AAAARCH Research Group
INTERNET DRAFT
Category: Experimental

S.M.C.M. van Oudenaarde
L.H.M. Gommans
C.T.A.M. de Laat
F. Dijkstra
A. Taal
September 2004

Prototype of a Generic AAA Server
draft-irtf-aaaarch-prototype-02.txt

Status of this Memo

Abstract

In this document a prototype of an AAA (Authentication, Authorization, Accounting) server is presented. The prototype is build in accordance with the RFCs 2903, 2904 and 2905.
As the AAA concept is a multi-tier concept we have chosen for JAVA Enterprise Beans (J2EE) to build the prototype. New techniques and protocols supported by the J2EE platform are discussed. Web service standards like SOAP are explored. A general architecture of an AAA server is outlined in Enterprise JavaBeans (EJB) component architecture.

Table of Contents

Status of this Memo 1
Copyright Notice 1
Abstract 2
1. Introduction 3
2. Generic AAA Architecture 3
2.1 Agent, pull and push sequences 4
2.1.1 The agent sequence 4
2.1.2 The 'push' and 'pull' sequence 5
3. The demo setup 5
4. Implementation details 6
4.1 AAA Requests in the Generic AAA Architecture 6
4.2 Interface(s) 8
4.3 Driving Policy 9
4.4 ASM Framework 10
4.5 Session 11
5. Conclusions and suggestions for further research 11
Appendix A 12
Appendix B 13
References 16
Authors' Addresses 16

1. Introduction

The RFCs 2903, 2904 and 2905 describe the concept of AAA, Authentication, Authorization and Accounting. These RFCs were a product of the AAAARCH research group of the IRTF. As members of this research group we present a prototype of an AAA server in accordance with these RFCs. We have chosen for the J2EE platform that offers a multitiered distributed application model. The J2EE environment offers support for Extensible Markup Language (XML)-based data interchange, flexible transaction control, an unified security model, and Web services that are based on SOAP and HTTP. Several key parts of the AAA concept, not specified in the above mentioned RFCs, are introduced. One newly introduced key part is a so called AAA Driving Policy. For each request an AAA server understands there exists a corresponding Driving Policy that is evaluated. The main task of a Driving Policy is to describe which pre-conditions have to be checked before actions, needed to fulfill an incoming AAA request, are delegated to Application Specific Modules. AAA requests are expressed in XML whereas Driving Polices are expressed in a special policy language (see Appendix A). AAA servers will be specialized in delivering certain kind of services. Support for WSDL, an XML format for describing network or web services, and support for UDDI, a meta service for discovering network or web services, are integrated into the AAA concept. These techniques give clients the possibility to discover to which AAA server they should address what kind of request in order to get the service of their desire delivered. Part of the J2EE platform is the Enterprise JavaBeans (EJB) standard. This is a component architecture for deployable server-side components in Java. The generic AAA server is build of EJB components, Session Beans and Entity Beans.

2. Generic AAA Architecture

Before discussing implementation details we will outline an abstract view of the Generic AAA Architecture. The Generic AAA Architecture comprises of four major components, fig. 1. An AAA Request is any kind of message that asks for a service.
When a Generic AAA Server receives an AAA Request it will initiate the evaluation of a policy, a so-called Driving Policy. This Driving Policy instructs the Generic AAA Server what conditions need to be checked before certain Service Equipment is told to do something. Service Equipment can be anything, like switches, routers, bandwidth brokers, network access equipment, remote instrumentation, etc. The ASM, Application Specific Module, forms an API to communicate with the Service Equipment. So an

Application Specific Module is a component of an AAA Server that allows Driving Policies to influence Service Equipment.
There exists a tight relationship between AAA Requests, Driving Policies, and Application Specific Modules. Firstly, AAA Requests and Driving Policies have a one-to-one and onto relationship.

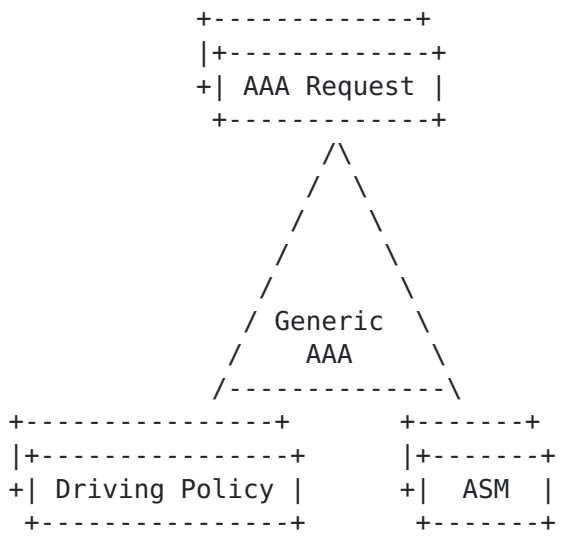

Fig. 1 Components of the Generic AAA Architecture

Based on the request type, the AAA server to retrieve the Driving Policy that needs evaluation. All actions in a Driving Policy refer either to a generic function the Generic AAA is equipped with or to an Application Specific Module.

2.1 Agent, pull and push sequences

In RFC 2904 "AAA Authorization Framework" three authorization sequences are described, the 'agent', the 'pull', and the 'push' sequence. In the following we focus on the æagentÆ model.

2.1.1 The agent sequence

In the agent sequence (fig. 2), the AAA Server functions as an agent between the User and the Service Equipment (SE). The User sends a Request to the AAA Server (1). The Interface unpacks the Request and sends it to a Rule Based Engine (RBE) (2). Before the RBE will retrieve the corresponding Driving Policy and Reply from the Policy Repository (PR) (4), it asks for a new Session to be created (3). Instructed by the Driving Policy the RBE calls one or more ASMs (5) and passes the arguments needed. While an AAA server has exactly one RBE defined, and one Session Manager, it may have multiple ASMs at its disposal. Arguments passed to an

ASM may originate from the incoming request or from values returned by previous calls. These arguments might be needed by the Service Equipment the ASM interfaces to (6). Values returned by an ASM (7) may also be inserted into the Reply to the User. Once the Driving Policy has been decided the Reply is returned to the User (8,9). When there is no need for the Session Manager to keep the information of this Session into persistent storage after

the User received an answer, the Session Manager might write that information into a log file.

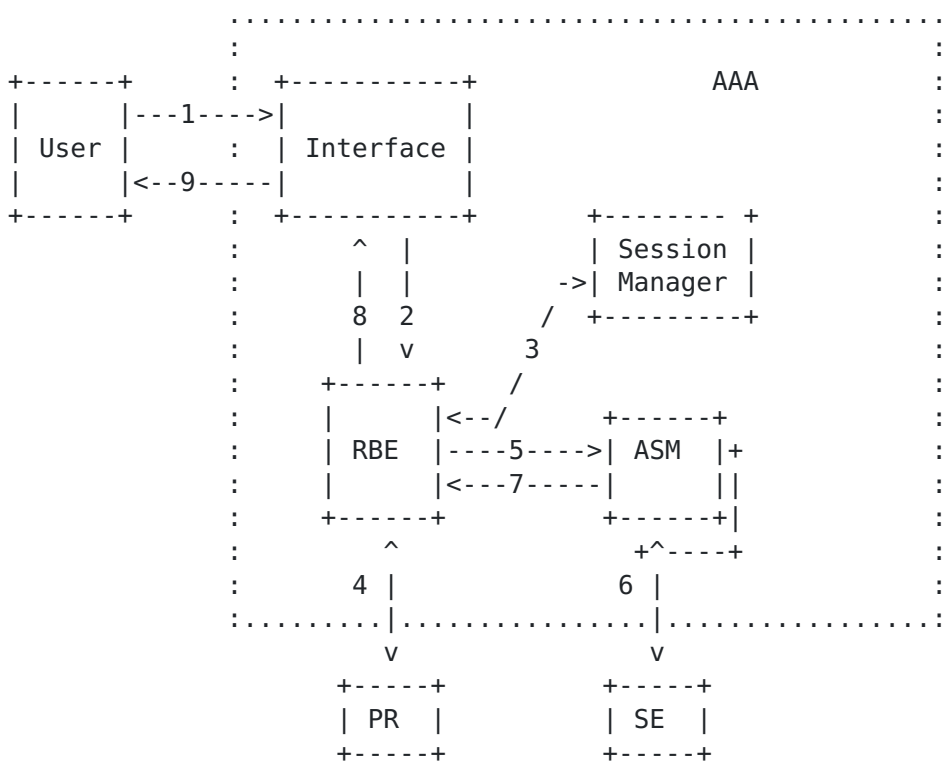

Fig. 2 Schematic view of an agent sequence.

2.1.2 The 'push' and 'pull' sequence

In the 'pull' sequence, as defined in [RFC 2904], the User sends a service request to the Service Equipment, which forwards it to an AAA Server. The AAA Server evaluates the request and returns an appropriate response to the Service Equipment, which sets up the service and tells the User it is ready. Here the Service Equipment sends an AAA Request to the AAA Server. In general the the User and Service Equipment apply a different protocol, and the Service Equipment has to translate the request from the User and the Reply from the AAA Server. In the 'push' sequence it is the User that in general applies two different protocols. There the User gets from an AAA Server a ticket or certificate verifying that it is o.k. for the User to have access to a Service Equipment.

3. The demo setup

The AAA server build of EJB components and discussed below is applied in the following setup (fig. 3). We setup a QoS (Quality of Service) path provision demo between two administrative domains.

In each administrative domain an AAA server (2 and 3) takes care of the admission control of the network elements needed for the QoS path. The network elements, Calient Optical Cross Connects (OXC), were interconnected by two optical links (lambda 1 and 2). One domain, called NetherLight, was situated at the Dutch optical research facility. The other domain, called StarLight, was at the USA Chicago-based counterpart. In fig. 3 the AAA server 1 acts as a broker that tries to find the resources, i.e. the AAA servers 2 and 3 needed to setup the QoS path. The authorization decision is a multi-domain decision. Both 'admission-AAAs' (2 and 3) will make local decisions that are used in an overall authorization decision by the 'broker-AAA' (1). All three AAA servers have in their own Driving Policy. The Driving Policies applied by the 'admission-AAAs' refer for complex tasks, like configuration checks on the state of the OXCs, to their own ASMs.
These ASMs are hiding the complexity of the tasked to be performed for the setup of the QoS path. If a User's request is satisfied, a dedicated optical path is provisioned, which could be used for large traffic between the two domains.

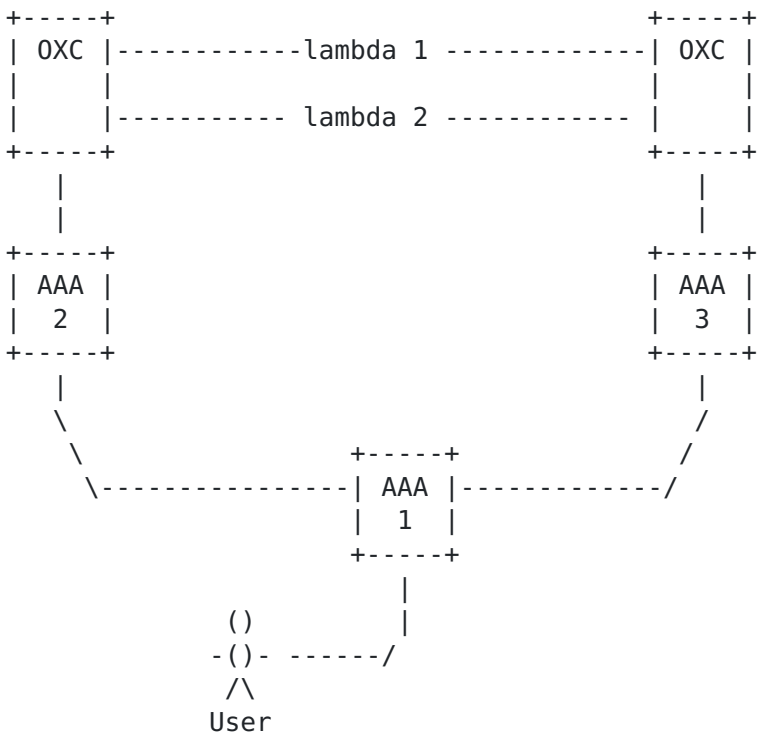

Fig. 3 Demo setup for QoS path provision.

4. Implementation details

4.1 AAA Requests in the Generic AAA Architecture

An AAA Request defines what can be asked of or provided to a Generic AAA Server. For each kind of AAA Request data objects

should be defined with the information that can be incorporated into a specific AAA Request. To accommodate all possible future services by a limited number of predefined AAA Requests is undoable. There is only a need to predefine AAA Requests for the inter communication among AAA Servers, e.g. a request for remote authorization or policy evaluation. Web services standards such as SOAP, WSDL and UDDI will enable the creation of AAA Server specific Requests and data objects contained, e.g. how construct an AAA Request to order a pizza at a pizza AAA Server.
The WSDL (Web Services Description Language) is an XML-based document that defines the inputs and outputs of a Web Service, including the XML Schemas that should be used to create the input and output documents. Using WSDL, AAA Request/Reply pairs might be described for special Web Services. Fig 4 shows a logical view of the Web services architecture. The Service Registry provides a centralized location for storing service descriptions. An UDDI registry is an example of this type of service registry.

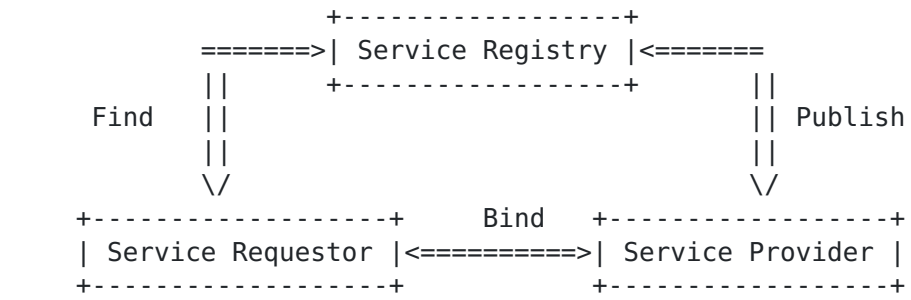

Fig. 4

In our setup the User sends its (XML) Request in the body of a SOAP message over HTTP. A simple Bandwidth on Demand (BoD) Request the 'broker-AAA' accepts, might look like:

```
<AAA:AAARequest
        xmlns:AAA="http://www.aaaarch.org/ns/AAA_BoD"
        xmlns:xsi ="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.aaaarch.org/ns/AAA_BoD
                http://169.254.0.1/LambdaBoDRequest.xsd"
          version="0.1" type="LambdaBoDCross" >
  <Authentication>
    <Signature>17520</Signature>
    <User>Joe</User>
  </Authentication>
  <Authorization>
    <CredentialID>7531</CredentialID>
```

```
            </Authorization>
            <BoDData>
              <Source>
                <Hostname>hp2</Hostname>
                <OXCName>BeautyCees</OXCName>
```

```
            <OXCDomain>NetherLight</OXCDomain>
            <OXCPort>2</OXCPort>
          </Source>
          <Destination>
            <Hostname>scyalla5</Hostname>
            <OXCName>CHI</OXCName>
            <OXCDomain>StarLight</OXCDomain>
            <OXCPort>2</OXCPort>
          </Destination>
          <Bandwidth>1000</Bandwidth>
          <StartTime>now</StartTime>
          <Duration>20</Duration>
        </BoDData>
      </AAA:AAARequest>
```

The 'type' attribute of the first XML-tag, 'LambdaBoDCross' indicates the kind of Request and is used by the Rule Base Engine to retrieve the corresponding Driving Policy from the repository. Also, this version of the policy is limited to two domains. A typical policy would incorporate usage of a route discovery ASM.

4.2. Interface(s)

We use a servlet, a web-tier component, as a facade for making the AAA functionality available to the outside world. There are a couple of advantages for this approach. As the AAA functionality is based on EJB-technology, both the Client tier and the EJB tier implementation are independent. Clients may access EJB components through a RMI-IIOP connection, but that excludes clients behind a firewall. This fact also favors the choice for a servlet as a single point of access, as firewalls are transparent for HTTP. Additionally an EJB component has to expose its remote interface in order to communicate with the outside world. This means that every single call of the client on an EJB component initiates a remote (RPC) call over the network. For each invocation the EJB Server checks security, transactionality, etc. This might lay a heavy load on the EJB Server in case several methods on several EJB components need to be called by the client in order to get a service done. All these problems are circumvent by a servlet as a single point of entry using the local interface of a session bean (EJB2.0). The session bean the servlet interfaces to is the RBE that will contact those EJB components needed for the service requested. Which EJB components are needed are described by a Driving Policy.
Only some simple actions are performed by the servlet. It extracts the Request from the body of the SOAP message and checks

the schema attributes. It are these attributes the parser of the RBE will apply and they should be correct.

4.3 Driving Policy

In the policy language designed and applied, a Driving Policy is of the form 'if(Condition) then (ActionList) else (ActionList)'. For the grammar see Appendix A. An 'if-then-else' structure has a Boolean truth-value, it is 'true' if its Condition is 'true', and it is 'false' if its Condition is 'false'.

To facilitate the discussion we present a simple Driving Policy in chunks (this is not supported by the grammar). A simple Driving Policy for an AAA server that accepts the above Service request looks like:

```
if( ASM::Authenticator.CheckSignature(
                          Request::Authentication.Signature,
                          Request::Authentication.User )
)
then( ACTION_LIST_1 )
else( Reply::Error.Message = "Authorization failed" )
```

The authentication is delegated to an ASM called Authenticator. Two arguments from the incoming Request are passed to a member function 'CheckSignature' of the authenticator. When the authentication succeeds the action list ' ACTION_LIST_1' in the then-part is executed, otherwise an error message is returned. Action list 'ACTION_LIST_1' consists of two actions, a call to an ASM and an if-then-else structure:

```
credential = ASM::Authorizor.CheckCredentials(
                       Request::Authorization.credentialID )
;
if(  ASM::RM.CheckMultiDomain(
                Request::BoDData.Source.OXCDomain,
                Request::BoDData.Destination.OXCDomain )
)
then( ACTION_LIST_2_1 )
else( ACTION_LIST_2_2 )
```

The return state of the 'CheckCredentials' call is assigned to a policy variable 'credential' for later use. Next an resource manager (RM) is asked whether the connection requested is a muti-domain connection or just a single domain set-up. In case a multi-domain set-up is required action list 'ACTION_LIST_2_1'

is executed. We confined the discussion to the multi-domain set-up. Action list ACTION_LIST_2_1 has the form:

```
   lambda = ASM::RM.Connection(
                   Request::BoDData.Source.OXCDomain,
                   Request::BoDData.Destination.OXCDomain )
   ;
```

```
    if( (lambda <= 0) )
    then( Reply::Error.Message = "No connection available between
                                  domains" )
    else( ACTION_LIST_3 )
```

The connection between the two domains is a virtual lambda and further actions are needed to resolve this virtual connection [FGCS]. In action list 'ACTION_LIST_3' the requested connection is provisioned:

```
   r1 = ASM::AAABean.Cross( credential,
                        Request::BoDData.Source.OXCDomain,
                        lambda,
                        Request::BoDData.Source.OXCPort,
                        Request::BoDData.Bandwidth,
                        Request::BoDData.StartTime,
                        Request::BoDData.Duration )
   ;
   if( (r1 <= 0) )
   then( Reply::Error.Message = "failed to make cross-connect to
                                 port in SRC domain" )
   else( ACTION_LIST_4 )
```

Here the call 'Cross' results in an AAA Request to the AAA Server of the source domain, in fig. 3 this is AAA Server 2.
Action list 'ACTION_LIST_4' is similar to 'ACTION_LIST_3':

```
   r2 = ASM::AAABean.Cross( credential,
              Request::BoDData.Destination.OXCDomain,
              lambda,
              Request::BoDData.Destination.OXCPort,
              Request::BoDData.Bandwidth,
              Request::BoDData.StartTime,
              Request::BoDData.Duration )
   ;
   etc.
```

We short cut the discussion with the remark that in case the call to the destination AAA Server fails (AAA Server 3 in fig. 3) an additional call to the source AAA Server has to be made to cancel the provisioning.
For details about the implementation of Driving Policies as Java objects see Appendix B.

4.4 ASM Framework

Application Specific Modules extend the J2EE environment to the outside world. For example in our Bandwidth on Demand (BoD) service an ASM monitors and controls the state of switches. This is realized with the Java Connector Architecture (JCA). The JCA is the bridge between J2EE and the Enterprise Information Systems.

JCA standard provides a mechanism to store and retrieve enterprise data in J2EE. To make a decision in the RBE about a BoD service the state of a switch needs to be checked. This is realized by a connection oriented control adapter (JCA), which translates the current state of the switch to an EJB entity bean in the J2EE container. All information of the switch(es) are translated into the entity bean, including the methods to control the switch.

4.5 Session

If an incoming Request is forwarded to the RBE, a Session Manager (entity bean) is contacted by the RBE. The Session Manager starts a session that will keep information about the Use Case of the AAA Request. Each session is characterized by a sessionID, in our case a primary key. The session information itself is persistent, as the information must be recoverable after failure of the AAA server. There are Use Cases for which the session information should be kept in persistent storage for a certain time span after the User received a Reply. In order to inform the RBE whether or not it should ask for a Session persistent after the Reply is returned, a special attribute in the Request tag might be defined. In case the Session information should be kept for a longer time, the Driving Policy has at least one Action to add the sessionID to the Reply returned. This sessionID is needed to retrieve information about or add information to the Session. For auditing or accounting purposes full information of all equipment involved should be available at the AAA server that received the Request. This description of an AAA Session is far from complete and further research is needed to complete it (see next section).

5. Conclusions and suggestions for further research

The Generic AAA architecture is best suited for policy based decision taking at the business level involving high level service abstractions and user definitions. The business level involves decision taking based on simple policy rules and simple messages exchanged. Due to the diversity in service operation methods, a flexible way is needed to interface with various service entities in different domains. The role assigned to Driving Policies in the AAA concept and the relatively simple policy language suffice to make the decision taking at the business level. Fine-grained policy decisions should not be made at the level of Driving Policies, but should be made by ASMs applying proprietary admission control software. The policy language forces the

developer to off-load all semantic handling of attributes not important at the business level to ASMs. This entails that multi-domain decision taking is purely made on business logic. Further research is needed to prove the correctness and usefulness of this approach. Another issue to be settled by further research is performance. To set-up a QoS path involves decision taking at

different levels and in different domains and might be a time consuming process. One way to cope with long set-up times is parallelization. The policy language should allow the use of concurrency operators. A consequence of concurrent actions is the need to cancel actions in process in case the failure of one action makes the outcome of other ongoing, unfinished actions irrelevant. Other questions we are interested in is how a dynamic trust-relation with other parties can be established and how to combine authorizations from different administrative domains applying pre-established trust relationships. Authorizations based on monetary units seem a promising approach.
Inter AAA communication might be based on a TLS based transport mechanism or on an RFC3281 attribute certificate in a the Request.
Integration of Security Assertion Markup Language (SAML) in the above AAA concept is worthwhile to look at.
Finally, some standardization of messages is required, especially for those messages exchanged among AAA Servers, e.g. to retrieve a policy from another AAA Server or error message including error codes.

Appendix A

A grammar for Driving Policies. The notation of the grammar below is in EBNF (Extended Backus Naur Formalism), terminal symbols are placed between double quotes:

```
DrivingPolicy ::= "if"   "(" Condition ")"
                  "then" "(" ActionList ")"
                  "else" "(" ActionList ")"


Condition ::= Bool
            | Var
            | {Var "="}? Procedure
            | ComputedBoolean
            | UnaryBooleanOperator Condition
            | "(" Condition BinaryBooleanOperator
                  Condition ")"


UnaryBooleanOperator ::= "!"
BinaryBooleanOperator ::= "&&" | "||"


Procedure ::= ProcedureName "(" ARGList ")"
```

```
ARGList ::= {ARG {"," ARG}*}?


ARG ::= Bool | String  | ComputedBoolean
      | NonBooleanExpr


ComputedBoolean ::= "(" NonBooleanExp  ComparisonOperator
                        NonBooleanExpr  ")"
```

```
ComparisonOperator ::= "=="
                     | ">" | ">=" | "<" | "<=" | "!="


NonBooleanExpr ::= Int | Float | Var | Procedure
                 | UnaryArithmeticOperator NonBooleanExpr
                 | "(" NonBooleanExpr  BinaryArithmeticOperator
                       NonBooleanExpr ")"


UnaryArithmeticOperator  ::= "-"
BinaryArithmeticOperator ::= "+" | "-"  | "/" | "*"
                           | "%" | "&" | "|"


ActionList ::= {Action {";" Action}*}?


Action ::= Var "=" Bool | Var "=" String
         | Var "=" ComputedBoolean
         | Var "=" NonBooleanExpr
         | Procedure
         | DrivingPolicy


Var ::= Source "::" Source {"." Source}*
Source ::= Identifier
ProcedureName ::= Identifier "::" Identifier "." Identifier


Identifier ::= "[a-zA-Z_].[a-zA-Z0-9_]*"
String ::= "\"[^"\n]*\""
Int ::= "-?[0-9]+"
Float ::= "-?[0-9]+\.[0-9]*(E-?[0-9]+)?"
Bool ::= "(true|false)"
```

Evaluation of a Boolean expression is performed according to the C-language convention. This makes an if-statement deterministic, and as such there is no need to allow nesting of Driving Policies in a Condition. Take for instance the following nested Driving Policy:

```
  if( A || Pol )
  then( a0 ) else ( a1 )
```

with Pol: if(B) then(b0) else (b1).

Adopting the C convention this Driving Policy is equivalent to

```
 if( A ) then( a0 )
 else ( if( B ) then( b0 ; a0 ) else( b1 ; a1 ) )
```

Appendix B

A parser for the grammar in Appendix A is discussed by applying Java Compiler Compiler (JavaCC). The semantic actions of the parser yield a Driving Policy as a serialized object.

Implementing Driving Policies as a serialized objects, as discussed below, simplifies the RBE to a 'RPN calculator-like' device. This is accomplished by defining some simple classes that cover the non-terminals of the grammar:

```
public class DrivingPolicy {
  public Stack conditionStack;
  public Stack thenStack;
  public Stack elseStack;
}


public class Procedure extends Stack {
  public String name = "";
}


public class Expression extends Stack {
  public static final int COMP_BOOLEAN =0;
  public static final int NON_BOOLEAN  =1;
  public int type = -1;
}


public class Literal {
  public static final int BOOL   =0;
  public static final int INT    =1;
  public static final int FLOAT  =2;
  public static final int STRING =3;
  public static final int VAR    =4;
  public int type = -1;
  public String stringValue = "";
}


public class Operator {
  public static final int NOT     =0; // '!'
  public static final int AND     =1;
  (...)
  public static final int GET     =6; // '>='
  (...)
  public static final int PLUS    =9;  // '+'
  public static final int MINUS   =10;
  public static final int MUL     =11; // '*'
  (...)
  public int type = -1;
}
```

```
public class Assignment {
  public Object lval;
  public Object rval;
}
```

The following example will illustrate how the parser produces a Driving Policy object. Take for example the following Driving Policy that checks whether some arithmetic manipulation of a

number 'Data' from a request is larger than the number 169:

```
if(
    ( ASM::Calculator.power(((2*3)+Request::Data),2) >= 169 )
)
then( Reply::Answer  = "Yes" )
else( Reply::Answer  = "No" )
```

The serialized JAVA object produced has three stacks that are populated as follows:

```
conditionStack:
| - Procedure(ASM::Calculator.power):
|   | - Expression(NON_BOOLEAN):
|   |   | - Literal(INT): "2"
|   |   | - Literal(INT): "3"
|   |   | - Operator(MUL)
|   |   | - Literal(VAR): "Request::Data"
|   |   | - Operator(PLUS)
|   |
|   | - Expression(NON_BOOLEAN):
|       | - Literal(INT):  "2"
|
| - Literal(INT): "169"
| - Operator(GET)
```

```
thenStack:
| - Assignment:
     | - lval:
     |     | -Literal(VAR): Reply::Answer
     |
     | - rval:
           | - Literal(STRING): "Yes"
```

```
elseStack:
| -  Assignment:
     | - lval:
     |     | -Literal(VAR): Reply::Answer
     |
     | - rval:
           | - Literal(STRING): "No"
```

The 'conditionStack' contains three objects, a Procedure object, a Literal object, and an Operator object. All objects are placed on the 'conditionStack' adhering to Reverse Polish Notation (RPN). This also holds for the stack of an Expression object. This simplifies the RBE to a 'RPN calculator' like device in determining the truth- value of the Condition. It just pops, resolves and pushes objects from and onto the 'conditionStack' until the stack contains a single Literal object of the type BOOL.

Depending on the 'stringValue', whether it is 'true' or 'false', the RBE continues with the 'thenStack' or the 'elseStack' of the DrivingPolicy object.

References

[FGCS] Leon Gommans, Cees de Laat, Bas van Oudenaarde, Arie Taal "Authorization of a QoS path based on generic AAA" in Future Generation Computer Systems, pp. 1009-1016.

Authors' Addresses

Bas van Oudenaarde
Faculty of Science, Informatics Institute,
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Phone: +31 20 5257586
Fax: +31 20 5257490
Email: oudenaar@science.uva.nl

Leon Gommans
Faculty of Science, Informatics Institute,
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Phone: +31 20 5257435
Fax: +31 20 5257490
Email: lgommans@science.uva.nl

Cees de Laat
Faculty of Science, Informatics Institute,
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Phone: +31 20 5257590
Fax: +31 20 5257490
Email: delaat@science.uva.nl

Freek Dijkstra
Faculty of Science, Informatics Institute,
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Phone: +31 20 5257531
Fax: +31 20 5257490
Email: fdijkstr@science.uva.nl

Arie Taal
Faculty of Science, Informatics Institute,
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Phone: +31 20 5257586
Fax: +31 20 5257490
Email: taal@science.uva.nl

Full Copyright Statement

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.