NORDUGRID

NORDUGRID-TECH-16
6/8/08

SECURITY FRAMEWORK OF ARC1

W.Qiang, A.Konstantinov

Abstract

This document is about security design concerns and ideas, as well as security framework implementation in the ARC1 middleware.

Table of Contents

1. Introduction

The security framework of the ARC1 includes two parts of capabilities: security capability embedded in hosting environment, and security capability implemented as plug-ins with well-defined interfaces which can be accessed by hosting environment and applications. The following design concerns were employed when designing:

- Interoperability and standardization

In consistent with the main design concern of the ARC1, interoperability and standardization is considered in security framework. For example, in terms of authentication, PKI infrastructure and proxy certificate (RFC3820 [1]) is used as most of the other grid middle-wares do. Since supporting of standardization is a way for implementing interoperability, some standard specifications have been implemented as prototype and tested, such as SAML specification.

- Modularity and extensibility

Besides the security functionality which is embedded in hosting environment, the other security functionality is implemented as plug-ins which has well-defined interfaces, and is configurable and dynamically loadable. Since the interoperation interface between security plug-in and hosting environment or applications is predefined, it is easy to extend the security functionality in order to support some other security capability by implementing the interface.

- Backward compatibility

The GSI (Grid Security Infrastructure) based mechanism has been a de-facto solution for grid security. The design of security framework should be compatible to it.

2. Security architecture in HED. SecHandler and PDP

2.1. Structure of SecHandler and PDP

In the implementation of the ARC1, there is a Service Container – the Hosting Environment Daemon (HED) (D1.2-2, [2]) which provides a hosting place for various services in application level, as well as a flexible and efficient communication mechanism.

HED contains a framework for implementing and enforcing authentication and authorization. Each Message Chain Component (MCC) or service has a common interface for implementing various authentication and authorization functionality. This functionality is implemented by using pluggable components (plug-ins) called SecHandler. The SecHandler components are C++ classes and provide method for processing messages traveling through Message Chains of the HED. Each MCC or Service usually implement two queues of SecHandlers – one for incoming messages and one for outgoing called “incoming” and “outgoing” respectively. It is possible for MCC or Service to implement other set of queues. Please check documentation of particular component for that particular information. All SecHandler components attached to the queue are executed sequentially. If any of them fails, message processing fails as well.

Each SecHandler is configured inside same configuration file used for configuring whole chain of MCCs. Some of implemented SecHandler components also make use of pluggable and configurable sub-modules which specifically handle various security functionalities, such as authorization, authentication, etc. The currently implemented sub-modules used by some SecHandlers are Policy Decision Point (PDP) components such as Arc PDP which can process ARC specific Request and Policy documents. Figure 1 gives the structure of a MCC/Service, and the message sequence inside it. And Figure 2 shows the configuration of SecHandler components for an example “Echo” service.

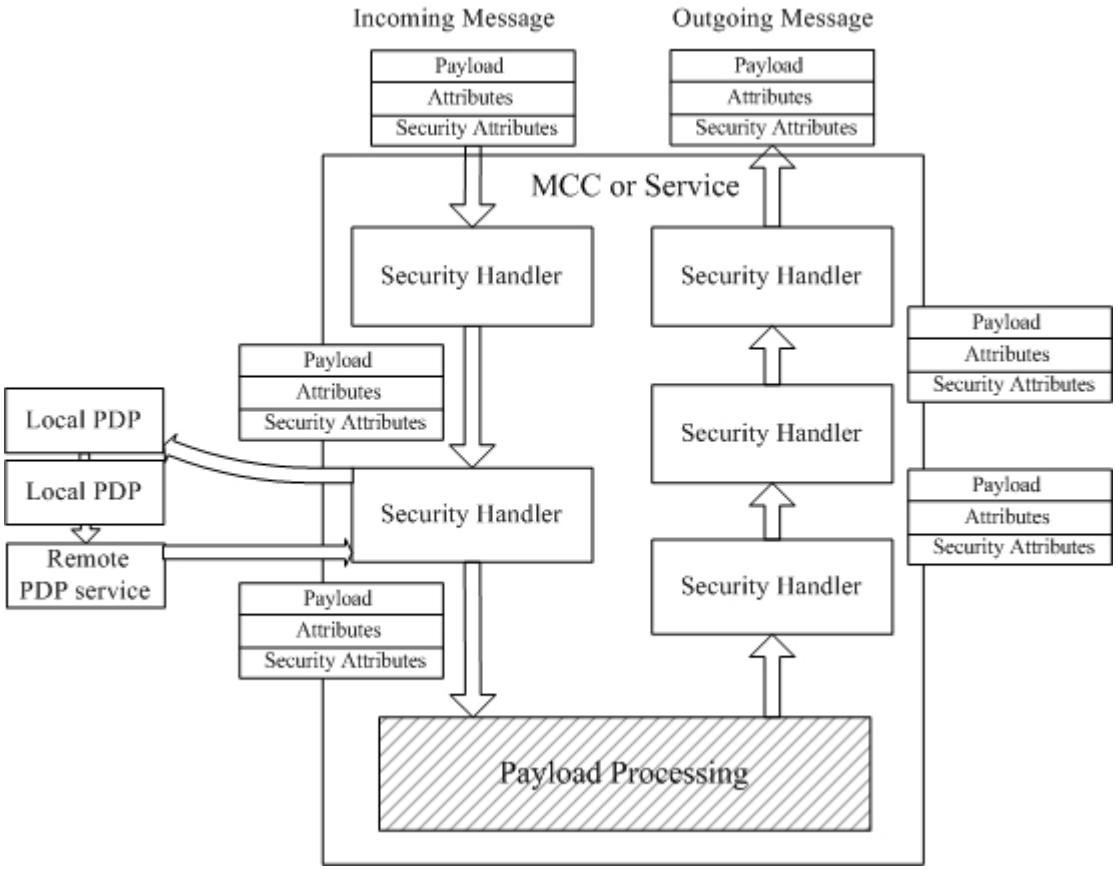

Figure 1. There are usually two chains of SecHandlers inside the MCC or service. Each SecHandler will parse the Security Attributes which are generated by the upstream MCC/services or probably upstream SecHandlers in the same or other MCC/Service, and do message processing or authenticate or authorize the incoming/outgoing message based on the collected information. The SecHandler can also change the payload and attributes of Messsage itself. For example, the Username-Token SecHandler will insert the WSS Username Token [3] into header part of SOAP message. The PDPs are called by the SecHandlers and are supposed to make authorization decision. Here the two local PDP and one remote PDP service is just for demonstration, and one or any number of PDPs can be configured under each SecHandler.

```
<Service name="echo" id="echo">
  <SecHandler name="identity.map" id="map" event="incoming">
    <PDP name="allow.pdp"><LocalName>test</LocalName></PDP>
  </SecHandler>
  <SecHandler name="arc.authz" id="authz" event="incoming">
    <PDP name="arc.pdp">
      <PolicyStore>
        <Location type="file">policy.xml</Location>
        <!-- other policy location-->
      </PolicyStore>
    </PDP>
    <PDP name="simplelist.pdp" location="pemittedlist.txt"/>
  </SecHandler>
</Service>
```

Figure 2. Example Echo service is configured to use two SecHandlers, both responsible for authorization. First SecHandler uses the identity of client extracted from the incoming message to map it into local identity like local Linux username. In this case all clients are mapped to local account "test". The second one uses two PDPs: one will compose ARC specific authorization request based on the Security Attributes collected from "incoming" message and evaluate it against the ARC specific authorization policy which is defined in local file "policy.xml"; the other will compare the X509 identity of client extracted from the incoming message against list of identities stored locally.

2.2. Interface of SecHandler

When one component (MCC or service) is loaded according to the configuration information, the SecHandler under the component and the plug-ins like PDP which are attached to the SecHandler will be loaded as well.

There is one simple interface (see Figure 3) defined in class SecHandler, which will be called by the containing MCC/Service once there is message (incoming or outgoing) need to be processed.

```
class SecHandler {
 public:
  SecHandler(Arc::Config*) {};
  virtual ~SecHandler() {};
  virtual bool Handle(Arc::Message *msg) = 0;
};
```

Figure 3. class SecHandler is an abstract class which includes a general interface called Handle which uses Message object as argument. Any security handler implementation should inherit class SecHandler and implement the interface according to the actual functionality. The interface only return simple Boolean value, and any useful information generated during the calling of this interface should be put into the security attribute of the message, or put into the payload itself.

Currently, the ARC1 comes with the following four security handler implemented:

- *arc.authz* – Authorization SecHandler

The *arc.authz* is responsible for calling the interface of policy decision point and getting back the authorization result, and then making decision according to this authorization result. There is one simple interface (see Figure 4) defined in PDP, which will be called by *arc.authz* if configured inside once there is message (incoming or outgoing) need to be processed.

- *identity.map* – Identity Mapping SecHandler

The *identity.map* is a specific authorization oriented security handler. It will map the global identity in the message into local identity like system username based on the result returned by Policy Decision Point components.

- *delegation.collector* – Delegation SecHandler

The *delegation.collector* is responsible for collecting the delegation policy information from the remote proxy credential (proxy certificate is compatible to RFC3820) inside the message, and putting this policy into message's security attribute for the usage of other components, such as *delegation.pdp*.

- *usernametoken.handler* – UseranemToken SecHandler

The task of the *usernametoken.handler* is to generate the WS-Security Username-Token and add it into header of SOAP message which is the payload of outgoing message. It can also extract the WS-Security Username-Token from the header of SOAP message which is the payload of incoming message.

2.3. Interface of PDP

Figure 4 shows the definition of abstract class PDP. The implementation could be some function which implements the interface by composing the policy evaluation request, evaluating this request against some policy, and returning the evaluation result, or just by composing the policy evaluation request, invoking some remote policy decision web service and getting back the evaluation result.

```
class PDP {
 public:
  PDP(Arc::Config* cfg) { };
  virtual ~PDP() {};
  virtual bool isPermitted(Arc::Message *msg) = 0;
`
```

Figure 4. class PDP is an abstract class which includes a general interface called isPermitted which uses Message object as argument. Any policy decision point implementation should inherit class PDP and implement the interface according to the actual functionality. The interface only return simple Boolean value, and any useful information generated during the calling of this interface should be put into the security attribute of the message, or put into the payload itself.

Currently, the ARC1 comes with the following four policy decision point implementation:

- *arc.pdp* – Arc PDP

The Arc PDP will organize the security attributes into the ARC specific authorization request, call the policy evaluator to evaluate the request against the policy (which is in ARC specific format) repository, and get back the evaluation result. See paragraph 3 for detail information about request schema and policy schema.

- *delegtion.pdp* – Delegation PDP

The Delegation PDP is basically similar to Arc PDP, except it uses the delegation policy parsed from remote proxy credential by *delegation.collector*, and evaluates the request against delegation policy. See section 6for the design idea and use case of delegation policy in fine-grained identity delegation.

- *simplelist.pdp* – Simplelist PDP

The Simplelist PDP is a simplest implementation of policy decision point. It will match the identity extracted from the remote credential (or proxy credential) with local list of permitted identities.

- *pdpservice.invoker* – PDP Service Invoker

The PDP Service Invoker is a client which can be used to invoke the PDP Service which implements the same functionality as Arc PDP, except that the evaluation request and response are carried by SOAP message. The benefit of implementing PDP Service and PDP Service Invoker is that the policy evaluation engine can be accessed remotely and maintained centrally.

3. Policy Evaluation Engine

3.1. Design of policy evaluation engine

The ARC1 defines specific evaluation request and policy schema. Based on the schema definition, one policy evaluation engine is implemented. The design principal of policy evaluation engine is generality by which the implementation of the policy evaluation engine can be easily extended to adopt some other policy schema, such as XACML policy schema.

Figure 5 shows the UML class diagram about the policy evaluation engine. It shows all classes and relations simultaneously for getting the overall picture.

The *Evaluator* class is the key class for policy evaluation. It accepts request evaluates it against loaded policy and returns evaluation response.

Three abstract factories - *FnFactory*, *AlgFactory*, *AttributeFactory* - are responsible for creating the *Function*, *CombiningAlg* and *AttributeValue* objects correspondingly. The classes inherited from *CombiningAlg* class take care of implementing various combining algorithms which define relations between <Rule/> elements in policy. The *AttributeValue* type of classes are used for processing different types of <Attribute/> and similar elements. The *Function* classes take care of comparing <Attribute/> elements of request and policy.

The *Policy* class parses <Policy/> or <Rule/> elements and creates *CombingAlg* objects according to the <RuleCombiningAlg/> attribute of <Policy/>, *Function* objects according to the <Function/> attribute of <Attribute/> and *AttributeValue* objects according to the <Type/> attribute of <Attribute/>. Those objects will be used when evaluating the request.

The *Request* class is responsible for parsing <Request/> element and creates corresponding *AttributeValue* objects according to the <Type/> attribute of <Attribute/>. When evaluating, each *AttributeValue* in request will be evaluated against corresponding *AttributeValue* in the policy by using relevant *Function*.

Due to extensible architecture of code it is relatively easy to add support for new types of *AttributeValue*, *Function and CombingAlg* objects in this way supporting various types of XML based policy languages.

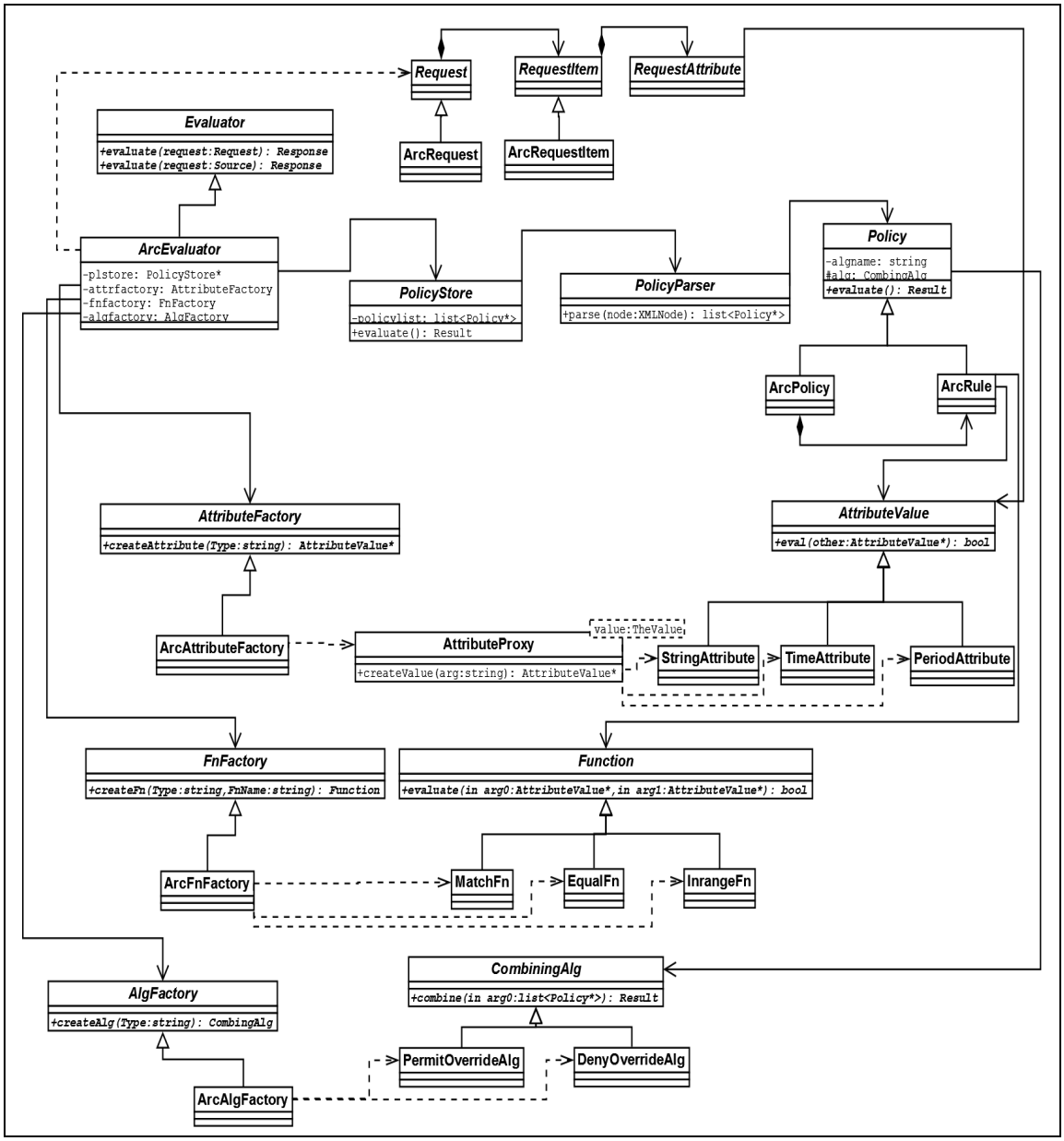

Figure 5. The UML class diagram of the classes inside policy evaluation engine.

3.2. Schemas for policy evaluation engine

The schema for ARC Policy is available at
http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/Policy.xsd .
The hierarchy tree of ARC Policy is show below (numbers show multiplicity of elements)

Policy (1)
Rule (1-)
Subjects (1)
Subject (1-)
Attribute (1-)
Resources (0-1)
Resource (1-)
Actions (0-1)

Action (1-)

Conditions (0-1)

Condition (1-)

Attribute (1-)

The schema for ARC Request is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/Request.xsd .

The hierarchy tree of ARC Request is show below (numbers show multiplicity of elements)

Request (1)

RequestItem (1-)

Subject (1-)

SubjectAttribute (1-)

Resource (0-)

Action (0-)

Context (0-)

ContextAttribute (1-)

The schema for ARC Response is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/Response.xsd .

The ARC Response is not used directly in code. It is in use by PDP Service which provides remote evaluation of policies.

3.3. Basic Elements of Policy

There are 2 basic objects - "policy" and "request". There is 1 main actor - Evaluator. Curretly there are two types of elements in policy: Policy and Rule. Policy element is made of Rule elements.

Evaluator matches request to policy and produces one of 4 following results:

- **PERMIT** - policy explicitely permits activity specified in request because request matches some part of policy and corresponding effect specified in policy is PERMIT.

Example:

Rule: *PERMIT person ALICE to PLAY in place called WONDERLAND*

Request: *person ALICE wants to PLAY in place called WONDERLAND*

- **DENY** - policy explicitely denies activity specified in Request because Request matches some part of policy and corresponding effect specified in policy is DENY.

Example:

Rule: DENY *fruit APPLE to GROW on PEACH tree*

Request*: fruit APPLE to be GROWN on PEACH tree*

- **INDETERMINATE** - request has some part which does not correspond to policy.

Example:

Rule*: DENY fruit APPLE to GROW on PEACH tree*

Request*: fruit APPLE to be GROWN on WHEAT ground*

Request*: flower SUNFLOWER to be grown on PEACH tree*

Explaination: Here, it is not possible to obtain any matching result - neither positive (DENY or PERMIT) nor negative (NOT_APPLICABLE, see below)

In the request, the "ground" is completely uncomparable to the "tree" in policy. One can compare

"PEACH tree" and "APPLE tree" because they are both "tree"; But it is impossible to compare "PEACH tree" and "WHEAT ground" becaue they of different kind (Policy is about tree and Request is about ground).

In a similar way one can't compare "fruit APPLE" and "flower SUNFLOWER" (here policy is about fruits and Request is about flower).

Any other situation which makes it impossibile to compare two attributes will also cause "INDETERMINATE".

- **NOT_APPLICABLE** - all parts of the Request have corresponding parts in the Policy, but some value of those parts are not the same. Hence request does not match policy.

Example:

Rule*: DENY fruit APPLE to GROW on PEACH tree*

Request*: fruit APPLE to be GROWN on APPLE tree*

Request*: fruit ORANGE to be GROWN on PEACH tree*

Request*: fruit ORANGE to be GROWN on APPLE tree*

Explanation: for each part of the Request evaluator can find relevant part in the Policy - both Policy and Request are about fruit and tree. But the values do not match.

If it is required to reduce evaluation results to boolean value PERMIT maps to TRUE and rest of results to FALSE.

Note*: It would be useful to make it possible to specify secondary effect which would become* active in case Request is NOT_APPLICABLE. For example:

DENY fruit APPLE to GROW on PEACH tree otherwise PERMIT

But one should be careful because example above would allow fruit PLUMS to grow on APPLE trees :)

This kind of requirement can be supported by using the algorithm between policies. For example, in case of above scenario, we can use some algorithm like "Permit-if-notapplicable". See below the "Policy matching" part for more explaination.

3.4. Policy MAtching

Policy is made of Rule elements. Request is evaluated against each Rule. Each evaluation produces same results as policy evaluation described above. The results from all Rules are then combined in order to produce final result for whole policy. Results Combining Algorithm is specified in Policy. There are 26 algorithms currently:

- ***Deny-Overrides*** - this is default if no algorithm specified.
  - If there is at least one DENY in results final result is DENY.
  - Otherwise if there is at least one PERMIT, the final result is PERMIT.
  - Otherwise if there is at least one NOT_APPLICABLE final result is NOT_APPLICABLE.
  - Otherwise final result is INDETERMINATE.

NOTE: CHECK if algorithm description is correct

Special case is Policy wih no rules. Probably such policy shoudl be treated as always producing DENY.

Discussion

If there is no Rule under Policy, it means no restriction from this Policy, what exactly shoud the Policy give? DENY or NOT_APPLICABLE?

It seems to be more logical to produce INDETERMINATE because there is no rule with elements which could be compared to request.

- ***Permit-Overrides.***
    - If there is at least one PERMIT in results final result is PERMIT.
    - Otherwise if there is at least one DENY the final result is DENY.
    - Otherwise if there is at least one NOT_APPLICABLE final result is NOT_APPLICABLE.
    - Otherwise final result is INDETERMINATE.

Special case is Policy wih no rules. Probably such policy should be treated as always producing DENY.

NOTE: CHECK if algorithm description is correct

Discussion

Question? The same as above.

- ***Ordered algorithms.***

These specify priorities for all four possible results. Their names look like Result1-Result2-Result3-Result4 with Result# naming result types, for example Permit-Deny-NotApplicable-Indeterminate. The results are combined in following way:

- If there is at least one result of Result1 type then final result is Result1.
- Otherwise if there is at least one result of Result2 type then final result is Result2.
- Otherwise if there is at least one result of Result3 type then final result is Result3.
- Otherwise final result is Result4.

There are 24 possible combinations of those algorithms.

Note: It would be useful to have more combining algorithms. For example

- Permit-if-notapplicable - the use case could be "DENY fruit APPLE to GROW on PEACH tree otherwise PERMIT". In this case there is only one Rule under Policy, and this Rule is with "Deny" effect.
    - If this Rule gives DENY in results, final result is DENY.
    - Otherwise if this Rule gives NOT_APPLICABLE, final result is PERMIT.
    - Otherwise final result is INDETERMINATE.
- “Permit-if-allPermit" - Permit if all the Rules gives Permit, this algorithm is useful in case if we are collecting different policies from a few sources, and we want the request to satisfy all of them.
    - If all of the Rule give PERMIT, the final result is PERMIT.
    - Otherwise if there is at least one DENY the final result is DENY.
    - Otherwise if there is at least one NOT_APPLICABLE final result is NOT_APPLICABLE.
    - Otherwise final result is INDETERMINATE.
- OnlyOneApplicable

- If there is one gives INDETERMINATE, final result INDETERMINATE is given immediately.
- Otherwise if there is exactly only one gives applicable result (DENY or PERMIT), final result is as this result.
- Otherwise if there is more than one gives applicable result, final result is INDTERMINATE.
- Otherwise final result is NOT_APPLICABLE.

This algorithm makes sure that only one Rule is selected when making decision.

- FirstApplicable
  - If there is one give DENY, PERMIT or INDETERMINATE result, final result is given immediately as this result.
  - Otherwise final result is NOT_APPLICABLE.

3.5. Request Structure

Request is made of RequestItem elements. Each RequestItem is evaluated against Policy Rule and for each evaluation separate result is generated as described above.RequestItem is made of 4 elements:

Subject - represents entity requesting specified action

- Resource - destination/object of the action
- Action - specifies what has to be done on resource
- Context - for additional information which does not fit anywhere else, like the current time.

Effectively RequestItem may have only one Subject, one Resource, one Action and one Context. If there are more than one element of any kind of sub-elelemt, then in the evaluator this RequestItem is split into several items containing all possible permutations and results are obtained for every item separately. How results are combined will be explained later.

Additionally Subject could contain sub-elements SubjectAttribute. Those are meant to represent different kinds of requesters' identities. Example:

- Subject
  - SubjectAttribute: name is ALICE
  - SubjectAttribute: age is YOUNG
  - SubjectAttribute: gender is GIRL

Context could also be made of ContextAttribute elements in the same way as Subject.

The following is an example of the Request:

```
<Request xmlns="http://www.nordugrid.org/schemas/request-arc">
 <RequestItem>
  <Subject>
   <SubjectAttribute AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</SubjectAttribute>
   <SubjectAttribute AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</SubjectAttribute>
  </Subject>
  <Subject AttributeId="urn:knowarc:shibboleth:attribute">member</Subject>
  <Action AttributeId="urn:knowarc:fileoperation">Read</Action>
  <Resource AttributeId="urn:knowarc:fileidentity>file:///home/test</Resource>
  <Context AttributeId="urn:knowarc:time" Type="time">2008-09-15T20:30:20</Context>
 </RequestItem>
```

```
</Request>
```

While evaluating this RequestItem will be split into two RequestItems:

```
<Request xmlns="http://www.nordugrid.org/schemas/request-arc">
 <RequestItem>
  <Subject>
    <SubjectAttribute AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</SubjectAttribute>
    <SubjectAttribute AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</SubjectAttribute>
  </Subject>
  <Action AttributeId="urn:knowarc:fileoperation">Read</Action>
  <Resource AttributeId="urn:knowarc:fileidentity>file:///home/test</Resource>
  <Context AttributeId="urn:knowarc:time" Type="time">2008-09-15T20:30:20</Context>
 </RequestItem>
 <RequestItem>
  <Subject AttributeId="urn:knowarc:shibboleth:attribute">member</Subject>
  <Action AttributeId="urn:knowarc:fileoperation">Read</Action>
  <Resource AttributeId="urn:knowarc:fileidentity>file:///home/test</Resource>
  <Context AttributeId="urn:knowarc:time" Type="time">2008-09-15T20:30:20</Context>
 </RequestItem>
</Request>
```

The following means this Subject possesses both of these Attributes.

```
  <Subject>
    <SubjectAttribute AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</SubjectAttribute>
    <SubjectAttribute AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</SubjectAttribute>
  </Subject>
```

However, the following means two Subject each of which possesses one Attribute.

```
  <Subject AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</Subject>
  <Subject AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</Subject>
```

The "Type" xml-attribute is for distinguishing how to process the xml-node value, which is critical when evaluate two value from request side and policy side because different type requires different evaluating/comparing approach. The default "Type" is "string", in this case (also with the "Function" xml-attribute on the policy side is "equal", which will be explained later), each letters of these two values will be compared one by one when evaluating them.

The "AttributeId" xml-attribute is for evaluator to find the Attribute with AttributeId from the request side which corresponds to the Attribute with the same AttributeId on the policy side. Only if two Attributes' AttributeId are equal, the evaluator will then compare the value.

Each RequestItem will be sequencialy and independently evaluated against policy/policies. So for one Request (including few RequestItems), some RequestItem could get positive evaluation result (PERMIT) from policy engine, others could get negative evaluation result (DENY, NOT_APPLICABLE, INDETERMINATE).

It is up to policy decision point to make final decision according to the evaluation results returned by evaluator, and the evaluator itself can not give this kind of final decision.

NOTE: This probably should be changed because evaluator is fed with complete policy and complete request. Hence it is illogical that it returns multiple decisions.

Basically the policy decision point will feed policy engine with request, get back evaluation results,

and make final decision.

3.6. Rule Composition and Matching

Policy rule is made of 4 elements - Subjects, Resources, Actions, Conditions (See the following example). Those are only used to group multiple elements Subject, Resource, Action, Condition. For instance, you can merge two Rules with the same Resources, Actions, Conditions, and the same "Effect" but different Subjects into one Rule.

NOTE: That is strange. Why do we need Rule at all?

There is no logical relationship between Subject, which means you can split one Rule into two Subject (under Subjects) into two Rule (each of which has one Subject (under Subjects)).From now only later ones (Subjects with only one Subject as sub-element, and the same for others) are described. Their meaning is same as in request with Condition corresponding to Context. Subject and Condition elements are also made of Attributes. All elements may be present more than one time. During procedure of matching each element in RequestItem is matched against all elements of same kind in Policy - Subject is matched to Subject, Resource to Resource, etc. For every combination 3 possible results are produced:

- MATCHED - element from RequestItem matched element in Policy Rule. Example:

RequestItem Resource*: place called WONDERLAND*

PolicyItem Resource*: place called WONDERLAND*

- NOT MATCHED - element from RequestItem did not match element in Policy Rule. Example:

RequestItem Resource*: place called WONDERLAND*

PolicyItem Resource*: place called PLAYGROUND*

- INDETERMINATE- element from RequestItem could not be compared to element in Policy Rule because they are of incompatible ids/belong to different namespaces. Example:

RequestItem Resource*: place called WONDERLAND (with namespace "place")*

PolicyItem Resource*: LEMON tree (with namespace "tree")*

The produced results then combined to produce final 4 types of results in following way:

- If for every element in RequestItem there is at least one MATCHED result then result for this Policy Rule is as specified in the corresponding Effect (Deny or Permit).
- Otherwise if for every element in RequestItem there is at least one gets INDETERMINATE result then result for Policy Rule is INDETERMINATE.
- Otherwise result is NOT_APPLICABLE.

Special case is then RequestItem does not have the element(s) of some kind (Subject, Action, Resource or Context/Condition). If there are elements of corresponding kind in the Policy Rule then such situation should be considered as INDETERMINATE.

The following is an example of the Policy:

```
<Policy xmlns="http://www.nordugrid.org/schemas/policy-arc" CombiningAlg="Permit-Overrides">
 <Rule Effect="Permit">
  <Subjects>
   <Subject>
    <Attribute AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</Attribute>
    <Attribute AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</Attribute>
```

```
  </Subject>
  <Subject AttributeId="urn:knowarc:shibboleth:attribute">member</Subject>
 </Subjects>
 <Actions>
  <Action AttributeId="urn:knowarc:fileoperation">Read</Action>
  <Action AttributeId="urn:knowarc:fileoperation">Delete</Action>
 </Actions>
 <Resources>
  <Resource AttributeId="urn:knowarc:fileidentity">file:///home/test</Resource>
 </Resources>
 <Conditions>
                                    <Condition AttributeId="urn:knowarc:period" Type="period"
Function="Inrange">2008-09-10T20:30:20/P1Y1M</Condition>
 </Conditions>
</Rule>
</Policy>
```

For the Subject which includes two Attributes in this example:

```
  <Subject>
    <Attribute AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</Attribute>
    <Attribute AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</Attribute>
  </Subject>
```

These two attributes mean the Rule requires the request should possess both of these two attributes.

However, if You put these above two Attribute into two Subject elements:

```
  <Subject AttributeId="urn:knowarc:x509:identity">/O=KnowARC/OU=UiO/CN=Physicist</Subject>
  <Subject AttributeId="urn:knowarc:voms:attribute>knowarc:atlasuser</Subject>
```

Then it means the Rule requires the request to possess at least one of these two attributes.

For the xml-attribute "Type" and "AttributeId", the explaination for Request example also applies here.

The "Function" xml-attribute is for distinguishing different comparison algorithm when comparing these two xml-node value. If Function is absent, "equal" will be used as default.

3.7. Rule Elements Matching

For elements without attributes those elements have:

- Kind specified by AttributeId XML attribute. There is no default.
- Matching algorithm specified by Id XML attribute. By default string-equal matching is used.
- Content

Example: *LEMON tree*

***Kind**: tree*

***Matching algorithm**: default*

***Content**: LEMON*

Matching procedure consists of following steps:

- Kinds are compared using simple string equal matching. If those do not match then result is INDETERMINATE.

- Matching algorithm is used to compare content of elements. Result is either MATCH or NO_MATCH according to matching algorithm.

Each element on the RequestItem must satisfy corresponding element in Rule.In detail, for Subjects element under Rule, if there is at least one Subject (with one Attribute or a few Attribute) which is matched by a Subject on this RequestItem, we say this Subjects is matched by the RequestItem; and also the same for the other elements (Actions, Resources, Conditions).

For elements with multiple Attribute sub-elements the way to judging whether elements match is if and only if all of the Attribute under the Rule have matching Attributes at RequestItem side.

Example of the Subject with three Attributes:

***Subject**:*

***SubjectAttribute**: name is ALICE*

***SubjectAttribute**: age is YOUNG*

***SubjectAttribute**: gender is GIRL*

In XML that is:

```
<Subject>
 <Attribute AttributeId="name">Alice</Attribute>
 <Attribute AttributeId="age>YOUNG</Attribute>
 <Attribute AttributeId="gender>GIRL</Attribute>
</Subject>
```

That requires the Subject in the RequestItem to possess at least these three Attributes.

```
<RequestItem>
 <Subject>
  <Attribute AttributeId="name">Alice</Attribute>
  <Attribute AttributeId="age">YOUNG</Attribute>
  <Attribute AttributeId="gender">GIRL</Attribute>
  <!--Some other Attribute-->
 </Subject>
</RequestItem>
```

The above example shows that the Subject in the RequestItem "MATCH" one Subject on the Rule side.

If the Subject in the RequestItem is like this:

```
<Subject>
 <Attribute AttributeId="name">Alice</Attribute>
 <Attribute AttributeId="age>YOUNG</Attribute>
 <Attribute AttributeId="from">OSLO</Attribute>
 <!--Some other Attribute, buts not a "gender"-->
</Subject>
```

Then evaluator will produce INDETERMINATE as the match-making result of this Subject.

If the Subject in the RequestItem is like this:

```
<Subject>
 <Attribute AttributeId="name">Bob</Attribute>
 <Attribute AttributeId="age>YOUNG</Attribute>
```

```
<Attribute AttributeId="gender">BOY</Attribute>
<!--Some other Attribute-->
</Subject>
```

Then evaluator will give NO_MATCH as the match-making result of this Subject.

Finally if and only if all of the elemens (Subjects, Actions, Resources, Conditions) which are not empty under the Rule have been matched (gets MATCH) to the RequestItem, then the whole Rule is considered to be matched (produces MATCH result). MATCH is then mapped to final evaluation result depending on the specified Effect. If Effect is set to Deny then DENY decision will be produced for this Rule; if Effect is Permit then PERMIT.

Otherwise if any of the element (Subjects, Actions, Resources, Conditions) of RequestItem got INDETERMINATE decision then the INDETERMINATE decision will be made for this Rule.

Otherwise the NOT_APPLICABLE decision will be made for this Rule. In other words that means at least one of the elements of this Rule got NO_MATCH and the other elements got MATCH.

3.8. Interface for using the policy evaluation engine

For making usage of policy evaluation engine more convenient basic *Evaluator* class is complemented by additional interfaces. Below are examples of steps needed to carry out policy evaluation and corresponding helper interfaces.

a) Create the policy evaluation object:

```
// Create object which provides an interface
// for loading other objects
ArcSec::EvaluatorLoader eval_loader;
//Load the Evaluator
ArcSec::Evaluator* eval = NULL;
// Define name of policy evaluator.
// This one is for evaluation ARC policies
std::string evaluator = "arc.evaluator";
eval = eval_loader.getEvaluator(evaluator);
```

b) Create the policy object:

```
ArcSec::Policy* policy = NULL;
// Define type of policy - ARC policy in this case
std::string policyclassname = "arc.policy";
// Define source from which policy to be taken
ArcSec::SourceFile policy_source("Policy_Example.xml");
// Load and parse policy
policy = eval_loader.getPolicy(policyclassname, policy_source);
```

c) Create the request:

```
ArcSec::Request* request = NULL;
// Define type of request - ARC request in this case
std::string requestclassname = "arc.request";
// Define source from which request to be taken
ArcSec::SourceFile request_source("Request.xml");
// Load and parse request
request = eval_loader.getRequest(requestclassname, request_source);
```

d) Add the policy into *Evaluator* object:

```
eval->addPolicy(policy);
```

e) Evaluate the request object:

```
ArcSec::Response *resp = NULL;
resp = eval->evaluate(request);
```

The steps d) and e) can also be replaced by:

```
resp = eval->evaluate(request, policy);
```

The *Evalutor::evaluate()* method can also be feed up with both *Policy*/*Request* objects and their sources in any combination. See example code at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/testinterface.cpp for more details about usage of the interface.

The description of mentioned classes and their methods are avaialble in API document at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/doc/KnowARC-API.pdf?format=raw .

4. Policy Decision Service

Policy decision service is a service implementation which contains the functionality of ArcPDP. It will accept the soap request containing policy decision request and return soap response containing policy decision response.

The WSDL description of policy decision service is available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/pdp/pdp.wsdl . It's configuration is presented in Section 7.5.

5. Security Attributes

5.1. Infrastructure

Security Attributes represent security related information inside HED framework and store information representing various aspects needed to perform authorization decison - identity of client, requested action, targeted resource, constraint policies.

Each kind of Security Attribute is represented by own class inherited from parent SecAttr class <arc/message/SecAttr.h>. Each Security Attribute stores it's information in internal format and is capable to export it to one of predefined formats using Export() method. Currently only supported format is ARC Policy/Request XML document described in Section 7.1and 7.2

Collectors of Security Attributes instantiate corresponding classes and link them to Secuirity Attributes containers - MessageAuth <arc/message/MessageAuth.h> and MessageAuthContext <arc/message/Message.h> storing collected attributes per request and per session correspondingly. Each attribute is assigned a name. Current implementations of Security Attributes Collectors are either integrated into existing MCCs or implemented as separate SecHandler plugins. See Section 5.2for available Collectors and corresponding Security Attributes.

Note for service developers: Services may implement own authorization algorithms. But they may use Security Atributes as well by providing instances of classes inherited from SecAttr and running them through either configured or hardcoded processors/PDPs.

Processors of Security Attributes are implemented as Policy Decision Point components. Currently there are 2 PDP components available:

- Arc PDP makes use of Security Attributes containing identities of client, resource and requested action. It evaluates either all or selected set of attributes against specified Policy documents thus making it possible to enforce policies defined/selected by service providers.
- Delegation PDP is described below in Section 6.3

5.2. Available collectors

Here Security Attribute collectors distributed as part of the ARC1 are described except those used for Delegation Restrictions. Those are described in Section 6.2

5.2.1. TCP

Information is collected inside TCP MCC. The Security Attribute is stored under name 'TCP' and exports ARC Request with following attributes:

Element	*AttributeId*	*Content*
Resource	http://www.nordugrid.org/schemas/policy-arc/types/localendpoint	service_ip[:service_port]
SubjectAttribute	http://www.nordugrid.org/schemas/policy-arc/types/remoteendpoint	client_ip[:client_port]

Table 1. Security Attributes collected at TCP MCC

5.2.2. TLS

Information is collected inside TLS MCC. Generated Security Attribute class is stored under name 'TLS' and exports ARC Request with following attributes:

Element	*AttributeId*	*Content*
SubjectAttribute	http://www.nordugrid.org/schemas/policy-arc/types/tls/ca	signer of first certificate in client's chain
SubjectAttribute	http://www.nordugrid.org/schemas/policy-arc/types/tls/chain	Subject of certificate in client's chain - multiple items
SubjectAttribute	http://www.nordugrid.org/schemas/policy-arc/types/tls/subject	Subject of last certificate in client's chain
SubjectAttribute	http://www.nordugrid.org/schemas/policy-arc/types/tls/identity	Subject of last non-proxy certificate in client's chain

Table 2. Security Attributes collected at TLS MCC

5.2.3. HTTP

Information is collected inside HTTP MCC. The Security Attribute is stored under name 'HTTP' and exports ARC Request with following attributes:

Element	*AttributeId*	*Content*
Resource	http://www.nordugrid.org/schemas/policy-arc/types/http/path	HTTP path without host and port part
Action	http://www.nordugrid.org/schemas/policy-arc/types/http/method	HTTP method

Table 3. Security Attributes collected at HTTP MCC

5.2.4. SOAP

Information is collected inside SOAP MCC. Security Attribute is stored under name 'SOAP' and exports ARC Request with following attributes:

Element	*AttributeId*	*Content*
Resource	http://www.nordugrid.org/schemas/policy-arc/types/soap/endpoint	To element of WS-Addressing structure
Action	http://www.nordugrid.org/schemas/policy-arc/types/soap/operation	SOAP top level element name without namespace prefix
Context	http://www.nordugrid.org/schemas/policy-arc/types/soap/namespace	Namespace of SOAP top level element

Table 4. Security Attributes collected at SOAP MCC

6. Delegation Restrictions

6.1. Delegation Architecture

In current implementation delegation is achieved through Identity Delegation implemented using X509 Proxy Certificates as defined in RFC 3820. Client wishing to allow service to act on it's behalf provides Proxy Certificate to the service using Web Service based Delegation interface described in Section 6.4

For limiting the scope of delegated credentials along with usually used time constraints it is possible to attach Policy document to Proxy Certificate. According to RFC 3820 Policy is stored in ProxyPolicy extension. In order not to introduce new type of object Policy is assigned id-ppl-anyLanguage identifier. RFC 3820 allows any octet string associated with such object. We are using textual representation of ARC Policy XML document.

Each deployment implementing Delegation Restrictions must use dedicated Security Handler plugin (see section 5.1) to collect all Policy documents from Proxy Certificates used for establishing secure connection. Then those documents must be processed by dedicated Policy Decision Point plugin (see section 2.3) to make a final decision based on collected Policies and various information about client's identity and requested operation. Service or MCC chain supporting Delegation Restrictions must accept negative decision of this PDP as final and do not override it with any other decision based on other policies.

6.2. Delegation Collector

This Security Attribute is collected by dedicated Security Handler plugin named "delegation.collector" avaialble as part of the ARC1 distribution. It extracts policy document stored inside X509 certificate proxy extension as defined in RFC3820 and described in Section 6.1All proxy certificates in a chain provided by client are examined and all available policies are extracted.

Extracted content is converted into XML document. Then document is checked to be of ARC Policy kind. If policy is not recognized as ARC Policy procedure fails and that causes failure of communication.

Proxy certificates with id-ppl-inheritAll [5. RFC3820. http://www.faqs.org/rfcs/rfc3820.html] property are passed through and no policy document is generated for them. Proxies with other type of policies including id-ppl-independent are not accepted and generate immediate failure.

6.3. Delegation PDP

The Delegation PDP is similar to the Arc PDP described above except that it takes it's Policy documents directly from Security Attributes. Differently from Arc PDP it is meant to be used for enforcing policies defined by client.

6.4. Delegation interface

Delegation interface in the ARC1 is implemented using Web Service approach. Each ARC1 service wishing to accept delegated credentials implements this interface. Here is how delegation procedure works:

- Step 1
    - Client contacts service requesting operation DelegateCredentialsInit. This operation has no arguments.
    - Service responds with DelegateCredentialsInitResponse message with element TokenRequest. That element contains credentials request generated by service in Value. Type of request is defined by attribute Format. Currently only supported format is x509. Along with Value service provides identifier Id which is used in second step.
- Step 2

- Client requests UpdateCredentials operation with DelegatedToken argument. This element contains Value with serialized delegated credentials and Id which links it to first step. Delegated token element may also contain multiple Reference elements. Reference refers to the object which these credentials should be applied to in a way specific to the service. The DelegatedToken element may also be used for delegating credentials when Step 2 is combined with other operations on service in service specific way.
- Service responds with empty UpdateCredentialsResponse message.

7. Schemas, descriptions and examples

7.1. Authorization Policy

XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/Policy.xsd .

7.2. Authorization Request

XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/Request.xsd .

7.3. Authorization Response

XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/Response.xsd .

7.4. Interface of policy decision service

WSDL with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/pdp/pdp.wsdl .

7.5. Configuration of PDP service

XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/pdp/pdp.xsd .

Below is an example configuration of PDP service which can evaluate ARC Request against ARC Policy stored in local file.

```
<Service name="pdp.service" id="pdp_service">
<!--The element <Evaluator/>, <Policy/> and <Request/> configuration
       are supposed to be used to load object; element <PolicyStore/> is
       supposed to be used to get the location of policy-->
   <pdp:PDPConfig>
       <pdp:PolicyStore>
           <Location Type="file">Policy_Example.xml</Location>
           <!-- other policy location-->
       </pdp:PolicyStore>
       <pdp:Evaluator name="arc.evaluator" />
       <pdp:Policy name="arc.policy" />
       <pdp:Request name="arc.request" />
   </pdp:PDPConfig>
</Service>
```

See Section 7.7for the explanation of ARC Policy.

7.6. SimpleList PDP configuration and Policy Example

XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/simplelistpdp/SimpleListPDP.

xsd .

Below is an example configuration of SimpleList PDP inside “echo” service.

```
<Service name="echo" id="echo">
  <SecHandler name="arc.authz" id="authz" event="incoming">
    <PDP name="simplelist.pdp" location="simplelist"/>
  </SecHandler>
  <echo:prefix>[ </echo:prefix>
  <echo:suffix> ]</echo:suffix>
</Service>
```

The attribute “name” of <PDP/> is critical for loading the object. Specifically, the name “simplelist.pdp” is for loading the SimpleList PDP object.

The policy file “simplelist” is a local file which contains the list of X509 subjects of authorized entities. It the peer certificate is proxy certificate, the identity in this list should only include the original DN of users's certificate.

For example content of *simplelist* file may look like this:

/C=NO/O=UiO/CN=test1

/C=NO/O=UiO/CN=test2

7.7. Arc PDP configuration and Policy Example

XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/arcpdp/ArcPDP.xsd .

Below is an example of configuration of Arc PDP inside “echo” service.

```
<Service name="echo" id="echo">
  <SecHandler name="arc.authz" id="authz" event="incoming">
    <PDP name="arc.pdp">
      <PolicyStore>
        <Location type="file">Policy_Example.xml</Location>
        <!--other policy location-->
      </PolicyStore>
    </PDP>
  </SecHandler>
  <echo:prefix>[ </echo:prefix>
  <echo:suffix> ]</echo:suffix>
</Service>
```

The name “arc.pdp” is for loading the ArcPDP object.

There could be a few policy files under <PolicyStore/>. The request will be checked against all of the policies.

There is an example policy for echo service below. See Section 7.1for the policy schema. The example policy is made of following elements:

1. Line 14 defines resource being protected. In this it is everything located under HTTP path “/Echo”.

2. Lines 17 and 18 define allowed HTTP operations to be “POST” and “GET”. Line 19 also defines SOAP operation “echo” to be applied to service at path defined above.

3. Lines 10 and 9 require the requester to present X509 certificate with specified identity and signed by specified Certification Authority.

4. No <Conditions/> defined.

5. Line 3 defines that if and only if all of the above constraints have been satisfied by requester, the <Rule/> evaluates to Permit decision.

The Secuirity Attributes used by Arc PDP are collected by different MCCs. It is possible for service to collect some application-specific attributes by implementing class inherited from SecAtt. And that should be the task of application developer.

Administrator of service can configure Authorization SecHandler - arc.authz - for each MCC and Service and define reasonable and meaningful policy. While defining policy the administrator must take into account that the attributes defined in the policy should be already collected by previous components in a chain. For instance, policy with AttributeId "http://www.nordugrid.org/schemas/policy-arc/types/http/path" should not be configured inside SecHandler attached to MCCTLS.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <Policy    xmlns="http://www.nordugrid.org/schemas/policy-arc"    PolicyId="sm-
    example:arcpdppolicy" CombiningAlg="Deny-Overrides">
3.     <Rule Effect="Permit">
4.       <Description>
5.         Example policy for echo service
6.       </Description>
7.        <Subjects>
8.           <Subject>
9.                 <Attribute AttributeId="http://www.nordugrid.org/schemas/policy-
    arc/types/tls/ca" Type="string">/C=NO/ST=Oslo/O=UiO/CN=CA</Attribute>
10.                <Attribute AttributeId="http://www.nordugrid.org/schemas/policy-
    arc/types/tls/identity" Type="string">/C=NO/ST=Oslo/O=UiO/CN=test</Attribute>
11.          </Subject>
12.       </Subjects>
13.       <Resources>
14.                <Resource  AttributeId="http://www.nordugrid.org/schemas/policy-
    arc/types/http/path" Type="string">/Echo</Resource>
15.       </Resources>
16.       <Actions>
17.                  <Action  AttributeId="http://www.nordugrid.org/schemas/policy-
    arc/types/http/method" Type="string">POST</Action>
18.                  <Action  AttributeId="http://www.nordugrid.org/schemas/policy-
    arc/types/http/method" Type="string">GET</Action>
19.                  <Action  AttributeId="http://www.nordugrid.org/schemas/policy-
    arc/types/soap/operation" Type="string">echo</Action>
20.       </Actions>
21.       <Conditions/>
22.    </Rule>
23. </Policy>
```

7.8. PDP Service Invoker configuration

Configuration XML schema with comments is available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/pdpserviceinvoker/ArcPDPServiceInvoker.xsd .

Below is an example of configuration of PDP Service Invoker inside "echo" service.

```
<Service name="echo" id="echo">
  <SecHandler name="arc.authz" id="authz" event="incoming">
    <!--Remote pdp service invoking-->
      <PDP name="pdpservice.invoker">
```

```
        <ServiceEndpoint>https://127.0.0.1:60001/pdp.service</ServiceEndpoint>
        <KeyPath>./key.pem</KeyPath>
        <CertificatePath>./cert.pem</CertificatePath>
        <CACertificatePath>./ca.pem</CACertificatePath>
      </PDP>
  </SecHandler>
  <next id="echo"/>
  <echo:prefix>[ </echo:prefix>
  <echo:suffix> ]</echo:suffix>
</Service>
```

The name “pdpservice.invoker” defines the PDP Service Invoker object.

The PDP Service Invoker is a client of PDP Service. The configuration options include endpoint of service and credentials to be used for establishing secure connection.

7.9. Delegation PDP configuration

Configuration XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/delegationsh/DelegationPDP.xsd .

Below is an example of configuration of Delegation PDP inside “echo” service.

```
<Service name="echo" id="echo">
  <SecHandler name="arc.authz" id="authz" event="incoming">
    <PDP name="delegation.pdp"/>
  </SecHandler>
  <next id="echo"/>
  <echo:prefix>[ </echo:prefix>
  <echo:suffix> ]</echo:suffix>
</Service>
```

For Delegation PDP, no specific configuration is needed. We only need to switch it on by adding <PDP name="delegation.pdp"/> under <SecHandler/>

7.10. Delegation SecHandler Configuration

Below is an example of configuration of Delegation SecHandler inside TLS MCC component.

```
        <Component name="tls.service" id="tls"> <next id="http"/>
           <tls:KeyPath>./key.pem</tls:KeyPath>
           <tls:CertificatePath>./cert.pem</tls:CertificatePath>
           <tls:CACertificatePath>./ca.pem</tls:CACertificatePath>
           <!--delegation.collector must be inside tls MCC-->
         <SecHandler name="delegation.collector"
             id="delegation" event="incoming"></SecHandler>
        </Component>
```

Current implementation of Delegation SecHandler must be attached to TLS MCC.

8. Web Service Security Support

8.1. UsernameToken SecHandler configuration

Configuration XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/usernametokensh/UsernameTokenSH.xsd .

Below is an example of configuration of UsernameToken SecHandler inside MCCSOAP component.

```
<Component name="soap.service" id="soap">
    <next id="echo"/>
                    <SecHandler  name="usernametoken.handler"  id="usernametoken"
event="incoming">
        <Process>extract</Process>
        <PasswordSource>password.txt</PasswordSource>
    </SecHandler>
</Component>
```

UsernameToken SecHandler must be configured under SOAP MCC.

8.2. X509Token SecHandler configuration

Configuration XML schema with comments available at http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/pdc/x509tokensh/X509TokenSH.xsd.

Below is an example of configuration of X509Token SecHandler inside MCCSOAP component.

```
<Component name="soap.service" id="soap">
    <next id="echo"/>
     <SecHandler name="x509token.handler" id="x509token" event="incoming">

        <Process>extract</Process>

        <CACertificatePath>ca.pem</CACertificatePath>

     </SecHandler>

</Component>
```

X509Token SecHandler must be configured under SOAP MCC.

9. Using Shibboleth IdP for Authentication and Attribute-based Authorization

In grid community, GSI (grid security infrastructure) is the de-facto standard about transport level communication for the legacy grid solution, which implements some enhancement (such as delegation) based on standard SSL3.0/TLS1.0. In ARC1, besides that GSI is supported for talking with external grid services which are based on GSI, the standard TLS/SSL is also supported.

No matter standard SSL/TLS or GSI is used, SSL/TLS based mutual authentication applies for both of them, and is the default configuration for grid deployment; and X.509 certificate is required for both of the client and service sides. X.509 certificates is issued by certificate authorities (CA), and then CAs constitute trust federation and guarantee two different X.509 certificates from different CAs can accomplish authentication to each other. So if a user would access grid system, he/she should own a X.509 certificate which is issued by a CA that is trusted by other's entity in the grid system.

AAI (Authentication and Authorization Infrastructure) is a solution for the authentication and authorization in inter-organization resource sharing, such as electronic resource sharing between libraries, etc. AAI implicitly applies to community or institutional based authentication where users from different home communities need to get resources from other communities by using some federation mechanism. Unlike the X.509 based authentication solution in current Grid systems, AAI does not require users to provide X.509 certificate, instead, it can support different types of authentication, such as username/password authentication, IP address authentication, etc. There are several implementations of AAI, among which Shibboleth is one implementation which has been

widely deployed.

Shibboleth provides cross-domain single sign-on and attribute-based authorization while preserving user privacy. It is based on the OASIS Security Assertion Markup Language (SAML), specifically, the new version of Shibboleth supports SAML 2.0 specification. For authentication, the main SAML profile that Shibboleth implements is the SAML2.0 web browser SSO profile, which defines two functional components, an Identity Provider and a Service Provider. The Identity Provider (IdP) is responsible for creating, maintaining, and managing user identity, while the Service Provider (SP) is responsible for controlling access to services and resources by using the SAML assertion produced and issued by IdP upon request. In order to discover which home community does a user come from, Shibboleth specifies an optional third component called "Where Are You From?" (WAYF) service to aid in the process of IdP discovery, and this IdP discovery process is also standardized and defined in SAML 2.0 specification and called as "Identity Profile Discovery Profile".

The SAML2.0 web browser SSO profile is utilized for the authentication in ARC middleware. But since the SSO profile is primarily supposed to protect Web applications and provide authentication for Web users, some external code on the client and service side is implemented to integrate the SSO profile. On the client side, apart from the client interface for writing Web Service client, the user agent functionality of the Web browser is implemented in order to mimic its behavior, such as HTTP redirection and cookie processing. In fact, implementation of the user agent is also based on the client interface of ARC, specifically, the HTTPs client interface, since the client interface of ARC can support different protocols which are incarnated by different MCCs. The client developers who would use SAML2.0 SSO profile should call the user agent interface and then the Web Service client interface. On the service side, the Service Provider functionality (based on the HTTP MCC configured together with TLS MCC) is implemented, which is called SP Service. For Identity Provider, the Shibboleth IdP implementation is used. Figure 6 shows the process of SAML2.0 SSO integrated in ARC client and service.

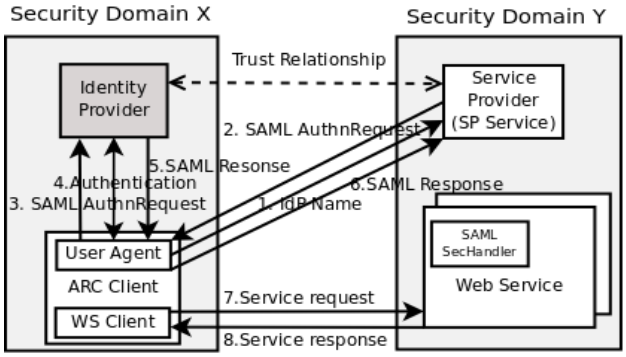

Figure 6. SAML2.0 SSO profile in ARC

The steps shows in Figure 6 are described as follows:

1. The client uses the user agent interface to launch a HTTP request including the IdP name (to which the user belongs) to the service side. The endpoint of the SP (Service Provider) service is the same as that of the other target services, except the last part of the endpoint is "saml2sp" which is specific for pointing to the SP Service. Note that we use Identity Provider (IdP) name here to simplify the IdP discovery process in order to avoid the IdP discovery process, because we suppose that the user who would access the target services should better know where is he from initially.
2. The SP Service searches the metadata (we use the same metadata format as defined in Shibboleth) and gets the location of the single sign-on service (hosted in IdP) and also the location of assertion consuming service (hosted in this SP itself) in order to compose the SAML <samlp:AuthnRequest> message. Then SP Service issues this <samlp:AuthnRequest>

message by using its own X.509 certificate (note that in the SAML SSO profile, X.509 certificates are still needed for IdP and SP) and sends back to user agent.

3. User agent sends the <samlp:AuthnRequest> message to the Identity Provider.
4. Identity Provider requires an act of authentication. The authentication mechanism is outside of the SAML2.0 SSO profile. Shibboleth IdP implementation chooses some login handlers for authentication. The current user agent implementation is compatible with the Username/Password login handler of Shibboleth IdP. Through the HTTP protocol, the user agent will feed IdP with the username/password which has been given by the caller of user agent interface.
5. Once the authentication has been succeeded, the IdP issues a SAML response including an encrypted (encrypted by destination SP's public key) SAML assertion, and then this SAML response will be delivered by the user agent to the Service Provider.
6. The SP Service verifies and checks the SAML response, decrypts and stores the SAML assertion into session/connection context. The SAML assertion includes the <saml:AuthnStatement> and <saml:AttributeStatement>.
7. The WS client launches the Grid/Web Service request via the same connection as the one which is used by user agent to contact SP Service.
8. The Grid/Web Service checks the <saml:AuthnStatement> from the session context to see if the session is still valid through the SecHandler called "SAML SecHandler". If valid, service handles the service processing and returns the response to WS client. Note that service requires that WS client is from the same connection as the one on which user agent contact SP service, in order to guarantee that the validity of SSO profile result effects the WS client/ Web Service interaction.

The SP service and other functional service(s) are hosted by the same container, and they use the same X.509 credential. The client authentication is switched off, so that client doesn't need to use any X.509 credential. Only the trusted certificates (CA certificates for both SP and IdP) need to be configured for the client side, so that SP and IdP can authenticate themselves to the client. As required by the SAML2.0 profile, the SP and IdP should have trust relationship to each other.

One benefit of the SAML2.0 SSO profile that is worth mentioning is: the Identity Provider could cache the authentication result through session management once the user agent has succeeded to authenticate; then for a short period this authentication result is valid so that the user agent doesn't need to feed IdP with user's username and password the next time (if this point of time is not out of the scope of valid period) it authenticates against IdP. So user (or the client on behalf of this user) can travel across multiple security domains with only providing his name and password once, which is the characteristic of single sign-on.

Since the Shibboleth implementation of SAML is standard-compliant and widely deployed, the solution implemented in ARC can easily interoperate with other SAML implementations with minimum change, and more importantly, this solution can succeed to utilize the widely deployed SAML implementation for authentication in Grid systems by avoiding the usage of X.509 certificate.

Moreover, even though the implementation is based on the ARC middleware, the idea can be adopted by other Grid middlewares if they only require server authentication instead of mutual authentication.

10. Short-Lived Credential Service

However, most of the widely used Grid middlewares are based on GSI while GSI requires mutual authentication. Also for Web Service based Grid solution, we cannot prevent service side from requiring client X.509 certificate. Based on the solution described in Section 9, a short lived credential service (SLCS) is implemented by which user can get a short-lived X.509 certificate without being bothered to contact any registration authority (RA) or certificate authority (CA).

The SLCS service is also a Web Service (standard Web Service implemented by using ARC service interface), and the SLCS client is a specific command-line interface (CLI) which includes the user agent and WS client. The whole process of SLCS invocation is showed in Figure 7 (from step 1 to step 8), which is the same as in Figure 6, except that step 7 and step 8 are invoked for the SLCS certificate request and response.

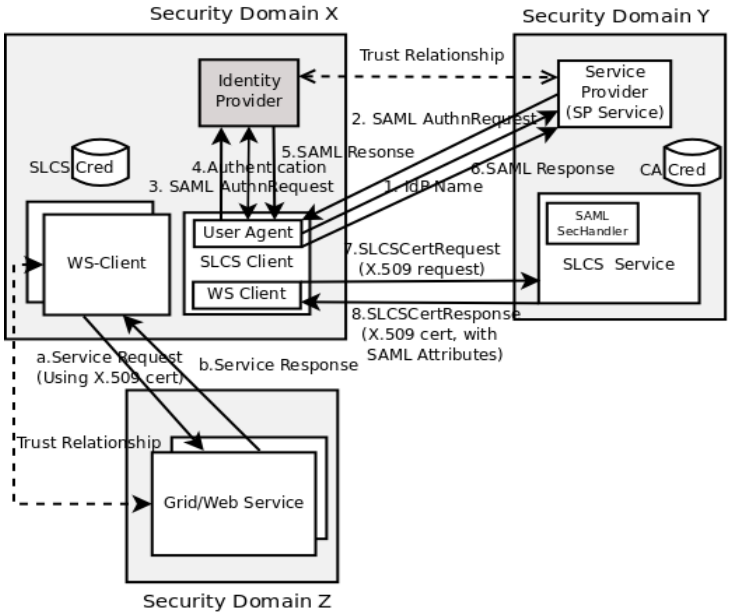

Figure 7.Short lived credential service

SLCS client generates a X.509 certificate request, launches a Web Service request which includes the certificate request; SLCS service then gets the certificate request, composes a distinguished name (DN), issues a certificate (short lived, 12 hours by default) with the SAML attribute (from the SAML2.0 SSO profile) as the X.509 certificate extension, and puts the certificate in to the Web Service response; SLCS client get the response and stores the X.509 certificate into local repository.

The CLI for the SLCS client is like this:

```
$ ./arcslcs -S https://127.0.0.1:60000/slcs -I https://idp.testshib.org/idp/shibboleth
-U myself -P myself
```

Since the lifetime of the short lived credential is normally short, it is not a must to protect the private key by a pass phrase. As illustrated in steps (a) and (b) in Figure 7, if the private key is not protected through the Web Service client, the user can use the X.509 certificate to access Grid Service or Web Service from any kind of middleware. If the private key is protected, she can use the X.509 certificate to generate a proxy certificate (by using a command-line interface utility such as grid-proxy-init , voms-proxy-init, or arcproxy), and then use the proxy certificate to access a Grid/Web Service.

It is worth mentioning that since the ARC middleware can support GSI communication by configuring the GSI MCC, together with the X.509 certificate, the Web Service client developed with the ARC Web Service client interface can interoperate with Grid Service that requires GSI communication.

It should be noticed that the process of composing the distinguished name (DN) for the certificate is a critical issue for the SLCS service. Since the Shibboleth Identity Provider uses the eduPerson schema for the definition of <saml:Attribute> in <saml:AttributeStatement>, we pick the relatively distinguishable attribute "eduPersonPrincipalName" for the DN. A typical eduPersonPrincipalName value could be alice@example.org, then the DN is "/O=knowarc/OU=example.org/CN=alice".

The obvious benefit of the SLCS service is that: If a user passes the authentication to her home

Identity Provider, she can get the X.509 credential anywhere simply by running the SLCS client command together with providing her username and password to this home IdP, and then access the Grid system conveniently. Thanks to the single sign-on characteristic of SAML2.0 SSO profile, this user doesn't not need to input her username and password in a valid period after the first time she succeeds to authenticate against her home IdP by running the SLCS client command on the same node, even if this SLCS client command is supposed to run against a few SLCS services to get a few SLCS credentials.

11. X.509 Credential Delegation Service

References

1. RFC3820- Internet X.509 Public Key Infrastracture (PKI) Proxy, http://rfc.net/rfc3820.html

2. D1.2-2 The ARC container (first prototype), http://www.knowarc.eu/documents/Knowarc_D1.2-2_07.pdf

3.Web Service Security Username Token Profile 1.1. http://www.oasis-open.org/committees/download.php/16782/wss-v1.1-spec-os-UsernameTokenProfile.pdf

4. OGSA® Basic Execution Service Version 1.0, http://www.ogf.org/documents/GFD.108.pdf

5. OASIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss