NORDUGRID-TECH-16
26/7/2013

SECURITY FRAMEWORK OF ARC

Documentation and developer's guide

Weizhong Qiang*
Aleksandr Konstantinov†

*weizhong.qiang@fys.uio.no
†aleksandr.konstantinov@fys.uio.no

Contents

1 Introduction

The security framework of the ARC 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 concerns were employed when designing this framework:

- Interoperability and standardization. In consistency with the main design concerns of the ARC middleware, interoperability and standardization is considered in security framework. For example, in terms of authentication, PKI infrastructure and X.509 proxy certificates (RFC3820 [1]) are 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, a lot of functionality is implemented as plug-ins which has well-defined interfaces, and are configurable and dynamically loadable. Since the interoperation interface between security plug-ins and hosting environment or applications is predefined, it is easy to extend the security functionality in order to support other new security capabilites.
- Backward compatibility. The GSI (Grid Security Infrastructure)[4] based mechanism has been a defacto solution for Grid security for long time alredy. Although it has drawback for compatibility reasons the security framework should include it as part of its capability.

2 Security architecture in HED: Security Handler and Policy Decision Point

2.1 Structure of Security Handler and Policy Decision Point

In the implementation of the ARC, there is a Service Container called Hosting Environment Daemon (HED) [3] which provides a hosting place for various services at application and protocol 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 Security Handlers (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 other 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 shows the structure of a MCC/Service, and the message processing sequence inside it.

The configuration of SecHandler components for an example “Echo” service is shown below. Example “Echo” service is configured to use two SecHandlers, both performing authorization. First SecHandler uses the X.509 identity of client (certificate subject) extracted from the incoming message to map it into local identity like 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 stored in local file “policy.xml”, the other will compare the X.509 identity of client extracted from the incoming message against list of identities stored locally.

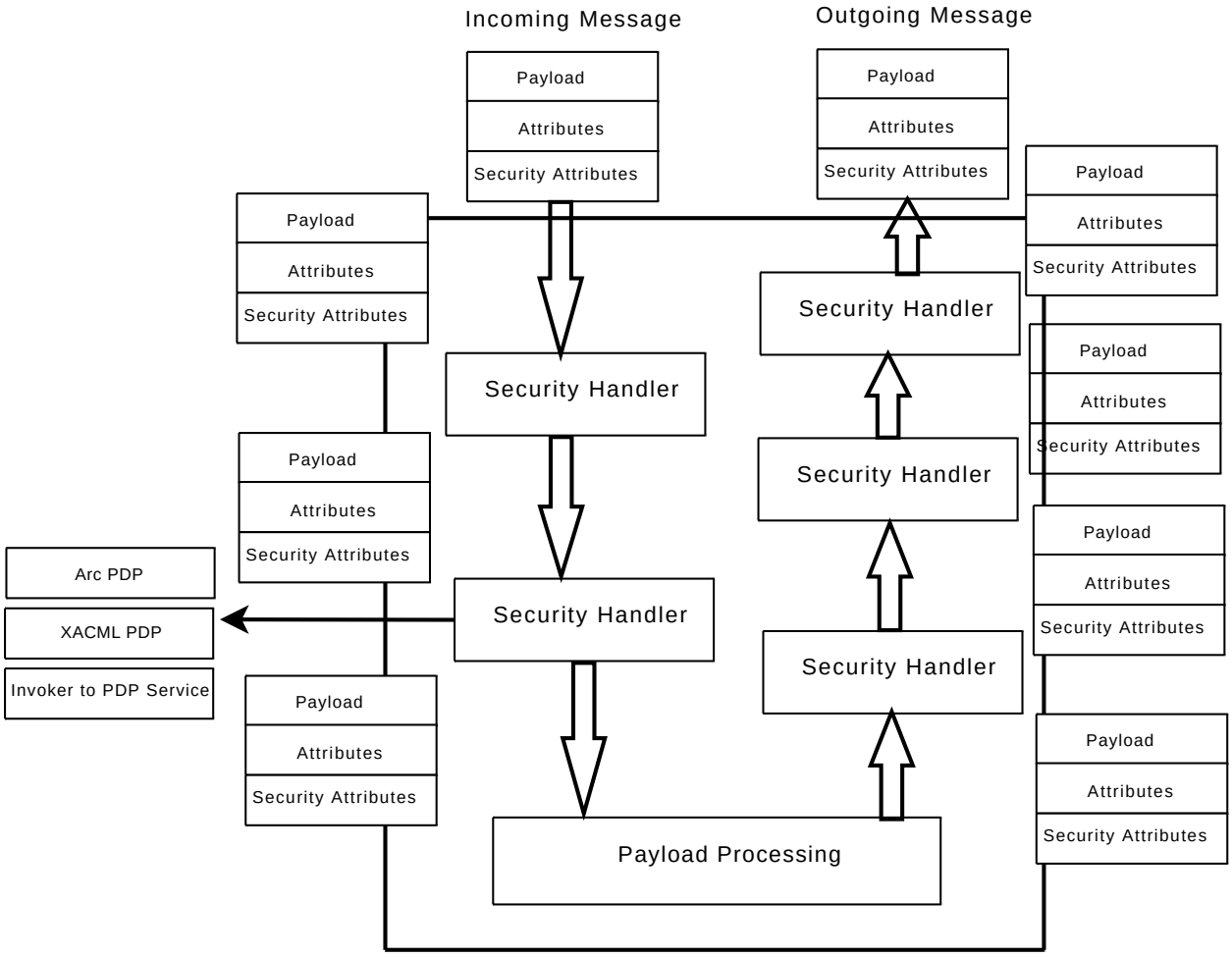

Figure 1: *There are usually two chains of SecHandlers inside the MCC or service. Each SecHandler will parse the Security Attributes which are generated by the upstream MCC/services or probably upstream SecHandlers in the same or other MCC/Service, and do message processing or authenticate or authorize 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 [8] into header part of SOAP message. The PDPs are called by the dedicated SecHandlers and are supposed to make authorization decision. In this example two local PDPs and one remote PDP service are presented, and any number of PDPs can be configured under corresponding 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>
```

2.2 Interface of SecHandler

When either MCC or Service are 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.

Each SecHandler implements one simple interface (see below), which is 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;
  };
```

Class *SecHandler* is an abstract class which includes a general interface method called *Handle* which takes Message object as argument. Any security handler implementation must inherit from class *SecHandler* and implement the interface according to the actual functionality. The method returns simple *Boolean* value, and any useful information generated during the calling of this interface should be put into the security attributes of the message, or put into the payload itself.

Currently, the ARC comes with the following security handlers implemented:

- arc.authz Authorization SecHandler The arc.authz and serves as container for the Policy Decision Point components. It is responsible for calling their interface and getting back the authorization result. Then obtained results are processed and combined decision is made. Description of configuration and examples can be found in section 8.7. Usually the Authorization SecHandler and included PDPs are used on the service side of communication channel. Although it is also possible to use them on the client side.
- 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. The obtained local identity string representaion is stored in LOCALID attribute of the message. Content of attribute is either "username"" or "username:groupname".
- delegation.collector Delegation Collector SecHandler The delegation.collector is responsible for collecting the delegation policy information from the remote proxy credential (proxy certificate compatible with RFC3820) inside the message, and putting this policy into the message security attribute for the usage of other components, such as the "delegation.pdp".
- usernametoken.handler UseranemToken SecHandler The task of the usernametoken.handler is to generate the WS-Security[8] 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.
- x509token.handler X.509 Token SecHandler This SecHandler generates and process the WS-Security[8] X.509 Token inside the header of SOAP message.
- samltoken.handler
- saml2ssoassertionconsumer.handler
- delegation.handler

2.3 Interface of PDP

Below is the definition of abstract class PDP. The implementation for example could implement method isPermitted() by composing the policy evaluation request, evaluating this request against some policy, and returning the evaluation result. Or it could compose the policy evaluation request, invoke some remote policy decision web service and return back the evaluation result.

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

Class *PDP* is an abstract class which includes a general interface method called *isPermitted* which uses Message object as argument. Any policy decision point implementation must inherit from class PDP and implement the interface according to the actual functionality. The interface method 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 ARC comes with the following PDP implementations:

- 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) stored in local repository, and return back the evaluation result. See section 3 for detailed information about request and policy schema.
- xacml.pdp XACML PDP The XACML PDP will organize the security attributes into the XACML authorization request (see: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-context-schema-os.xsd), call the policy evaluator to evaluate the request against the policy (which is in XACML format: http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-policy-schema-os.xsd) stored in local repository, and return back the evaluation result.
- delegation.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 configured delegation policy. See section 6 for 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) to 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 or XACML 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.
- allow.pdp Allow PDP This PDP always returns positive result.
- deny.pdp Deny PDP This PDP always returns negative result.

3 Policy Evaluation Engine

3.1 Design of policy evaluation engine

The ARC defines specific evaluation request and policy schema. Based on the schema definition the 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 2 and 3 respectively show the UML class diagram about the policy evaluation engine for ARC policy and XACML policy. They show 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.

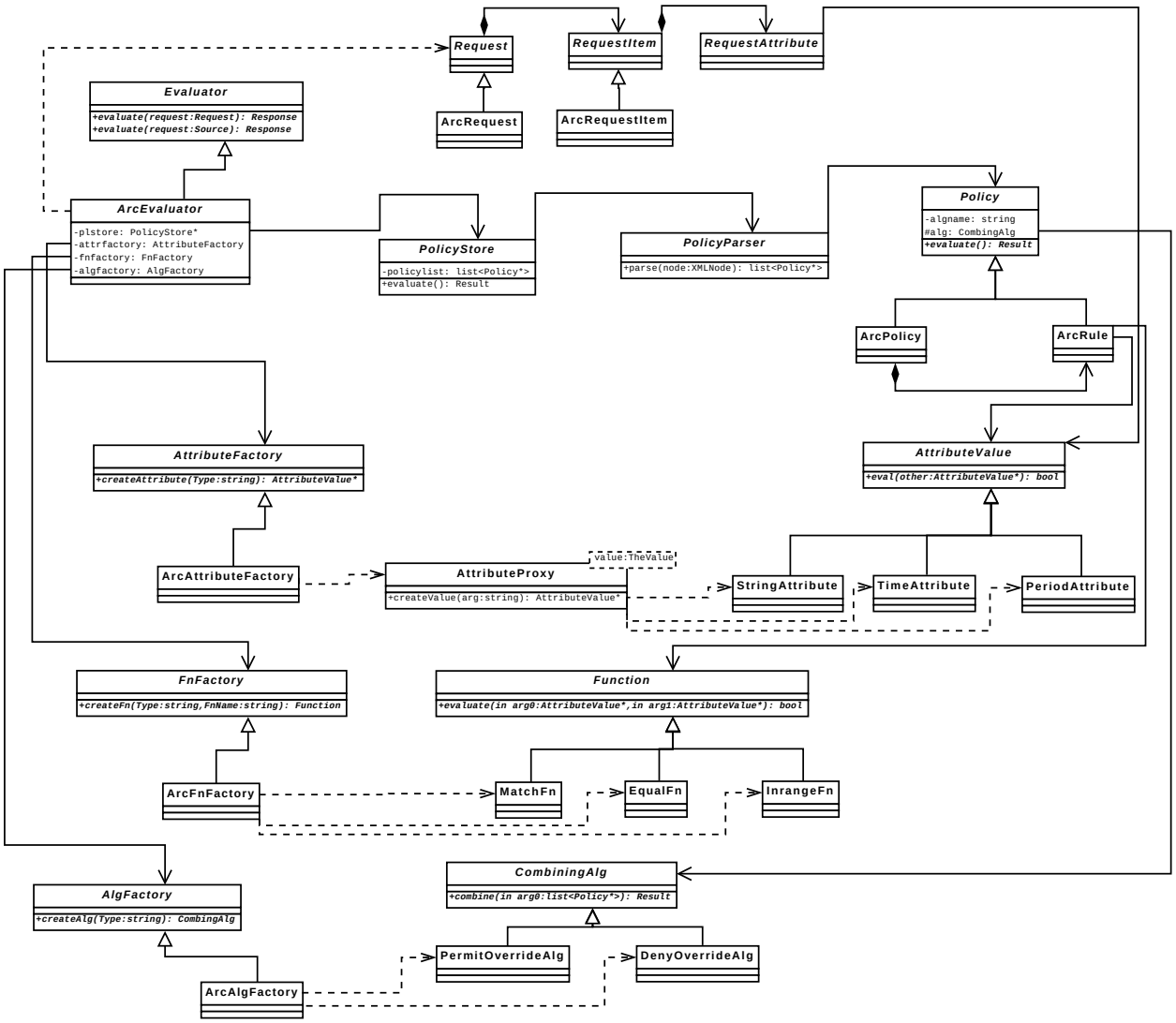

Figure 2: The UML class diagram of the classes inside policy evaluation engine that support ARC policy

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.

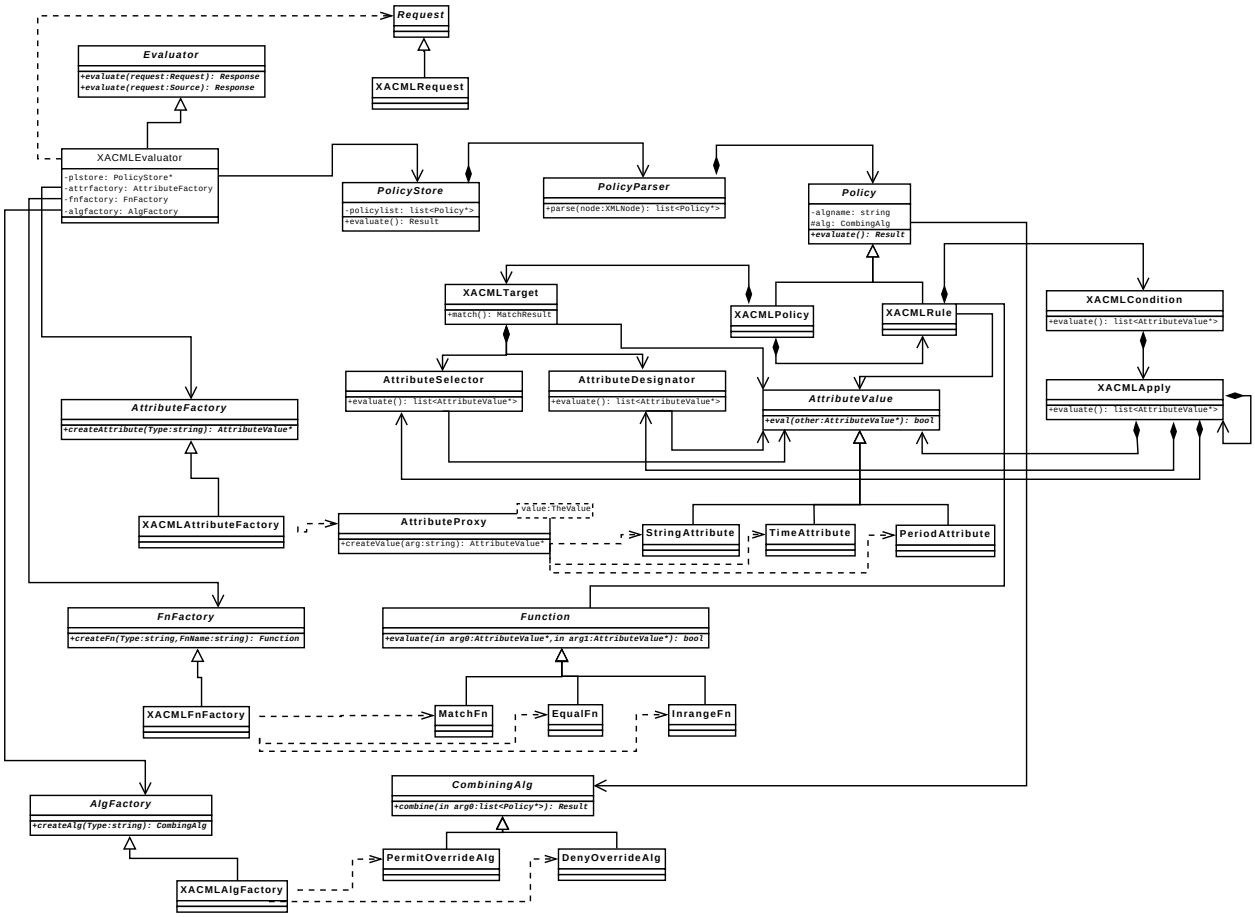

Figure 3: The UML class diagram of the classes inside policy evaluation engine that support XACML policy

3.2 Policy evaluation engine — Support of ARC policy and request

3.2.1 Schemas for ARC policy and request

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 shown 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.

3.2.2 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.

3.2.3 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.

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

3.2.4 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.

Basically the policy decision point will feed policy engine with request, get back evaluation results, and make final decision.

3.2.5 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.

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.

3.2.6 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.

3.3 Policy evaluation engine. Support for XACML policy and request

Currently, XACML specification [9] is partially supported/implemented in ARC. More specifically, except the <*Obligation*> element, other elements are supported.

http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-policy-schema-os.xsd

http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-context-schema-os.xsd

3.4 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";
// If xacml evaluation engine is used,
// std::string evaluator = "xacml.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";
// If xacml policy is used,
// std::string policyclassname = "xacml.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";
// If xacml request is used,
// std::string requestclassname = "xacml.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/shc/testinte or http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/testinterface_xacml.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 (Charon Service)

Policy decision service is a service implementing Arc PDP and XACML PDP depending on its configuration. 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/charon/charon.wsdl

Its configuration is presented in section 8.6.

5 Security Attributes. How to compose policy decision request to policy evaluation engine

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 its information in internal format and is capable to export it to one of predefined formats using *Export()* method. Currently only fully supported format is ARC Policy/Request XML document described in sections 8.1 and 8.2. It is also possible to access every item of each Security Attribute object through *get()* method using item names.

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 5.2 for 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 ARC 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 attributes described in table 1.

Table 1: Security Attributes collected at TCP MCC

Element	**Name(s)**	**AttributeId**	**Content**
Resource	LOCALIP, LOCALPORT	`http://www.nordugrid.org/schemas/policy-arc/types/localendpoint`	service_ip[:service_port]
SubjectAttribute	REMOTEIP, REMOTEPORT	`http://www.nordugrid.org/schemas/policy-arc/types/remoteendpoint`	client_ip[:client_port]

5.2.2 TLS and VOMS

Information is collected inside TLS MCC. Generated Security Attribute class is stored under name 'TLS' and exports ARC Request with attributes described in table 2.

As one can see in addition to information extractable from generic TLS/SSL session this collector also understands attribute certificates provided by VOMS server and embedded into X.509 certificate

Table 2: Security Attributes collected at TLS MCC

Element	Name(s)	AttributeId	Content
SubjectAttribute	CA	`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	SUBJECT	`http://www.nordugrid.org/schemas/policy-arc/types/tls/subject`	Subject of last certificate in client's chain
SubjectAttribute	IDENTITY	`http://www.nordugrid.org/schemas/policy-arc/types/tls/identity`	Subject of last non-proxy certificate in client's chain
SubjectAttribute	VOMS	`http://www.nordugrid.org/schemas/policy-arc/types/tls/vomsattribute`	VOMS attributes extracted from whole client's chain of certificates
	VO		VO names extracted from VOMS attributes of whole client's chain of certificates
	CERTIFICATE		PEM encoded X.509 certificated of remote peer
	CERTIFICATECHAIN		PEM encoded chain of X.509 issuers of remote peer certificate
Resource	LOCALSUBJECT	`http://www.nordugrid.org/schemas/policy-arc/types/tls/hostidentity`	Subject of certificate of local peer

The VOMS attributes are presented in format similar to VOMS FQAN with slight modifications. Differently from FQAN all values are prepended with their names like VO and Group. Missing elements are not reported. All FQANs which define groups also have VO prepended. Examples of VOMS attributes look like:

/VO=knowarc.eu/Group=knowarc.eu/Role=admin

/VO=knowarc.eu/Group=knowarc.eu/Group=testers

Each set of attributes is accompanied by identifier of service which provided those attributes. It is made of *voname* element with name of VO followed by optional element *hostname* with hostname and port of service. Here is an example:

/voname=knowarc.eu/hostname=arthur.hep.lu.se:15001

If VOMS extensions contain user definable attributes those are presented together with the information of their grantor. They consist of *voname* and *hostname* elements presented above (if *hostname* is missing it is assigned string value NULL) followed by user attribute. Its pattern is *qualifier:name=value*. Here qualifier acts as namespace and is usually same as VO name. Below is an example of such attribute:

/voname=knowarc.eu/hostname=arthur.hep.lu.se:15001/knowarc.eu:UniqueKnowarcAttribute=False

Configuration of the TLS MCC is described in section 8.5.

5.2.3 HTTP

Information is collected inside HTTP MCC. The Security Attribute is stored under name 'HTTP' and exports ARC Request with attributes described in table 3.

Table 3: Security Attributes collected at HTTP MCC

Element	Name(s)	AttributeId	Content
Resource	OBJECT	`http://www.nordugrid.org/schemas/policy-arc/types/http/path`	HTTP path without host and port part
Action	ACTION	`http://www.nordugrid.org/schemas/policy-arc/types/http/method`	HTTP method

5.2.4 SOAP

Information is collected inside SOAP MCC. Security Attribute is stored under name 'SOAP' and exports ARC Request with attributes described in table 4.

Table 4: Security Attributes collected at SOAP MCC

Element	Name(s)	AttributeId	Content
Resource	OBJECT	`http://www.nordugrid.org/schemas/policy-arc/types/soap/endpoint`	To element of WS-Addressing structure
Action	ACTION	`http://www.nordugrid.org/schemas/policy-arc/types/soap/operation`	SOAP top level element name without namespace prefix
Context	CONTEXT	`http://www.nordugrid.org/schemas/policy-arc/types/soap/namespace`	Namespace of SOAP top level element

5.2.5 ARC Legacy (Authorization Groups)

Information is collected inside Legacy SecHandler MCC. The Security Attribute is stored under name 'ARCLEGACY'. Currently this object does not support Export() method. Instead it provides access to collected information through get(). Collected are names of matching authorization groups and VOs as described in [6] after processing configuration file.

Table 5: Security Attributes collected by ARC Legacy SecHandler

Element	Name(s)	AttributeId	Content
	GROUP		Multiple items contain name of matching group each
	VO		Multiple items contain name of matching VO each

6 Delegation

6.1 Delegation Architecture

In current implementation delegation is achieved through Identity Delegation implemented using X.509 Proxy Certificates as defined in RFC 3820 [1]. Client wishing to allow service to act on its 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 6.2) 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.

To have Delegation Restriction working their processing must be enabled in all participating clients and services. Because Delegated Restrictions are marked as critical extension in X.509 proxy certificate any service which does not support such extension will fail to autheticate client presenting such certificate.

6.2 Delegation Collector

This Security Attribute is collected by dedicated Security Handler plugin named "delegation.collector" avaialble as part of the ARC distribution. It extracts policy document stored inside X.509 certificate proxy extension as defined in RFC3820 and described in section 6.1. All proxy certificates in a chain provided by client are examined and all available policies are extracted. Configuration of Delegation SecHandler is described in section 8.12.

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 [1] 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 its Policy documents directly from Security Attributes. Differently from Arc PDP it is meant to be used for enforcing policies defined by client. Configuration of Delegation PDP is described in section 8.11.

6.4 Delegation Interface

Delegation interface in the ARC is implemented using Web Service approach. Each ARC service wishing to act on behalf of client identity implements this interface in order to accept delegated credentials. Here is how delegation procedure works (also shown in figure 4 and figure 5) :

- *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

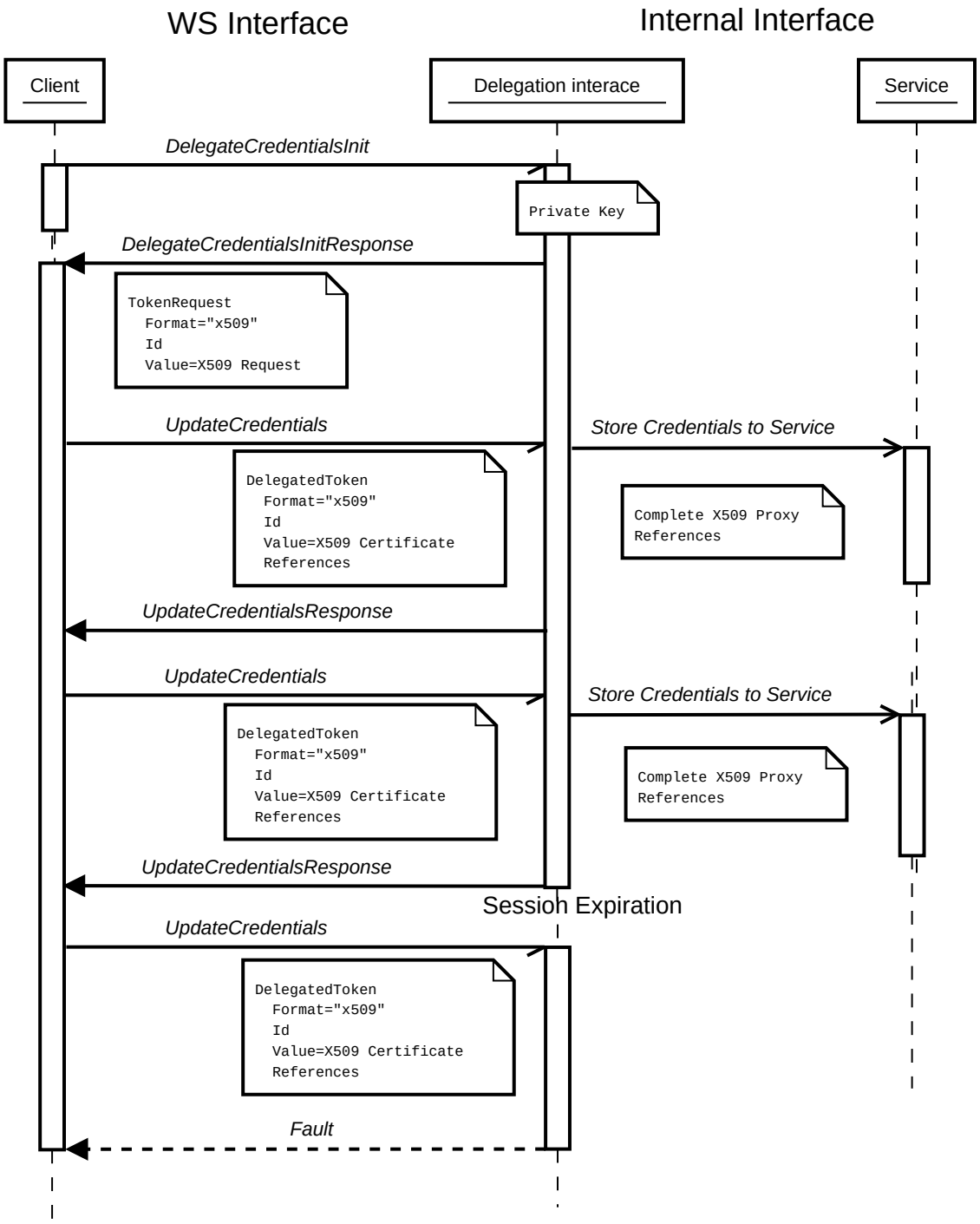

Figure 4: The flow diagram of delegation procedure with multiple second step and session expiration

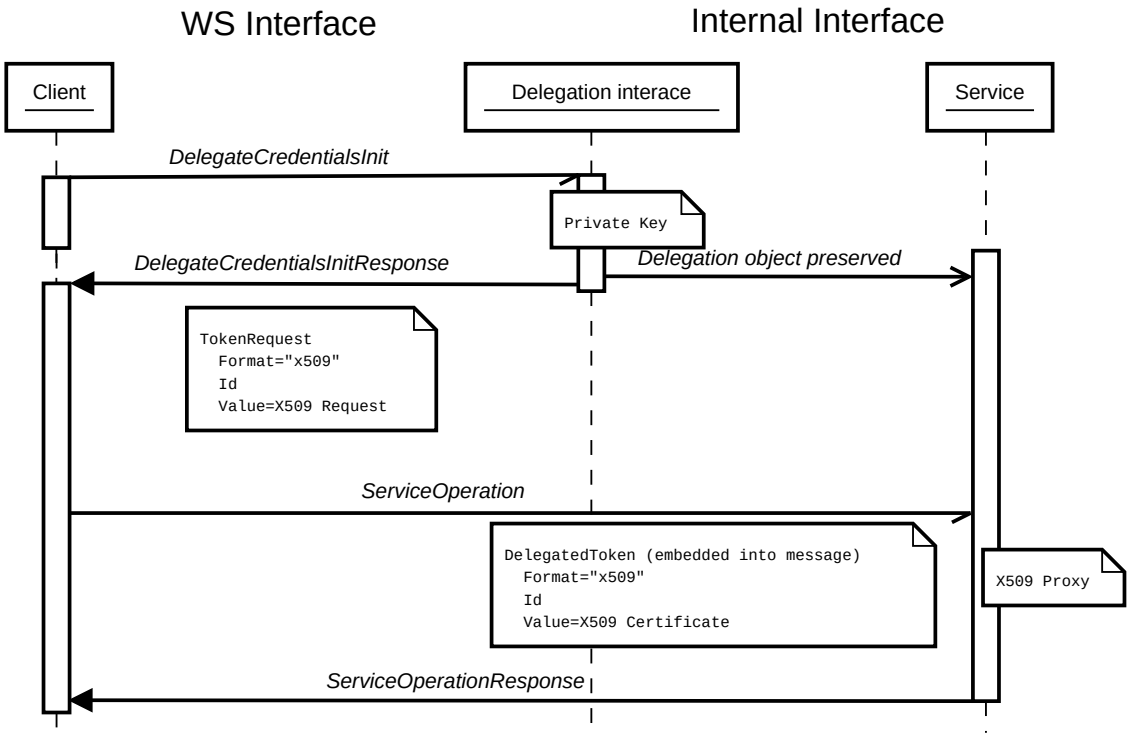

Figure 5: The flow diagram of delegation procedure with certificate transferred as payload of service specific message

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.

Optionally step 2 can be skipped and the DelegatedToken element provided to Service as additional payload of other service specific message.

The Id element obtained in the step 1 can be reused multiple times with different content of the Value element.

WSDL of portType implementing delegation functionality can be found at

`http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/libs/delegation/delegation.wsdl` .

6.5 Delegated Credentials (Proxy) Generation Utility

Command line utility arcproxy can be used to generate X.509 Proxy Certificate with (or without) Policy embedded. The arcproxy my be used in following way:

approxy -P proxy.pem -C cert.pem -K key.pem -c constraint

Here options -P, -C and -K specify path to files containing generated Proxy, user's credentials and user's private key respectively. By using argument "-c", some constraints can be specified for proxy certificate. Each constraint string is a key and value pair with key representing type of contraint. There may be multiple -c options specified. Currently suported contraint types are:

- validityStart, validityEnd and validityPeriod specify when Proxy becomes valid, when its validity finishes or for how long the Proxy is valid respectively. For example

  -c validityStart=2008-05-29T10:20:30Z

  -c validityEnd=2008-06-29T10:20:30Z

- proxyPolicy and proxyPolicyFile specify the Policy document to be embedded into the Proxy either directly or by pointing to the file which contains that document. Like

  -c proxyPolicyFile=delegation_policy.xml

The Policy maybe any of any type supported by ARC middleware (or third-party plugins) installed on the services where that policy is processed. Currenlty supported Policies include ARC Policy (described in section 8.1) and GACL Policy [7].

Simple example below renders delegated credentials usable only for contacting service attached to HTTP communication channel under path /arex (line 5) and allows HTTP operation POST (line 8) on it.

```
1.<?xml version="1.0" encoding="UTF-8"?>
2.<Policy xmlns="http://www.nordugrid.org/schemas/policy-arc" PolicyId="sm-example:policy1"
      CombiningAlg="Deny-Overrides">
3.    <Rule RuleId="rule1" Effect="Permit">
4.    <Resources>
5.      <Resource Type="string"
                  AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/path">
                 /arex
              </Resource>
6.    </Resources>
7.    <Actions>
8.      <Action Type="string"
                  AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/method">
                  POST
              </Action>
9.    </Actions>
10.   </Rule>
11.</Policy>
```

Another example of the delegation policy is presented below. This policy restricts usage of delegated credentials to SOAP operation CreateActivity (line 5) of Basic Execution Service (BES) [5] namespace (line 9). Such policy could be embedded into credentials delegated to high level Brokering service performing Grid job submission to low level BES on behalf of user.

```
1.<?xml version="1.0" encoding="UTF-8"?>
2.<Policy xmlns="http://www.nordugrid.org/schemas/policy-arc" PolicyId="sm-example:policy1"
    CombiningAlg="Deny-Overrides">
3.    <Rule RuleId="rule1" Effect="Permit">
4.        <Actions>
5.      <Action Type="string"
                AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/soap/operation">
                CreateActivity
              </Action>
6.    </Actions>
7.    <Conditions>
8.      <Condition>
9.        <Attribute Type="string"
                     AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/soap/namespace"
                     http://schemas.ggf.org/bes/2006/08/bes-factory
                  </Attribute>
10.     </Condition>
11.   </Conditions>
12.   </Rule>
13.</Policy>
```

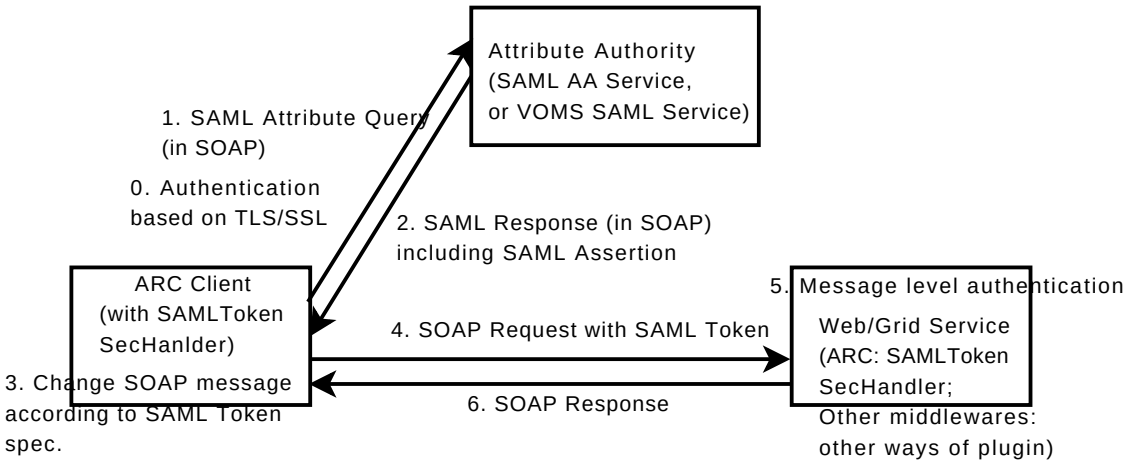

Figure 6: Interaction among Client, Grid/Web Service and Attribute Authority

6.5.1 Delegated Credentials with VOMS Attributes

Currently the proxy creation utility arcproxy can also be used for creating VOMS Proxy Certificate, as the way to replace the "voms-proxy-init" utility.

7 Web Service Security Support

7.1 UsernameToken SecHandler

The UsernameToken SecHandler is meant for processing - generating and extracting - WS-Security [8] UsernameToken in the SOAP header. Hence it must be attached to the MCC which processes SOAP payloads - like SOAP MCC or Service accepting SOAP messages. For description of configuration see section 8.13.

For the incoming message this SecHandler authorizes SOAP message according to specified configuration.

For the outgoing message this SecHandler creates and adds proper token into SOAP header according to configuration.

7.2 X509Token SecHandler

The X.509 Token SecHandler is meant for processing - generating and extracting - WS-Security [8] X.509 Token from SOAP header. Hence it must be attached to the MCC which processes SOAP payloads like SOAP MCC or Service accepting SOAP messages. For description of configuration see section 8.14.

For the incoming message this SecHandler decrypts and checks signature of SOAP message using attached public key and verifies that key against specified CA certificate.

For the outgoing message this SecHandler creates X.509 Token in SOAP header. SOAP message body is encrypted and signed.

7.3 SAMLToken SecHandler

The SAMLToken SecHandler is meant for processing - generating and extracting - WS-Security [8] SAMLToken from SOAP header. Hence it must be attached to the MCC which processes SOAP payloads like SOAP MCC or Service accepting SOAP messages. For description of configuration see section 8.15.

Figure 6 shows the interaction among Client, Grid/Web Service and Attribute Authority when SAMLToken security handler is deployed. For the Grid/Web service, if the service is hosted in ARC middleware, then the SAMLToken security handler should be deployed; if the service is hosted in other middlewares, then there

should be other ways for supporting SAML Token authentication, e.g., Rampart (WS-Security module for Axis2).

For the incoming message this SecHandler decrypts and checks signature of SOAP message using attached public key and verifies that key against specified CA certificate.

For the outgoing message this SecHandler creates SAMLToken in SOAP header. SOAP message body is encrypted and signed.

8 Schemas, descriptions and examples

8.1 Authorization Policy

XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/arcpdp/Policy.xsd

8.2 Authorization Request

XML schema with comments is available at

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

8.3 Authorization Response

XML schema with comments is available at

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

8.4 Interface of policy decision service (Charon service)

WSDL with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/charon/charon.wsdl

The following is the configuration of the Charon service, which is configured to use xacml policy engine. The policy engine is configurable by changing the "name" attribute of three elements: <*Evaluator*>, <*Policy*>, <*Request*> into "arc.evaluator", "arc.policy", "arc.request". Also the policy should also be changed to the one with ARC specific format.

```
<Service name="charon" id="charon_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-->
    <charon:PDPConfig>
        <charon:PolicyStore>
            <charon:Location Type="file">charon_policy_xacml.xml.example</charon:Location>
            <!-- other policy location-->
        </charon:PolicyStore>
        <charon:Evaluator name="xacml.evaluator" />
        <charon:Policy name="xacml.policy" />
        <charon:Request name="xacml.request" />
    </charon:PDPConfig>
</Service>
```

8.5 TLS MCC configuration

For full description of TLS MCC configuration please read "The Hosting Environment of the Advanced Resource Connector middleware". Here only part related to VOMS attributes extraction is provided for convenience. Configuration schema can be found at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/mcc/tls/tls.xsd

While processing VOMS extension of X.509 certificate only attributes which can be verified are extracted and collected in the list of Security Attributes. To ensure proper authentication trusted VOMS services must to be configured. The VOMS services are identified by certificates which they use to sign AC with VOMS related information. And also by whole certificates chain used to sign VOMS service certificate.

The TLS MCC configuration makes it possible to specify DNs of all certificates in such chains. Each chain is stored in separate *<VOMSCertTrustDNChain>* element. Each such element is composed either of multiple *<VOMSCertTrustDN>* elements or single *<VOMSCertTrustRegex>*

Each *<VOMSCertTrustDN>* element defined DN of one certificate in a chain starting from certificate of VOMS service and down to DN of last CA in the chain.

The *<VOMSCertTrustRegex>* element defines regular expression which is applied to every certificate in the chain.

Along with *<VOMSCertTrustDNChain>* it is also possible to specify *<VOMSCertTrustDNChainsLocation>*.

The *<VOMSCertTrustDNChainsLocation>* specifies path to file containing XML document with single *<VOMSCertTrustDNChain>* element.

Below is an example presenting all possible options.

```
<tls:VOMSCertTrustDNChain>
  <tls:VOMSCertTrustDN>/O=Grid/O=NorduGrid/CN=host/arthur.hep.lu.se</tls:VOMSCertTrustDN>
  <tls:VOMSCertTrustDN>/O=Grid/O=NorduGrid/CN=NorduGrid CA</tls:VOMSCertTrustDN>
</tls:VOMSCertTrustDNChain>
<tls:VOMSCertTrustDNChain>
  <tls:VOMSCertTrustDN>/DC=ch/DC=cern/OU=computers/CN=voms.cern.ch</tls:VOMSCertTrustDN>
  <tls:VOMSCertTrustDN>/DC=ch/DC=cern/CN=CERN CA</tls:VOMSCertTrustDN>
</tls:VOMSCertTrustDNChain>
<tls:VOMSCertTrustDNChain>
  <tls:VOMSCertTrustRegex>^/O=Grid/O=NorduGrid</tls:VOMSCertTrustRegex>
</tls:VOMSCertTrustDNChain>
<tls:VOMSCertTrustDNChainsLocation>./voms_trust.xml</tls:VOMSCertTrustDNChainsLocation>
```

8.6 Configuration of PDP service

XML schema with comments available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/charon/charon.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 8.9 for the explanation of ARC Policy.

8.7 Authorization SecHandler configuration

XML schema with comments available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/arcauthzsh/SimpleListAuthZ.xsd

Default behavior of Authorization SecHandler is to execute all PDPs corresponding to elements <*PDP*> in configuration sequentially till either one of them fails or all produced positive results. This behavior may be modified by attribute "action" of embedded <*PDP/*> elements. Following options are supported:

- breakOnAllow - if PDP returned positive result stop PDPs processing and return positive result. Otherwise continue to next PDP or return negative result if no more PDPs to process. That is a default behavior.
- BreakOnDeny - if PDP returned negative result stop PDPs processing and return negative result. Otherwise continue to next PDP or return negative result if no more PDPs to process.
- BreakAlways stop processing PDPs and return result which this PDP returned.
- BreakNever continue to next PDP. If there are no more PDPs to process then return result which this PDP returned.

Below is an example of the Authorization SecHandler with 4 PDPs in the list:

simplelist.pdp for comparing client's credentilals to list of DNs

arc.pdp for comparing collected information to specified policies

pdpservice.invoker for contacting external PDP service.xml

delegation.pdp for evaluating restrictions embedded into X.509 proxy certificates

```
<SecHandler name="arc.authz" id="authz" event="incoming">
  <PDP name="simplelist.pdp" location="simplelistfile"/>
  <PDP name="arc.pdp">
    <PolicyStore>
      <Location type="file">Policy_Example.xml</Location>
    </PolicyStore>
  </PDP>
  <PDP name="pdpservice.invoker">
    <ServiceEndpoint>https://127.0.0.1:60001/pdp.service</ServiceEndpoint>
    <KeyPath>./testkey-nopass.pem</KeyPath>
    <CertificatePath>./testcert.pem</CertificatePath>
    <CACertificatePath>./cacert.pem</CACertificatePath>
  </PDP>
  <PDP name="delegation.pdp"/>
</SecHandler>
```

8.8 SimpleList PDP configuration and Policy Example

XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/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 X.509 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

8.9 Arc PDP configuration and Policy Example

XML schema with comments available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/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 8.1 for 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 X.509 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.

```
<?xml version="1.0" encoding="UTF-8"?>
<Policy xmlns="http://www.nordugrid.org/schemas/policy-arc" PolicyId="sm-example:arcpdppolicy"
    CombiningAlg="Deny-Overrides">
  <Rule Effect="Permit">
    <Description>
      Example policy for echo service
    </Description>
     <Subjects>
        <Subject>
           <Attribute
            AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/tls/ca"
            Type="string">
            /C=NO/ST=Oslo/O=UiO/CN=CA
           </Attribute>
           <Attribute
            AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/tls/identity"
            Type="string">
            /C=NO/ST=Oslo/O=UiO/CN=test
           </Attribute>
        </Subject>
     </Subjects>
     <Resources>
        <Resource
         AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/path"
         Type="string">
         /Echo
        </Resource>
     </Resources>
     <Actions>
        <Action
         AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/method"
         Type="string">
         POST
        </Action>
        <Action
         AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/http/method"
         Type="string">
         GET
        </Action>
        <Action
         AttributeId="http://www.nordugrid.org/schemas/policy-arc/types/soap/operation"
         Type="string">
         echo
        </Action>
     </Actions>
     <Conditions/>
  </Rule>
</Policy>
```

8.10 PDP Service Invoker configuration

Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/pdpserviceinvoker/PDPServiceInvoker.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>
        <RequestFormat>XACML</RequestFormat>
        <TransferProtocol>SAML</TransferProtocol>
      </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. In addition, the <RequestFormat> element is for specifying the format of the request, and the <TransferProtocol> element is for specifying the protocol of tranfering the request.

Table 6 shows the support of request and protocol in remote policy decision making. Note that if "SAML 2.0 profile of XACML v2.0" is configured in "pdpservice invoker", besides the authorizaton service (called charon service) implemented in ARC, it can interact with external authorization services, such as gLite authorization service (Yet the interoperation test has not been done).

Table 6: Support of request and protocol in remote policy decision making

	ARC Request/Response	**XACML Request/Response**
ARC protocol	supported	supported
SAML protocol	not supported	supported (SAML 2.0 profile of XACML v2.0)

ARC protocol:

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/charon/charon.wsdl

SAML protocol:

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/services/charon/charon.wsdl

http://www.oasis-open.org/committees/download.php/11475/access_control-xacml-2.0-saml-protocol-schema-os.xsd

ARC request/response:

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

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

Note that the format of "Response" is supposed to correspond with the format of "Request".

XACML request/response:

http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-context-schema-os.xsd

http://www.oasis-open.org/committees/download.php/11474/access_control-xacml-2.0-saml-assertion-schema-os.xsd

8.11 Delegation PDP configuration

Configuration XML schema with comments available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/delegationpdp/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. It is enough to switch it on by adding *<PDP name="delegation.pdp"/>* under *<SecHandler>* which supports processing of PDPs (currently arc.authz). The precondition for using Delegation PDP is that there must be Delegation SecHandler instantiated earlier in the chain.

8.12 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.

8.13 UsernameToken SecHandler Configuration

Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/usernametokensh/UsernameTokenSH.xsd

Below is an example of configuration of UsernameToken SecHandler inside MCCSOAP component of the service side. This example processes UsernameToken related information in SOAP message and returns failure if any problem found. In detail, this SecHandler will check header of the incoming SOAP message for the presence of UserName WS-Security token and compare provided password value to those stored in the local file password.txt.

```
<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>
```

This example will add token with username “user” and password “pass” (using “digest” encoding algorithm) into outgoing SOAP message.

For the client side, the developer should add the configuration information about X.509 security handler into client's configuration, in order to generate X.509 Token into any SOAP message. Below is an example of how to use it:

```
    Arc::XMLNode sechanlder_nd_ut("\
        <SecHandler name='usernametoken.handler' id='usernametoken' event='outgoing'>\
            <Process>generate</Process>\
            <PasswordEncoding>digest</PasswordEncoding>\
            <Username>user</Username>\
            <Password>passwd</Password>\
        </SecHandler>");
  Arc::ClientSOAP *client;
  client = new Arc::ClientSOAP(mcc_cfg,url);
  client->AddSecHandler(sechanlder_nd_ut, "arcshc");
```

8.14 X509Token SecHandler configuration

Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/x509tokensh/X509TokenSH.xsd

Below is an example of configuration of X.509 Token SecHandler inside MCCSOAP component of the service side. This example processes X.509 Token related information in SOAP message and returns failure if any problem found.

```
<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>
```

For the client side, the developer should add the configuration information about X.509 security handler into client's configuration, in order to generate X.509 Token into any SOAP message. Below is an example of how to use it:

```
  Arc::XMLNode sechanlder_nd_xt("\
        <SecHandler name='x509token.handler' id='x509token' event='outgoing'>\
            <Process>generate</Process>\
            <CertificatePath>./testcert.pem</CertificatePath>\
            <KeyPath>./testkey-nopass.pem</KeyPath>\
        </SecHandler>");
  Arc::ClientSOAP *client;
  client = new Arc::ClientSOAP(mcc_cfg,url);
  client->AddSecHandler(sechanlder_nd_xt, "arcshc");
```

8.15 SAMLToken SecHandler Configuration

Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/samltokensh/SAMLTokenSH.xsd

8.16 ARC Legacy SecHandler Configuration

Name of plugin which contains ARC Legacy SecHandler is "arclegacy.handle". Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/legacy/schema/ARCSHCLegacy.xsd

Only configuration element "ConfigFile" defines configuration file to be processed. The format of file is described in [6]. Only "vo" and "group" blocks are processed and corresponding matching groups and VOs are identified. There may be multiple "ConfigFile" elements specified. Matching VOs and groups are recorded in dedicated Security Attribute object.

8.17 ARC Legacy PDP Configuration

Name of plugin which contains ARC Legacy PDP is "arclegacy.pdp". Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/legacy/schema/ARCSHCLegacy.xsd

Configuration elements "VO" and "Group" define VOs or groups which trigger positive result of this PDP. If Security Attribute created by ARC Legacy SecHandler contains any of VO or group listed in configuration of this PDP result is positive.

8.18 ARC Legacy Identity Mapping SecHandler Configuration

Name of plugin which contains ARC Legacy SecHandler is "arclegacy.map". Configuration XML schema with comments is available at

http://svn.nordugrid.org/trac/nordugrid/browser/arc1/trunk/src/hed/shc/legacy/schema/ARCSHCLegacy.xsd

Configuration specifies configuration file and multiple configuration blocks which are processes for identity mapping commands defined in [2] section "General Configuration Parameters". The format of configuration file is described in [6]. Processing of mapping commands stops and first match and obtained local identity name is stored in "SEC:LOCALID" attribute of the Message.

References

[1] Public-Key Infrastructure (X.509) (PKI), Proxy Certificate Profile. URL `http://rfc.net/rfc3820.html`.

[2] D. Cameron A. Konstantinov. *The NorduGrid GridFTP Server: Description And Administrator's Manual*. The NorduGrid Collaboration. URL `http://www.nordugrid.org/documents/gridftpd.pdf`. NORDUGRID-TECH-26.

[3] D. Cameron et al. *The Hosting Environment of the Advanced Resource Connector middleware*. URL `http://www.nordugrid.org/documents/ARCHED_article.pdf`. NORDUGRID-TECH-19.

[4] I. Foster et al. A Security Architecture for Computational Grids. In *CCS '98: Proceedings of the 5th ACM conference on Computer and communications security*, pages 83–92. ACM Press, November 1998. ISBN 1-58113-007-4.

[5] I. Foster et al. OGSA™ Basic Execution Service Version 1.0. GFD-R-P.108, August 2007. URL `http://www.ogf.org/documents/GFD.108.pdf`.

[6] A. Konstantinov. *Configuration and Authorisation of ARC (NorduGrid) Services*. The NorduGrid Collaboration. URL `http://www.nordugrid.org/documents/Config_Auth.pdf`. NORDUGRID-TECH-6.

[7] A. McNab. The GridSite Web/Grid security system: Research Articles. *Softw. Pract. Exper.*, 35(9): 827–834, 2005. ISSN 0038-0644.

[8] OASIS. OASIS Web Services Security specification. February 2006. URL `http://www.oasis-open.org/specs/index.php#wssv1.1`.

[9] OASIS. OASIS eXtensible Access Control Markup Language. February 2005. URL http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.