name = "group", value = "anl", issuer = entity4

entity5 = entity5.add(attr11)
```

Finally, the PDP call will look like:

```
decisionResult = PDP.canAccess( List(entity1, entity2, entity3, entity4, entity5),
                    theRequester, theResource, theAction)
```

Note that Frank's authorization assertion has also yielded a new PDP instance for that authorization statement that may or may not be called in subsequent PDP function call inside of the accessPDP call to find the delegation chains.
So that function call would look like:

```
theAdmin = entity5

decisionResult = PDP.canAdmin( List(entity1, entity2, entity3, entity4, entity5),
```

theAdmin theRequester, theResource, theAction)