A Policy Based Framework for Access Control

Ricardo Nabhen, Edgard Jamhour, Carlos Maziero

PPGIA – PUC PR – CURITIBA – PARANÁ - BRAZIL
{rcnabhen, jamhour, maziero}@ppgia.pucpr.br

Abstract. This paper presents a policy-based framework for managing access control in distributed heterogeneous systems. This framework is based on the PDP/PEP approach. The PDP (Policy Decision Point) is a network policy server responsible for supplying policy information for network devices and applications. The PEP (Policy Enforcement Point) is the policy client (usually, a component of the network device/application) responsible for enforcing the policy. The communication between the PDP and the PEP is implemented by the COPS protocol, defined by the IETF. The COPS (Common Open Policy Service) protocol defines two modes of operation: outsourcing and provisioning. The choice between outsourcing and provisioning is supposed to have an important influence on the policy decision time. This paper evaluates the outsourcing model for access control policies based on the RBAC (Role-Based Access Control) model. The paper describes a complete implementation of the PDP/PEP framework, and presents the average response time of PDP under different load conditions.

1. Introduction

In policy-based networking (PBN), a policy is a formal set of statements that define how the network's resources are allocated among its clients. Policies may be used to achieve better scaling in network management by describing common attributes of classes of objects, such as network devices, software services and users, instead of individually defining attributes for these elements. In order to implement PBN it is important to define a vendor independent method for representing and storing network policies. A formal method for representing users, services, groups and network elements is also required. An important work in this field, called CIM (Common Information Model), was proposed by the DMTF (Distributed Management Task Force) [4]. The CIM model addresses the problem of representing network resources. PCIM (Policy Core Information Model) is an information model proposed by IETF that extends CIM classes in order to support policy definitions for managing these resources [5]. PCIM is a generic policy model. Application-specific areas must be addressed by extending the policy classes and associations proposed by PCIM. For example, QPIM (QoS Policy Information Model) is a PCIM extension for describing quality of service polices [11]. In this context, this paper describes a PCIM extension for access control, called RBPIM (Role Based Policy Information Model), which permits to represent network access control policies based on roles, as well as static and dynamic constraints, as defined by the proposed NIST RBAC standard [1].

Typically, PCIM is implemented using a PDP/PEP approach [9]. The PDP (Policy Decision Point) is a network policy server responsible for supplying policy information for network devices and applications. The PEP (Policy Enforcement Point) is the policy client (usually, a component of the network device/application) responsible for enforcing the policy. The communication between the PDP and the PEP is implemented by the COPS protocol, defined by the IETF [10]. The COPS (Common Open Policy Service) protocol defines two modes of operation: outsourcing and provisioning. In the outsourcing model, the PDP receives policy requests from the PEP, and determines whether or not to grant these requests. Therefore, in the outsourcing model, the policy rules are evaluated by the PDP. In the provisioning model the PDP prepares and "pushes" configuration information to the PEP. In this approach, a PEP can take its own decisions based on the locally stored policy information.

The motivation for defining RBAC in PCIM terms can be summarized as follows. First, there are several situations where the same set of access control policies should be available for heterogeneous applications in a distributed environment. This feature can be achieved by adopting the PDP/PEP framework. Second, an access control framework requires having access to information about users, services and applications already described in a CIM/PCIM repository. Implementing access control in PCIM terms permits to leverage the existing information in the CIM repository, simplifying the task of keeping a unique source of network information in a distributed environment.

The remaining of this paper is organized as follows: Section 2 presents a short description of the RBAC model used in this paper. Section 3 reviews some related works. Section 4 presents RBPIM. Section 5 presents the RBPIM framework implemented using the outsourcing model. Section 6 presents the performance evaluation results of a prototype of the RBPIM framework under various load conditions. Finally, the conclusion summarizes the main aspects in this project and points to future works.

2. RBAC Model

RBAC models have received a broad support as a generalized approach to access control, and are well recognized for their many advantages in performing large-scale authorization control. Several RBAC models have been proposed, each one exploring features that, supposedly, exhibit true enterprise value. The RBAC model adopted by the RBPIM framework is based on the proposed NIST (National Institute of Standards and Technology) Standard [1]. The RBPIM framework accommodates the most important RBAC features described in [1]. Also, the PEP implementation in the RBPIM framework (called RBPEP – Role Based PEP) is based on API's described in the proposed NIST RBAC functional specification [1]. This section will present a summary of the RBAC features used in the RBPIM framework. The purpose of this summary is to define a standard nomenclature for presenting the RBPIM framework in sections 4 and 5. For a more complete description, please, refer to the proposed NIST standard [1].

The proposed NIST standard presents a RBAC reference model based on four components: Core RBAC, Hierarchical RBAC, Static Separation of Duty Relations and Dynamic Separation of Duty Relations. The idea of organizing the reference model in components is to permit vendors to partially implement RBAC features in their products. The Core RBAC model element includes sets of five basic data elements called users (USER), roles (ROLES), objects (OBS), operations (OPS), and permissions (PRMS). The main idea behind the RBAC model is that permissions are assigned to roles instead of being assigned to users. The User Assignment (UA) is a many-to-many relationship. An important concept in RBAC is that roles must be activated in a session. That means that user must select the roles he wants to activate within a session in order to get the permissions associated to the roles. A session is associated with a single user, and each user is associated with one or more sessions. The Permission Assignment (PA) is also a many-to-many relationship (i.e., a permission can be assigned to one or more roles, and a role can be assigned to one or more permissions). A permission is an approval to perform an operation (e.g., read, write, execute, etc.) on one or more RBAC protected objects (e.g., a file, directory entry, software application, etc.). The Hierarchical RBAC model element introduces role hierarchies (RH). Role hierarchies simplify the process of creating and updating roles with overlapping capabilities. In the proposed RBAC model, role hierarchies define an inheritance relation of permissions among roles; e.g., r_1 "inherits" role r_2 if all privileges of r_2 are also privileges of r_1. The Static Separation of Duty (SSD) model element introduces static constraints to the User Assignment (UA) relationship by excluding the possibility of the user to assume conflicting roles. The proposed RBAC model defines SSD with two arguments: a role set that includes two or more roles, and a cardinality greater than one indicating the maximum combination of roles in the set a user can be assigned, e.g., for constraining a user to assume the roles "r_1" and "r_2", one must define a set $\{r_1, r_2\}$ with cardinality 2 (the user can assume cardinality-1 roles in the set). The Dynamic Separation of Duty (DSD) model element introduces constraints on the roles a user can activate within a session. The strategy for imposing constraints on the activation of roles is similar to the SSD approach, using a set of roles and cardinality greater the one. Note that SSD imposes general constraints on which roles a user can assume, while DSD imposes constraints on which roles a user can simultaneously activate in a session.

The RBPIM framework described in sections 4 and 5 supports all four elements of the proposed NIST standard and proposes a more flexible method for defining UA relationships by combining Boolean conditions as defined by the PCIM standard and its extensions [6].

3. Related Works

Recent works starts exploring the advantages of the PDP/PEP approach for implementing an authorization service that could be shared across a heterogeneous system in a company. An interesting work in this field is the XACML (eXtensible Access Control Markup Language), proposed by the OASIS consortium [12]. XACML is a XML based language that describes both an access control policy language and a request/response language. The policy language is used to express

access control policies. The request/response language is used for supporting the communication between PEP clients and PDP servers. RBPIM framework described in this paper also uses the PDP/PEP approach. However, our approach differs from XACML on several points. First, the RBPIM uses a standard COPS protocol for supporting the PEP/PDP communication, instead of XML. Second, the information model used for describing policies is based on a PCIM extension. Third, RBPIM has been implemented for supporting a specific access control method, the RBAC. That permits to define a complete framework that includes the algorithms in the PDP, especially conceived for evaluating policies that includes hierarchy of roles and both, dynamic and static separation of duties.

Most of the research efforts found in the literature refer to the use of the PCIM model and its extensions for developing policy management tools for QoS support [11]. However, a pioneer work for defining a PCIM extension for supporting RBAC, called CADS-2, has been proposed by BARTZ, L.S. [3]. The CADS-2 is a review of a previous work, called *hyperDRIVE*, also proposed by BARTZ [2]. The *hyperDRIVE* is a LDAP [7] schema for representing RBAC. This schema can be considered as a first step for implement RBAC using the PDP/PEP approach. However, *hyperDRIVE* was elaborated before the PCIM standard, and has been discontinued by the author. As *hyperDRIVE*, CADS-2 defines classes suitable to be implemented in a directory-based repository, such as LDAP. CADS-2 defines RBAC roles in terms of policy objects, and introduces classes to support different comparison operators, e.g., *equal*, *greaterThan*, *lessThan*. These operators permit to represent complex comparison expressions with the attribute values of other object stored in a LDAP repository. These expressions are used to represent the conditions a user must satisfy in order to assume a RBAC role. The RBIM model described in the section 5 uses some ideas presented in the CADS-2 model. Specially, the idea of mapping roles to users using Boolean expressions. Note that this approach offers an additional degree of freedom for creating RBAC policies because the UA (*User Assingment*) relationship can be expressed through Boolean expressions instead of a direct mapping between user and roles. However, a recent IETF publication called PCIMe (PCIM Extensions) proposes a different approach for representing Boolean expressions [6]. The RBPIM framework adopts the PCIMe strategy. Also, many features have been introduced in order to support the other elements of the RBAC model, such as hierarchy of roles, DSD and SSD, not supported in the original CADS-2 model.

4. RBPIM: The Role Based Policy Information Model

Figure 1 shows the PCIM model, and the proposed RBPIM extensions for supporting RBAC policies. In the PCIM approach, a policy is defined as a set of policy rules (*PolicyRule* class). Each policy rule consists of a set of conditions (*PolicyCondition* class) and a set of actions (*PolicyAction* class). If the set of conditions described by the class *PolicyCondition* evaluates to true, then a set of actions described by the class *PolicyAction* must be executed. A policy rule may also be associated with one or more policy time periods (*PolicyTimePeriodCondition* class), indicating the schedule according to which the policy rule is active and inactive. Policy rules may be

aggregated into policy groups (*PolicyGroup* class) and these groups may be nested, to represent a hierarchy of policies.

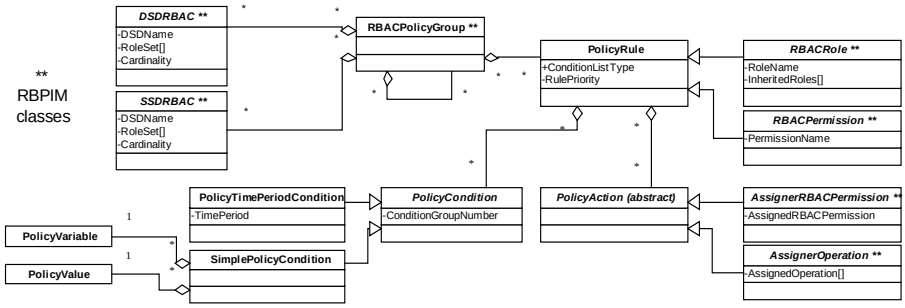

Fig.1. PCIM class hierachy and RBPIM extensions.

In a *PolicyRule*, rule conditions can be grouped by two different ways: DNF (Disjunctive Normal Form) or CNF (Conjunctive Normal Form). The way of grouping policy conditions is defined by the attribute *ConditionListType* in the *PolicyRule* class. Additionally, the attributes *GroupNumber* and *ConditionNegated*, in the association class *PolicyConditionInPolicyRule* helps to create condition expressions. In DNF, conditions within the same group number are ANDed ($\wedge$) and groups are Ored ($\vee$). In CNF, conditions within the same group are ORed ($\vee$) and groups are ANDed ($\wedge$). In order to illustrate this approach, suppose we have a set of five *PolicyConditions* C_i(*GroupNamber*,*ConditionNegated*) as follows: $\mathbf{C}$={C_1(1,*false*), C_2(1,*true*), C_3(1,*false*), C_4(2,*true*), C_5(2,*false*)}. Then, the overall condition for the *PolicyRule* will be defined as:

If *ConditionListType* = DNF then: $\text{evaluate}(\mathbf{C}) = (C_1 \wedge !C_2 \wedge C_3) \vee (C_4 \wedge C_5)$

If *ConditionListType* = CNF then: $\text{evaluate}(\mathbf{C}) = (C_1 \vee !C_2 \vee C_3) \wedge (C_4 \vee C_5)$

The RFC 3460 proposes several modifications in the original PCIM standard. These modifications are called PCIMe (Policy Core Information Model Extensions) [6]. PCIMe solves many practical issues raised after the original PCIM publication. For example, *PolicyCondition* have been extended in order to support a straightforward way for representing conditions by combining variables and values. This extension is called *SimplePolicyCondition*.

The strategy defined by *SimplePolicyCondition* is to build a condition as a Boolean expression evaluated as: does <variable> MATCH <value>? Variables are created as instances of specializations of *PolicyVariable*. The values are defined by instances of specializations of *PolicyValue*. The MATCH element is implicit in the model. PCIMe defines two types of variables: explicit (*PolicyExplicitVariable*) and implicit (*PolicyImplicitVariable*).

Explicit variables are used to build conditions that refer to objects stored in a CIM repository. For example, considers the following condition: *Person.Surname* MATCH “Doe”. *Person.Surname* refers to the *Surname* attribute of the class *Person* in the CIM model. This condition is expressed as *PolicyExplicitVariable.ModelClass*

= “Person” and *PolicyExplicitVariable.Property* = *“Surname”*. Because *Person.Surname* is a string, the *PolicyStringValue* subclass must be used in this condition, i.e., *PolicyStringValue.StringList* = “Doe”. Observe that explicit variables are a very powerful instrument for reusing CIM information in policy based management tools.

Implicit variables are used to represent objects that are not stored in a CIM repository. They are especially useful for defining filtering rules with conditions based on protocol headers, such as source and destination addresses or protocol types. For supporting filtering rules, PCIMe defines several specializations of *PolicyImplicitVariable*, such as *PolicySourceIPv4Variable*, *PolicySourcePortVariable*, etc. These specializations have no properties. For example, the condition “source IPv4 address” MATCH “192.168.0.0/24” would be represented using the class *PolicySourceIPv4Variable* and *PolicyIPv4AddrValue. IPv4AddrList* = “192.168.0.0/24”. PCIMe offers also the possibility of creating conditions that use sets or range of values instead of single values. For example, the condition “source port” MATCH “[1024 to 65535]” would be represented using the class *PolicySourcePortVariable* and *PolicyIntegerValue.IntegerList=”1024..65535”*. Values with wildcards are also permitted. Please, refer to the RFC 3460 for more details about this approach.

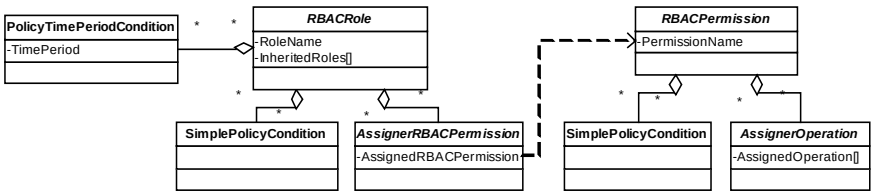

Fig.2. RBPIM class associations

The RBPIM model is a PCIM extension for representing RBAC policies. The RBPIM class hierarchy is shown in the Figure 1. The following classes have been introduced: *RBACPermission* and *RBACRole* (specializations of *PolicyRule*), *AssignerPermission* and *AssignerOperation* (specializations of *PolicyAction*), *DSDRBAC* and *SSDRBAC* (specializations of *Policy*). The *RBACPolicyGroup* class (specialization of *PolicyGroup*) is used to group the information of the constrained RBAC model. As shown in Figure 2, the approach in the RBPIM model consists in using two specializations of *PolicyRule* for building the RBAC model: *RBACRole* (for representing RBAC roles) and *RBACPermission* (for representing RBAC permissions). *RBACRole* can be associated to lists of *SimplePolicyCondition*, *AssignerRBACPermission* and *PolicyTimePeriodCondition* instances. The instances of *SimplePolicyCondition* are used to express the conditions for a user to be assigned to a role (UA relationship). The instances of *AssignerRBACPermission* are used to express the permissions associated to a role (PA relationship). The instances of *PolicyTimePeriodCondition* define the periods of time a user can activate a role. *RBACPermission* can be associated to a list of *SimplePolicyCondition* and *AssignerOperation* instances. The instances of *SimplePolicyCondition* are used to

describe the protected RBAC objects and the instances of *AssignerOperation* are used to describe approved operation on these objects.

5. RBPIM Framework

5.1. Overview

Several IETF works describe the implementation of policy-based network management tools using the PDP/PEP approach [9,10]. The IETF defines that the PEP and the PDP communicates using the COPS (Common Open Policy Service) protocol [10]. The COPS is an object-oriented protocol that defines a generic message structure for supporting the exchange of policy information between a PDP and its clients (PEPs). The COPS protocol defines two models of operation: outsourcing and provisioning. The choice between outsourcing and provisioning is supposed to have an important influence on the policy decision time. In environments where network polices are mostly static, one can suppose that the provisioning approach will be faster than the outsourcing approach. However, if external events trigger frequently policy changes, the performance in the provisioning approach can be significantly reduced, and outsourcing model could be a better choice. Also, it is possible to conceive hybrid approaches, combining the outsourcing and provisioning features.

The RBPIM framework described in this paper uses a "pure" outsourcing model. Figure 3 illustrates the main elements in the RBPIM framework. RBPIM framework adopts the PDP/PEP model using the outsourcing approach, i.e., the PDP carries most of the complexity and the PEP is comparatively light. In the RBPIM framework, the PEP is called Role-Based PEP (RBPEP). The Role-Based PDP (RBPDP) is a specialized PDP responsible for answering the RBPEP questions. Observe that the RBPDP has an internal database (called State DataBase) used for storing the state information of the RBPEP. The CIM/Policy Repository is a LDAP server that stores both: objects that represent network information such as users, services and network nodes and objects that represents policies (including the RBPIM model described in the section 4). The PCLS (Policy Core LDAP Schema) supplies the guidelines for mapping PCIM into LDAP classes [8]. RBPIM is mapped to a LDAP schema as defined by PCLS. The Policy Management Tool is the interface for updating CIM/Policy repository information and for administrating the PDP service.

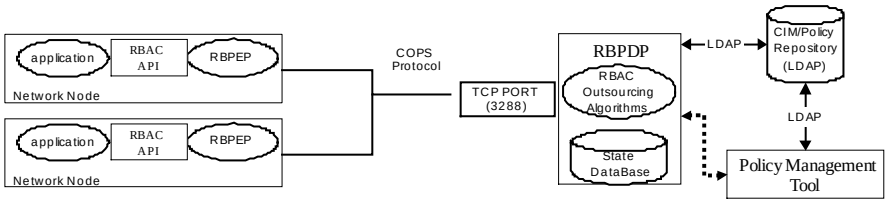

Fig.3. RBPIM Framework Overview

5.2. RBAC API's

As show in Figure 3, the RBPEP offers a set of API for permitting developers to build RBAC-aware applications without implementing a COPS interface. The RBPIM framework defines a set of five APIs:

- *RBPEP_Open* ()
- *RBPEP_CreateSession(userdn*:string; out *session*:string, *roleset[]*:string, *usessions*:int)
- *RBPEP_SelectRoles*(*session*: string, *roleset[]*:string; out *result*:BOOLEAN)
- *RBPEP_CheckAccess(session*: string, *operation*:string, *objectfilter[ ]*: string*; out result:BOOLEAN)*
- *RBPEP_CloseSession*(*session*:string)

The *RBPEP_Open* is the only API not related to RBAC. It establishes the connection between the PEP and the PDP. The API could be used by an application to ask the RBPEP to initiate the RBAC service. The RBPEP will process the API only if it is not already connected to the PDP.

The *RBPEP_CreateSession* API establishes a user session and returns the set of roles assigned to the user that satisfies the SSD constraints. This approach differs from the standard *CreateSession()* function proposed by the NIST because it does not activate a default set of roles for the user. Instead, the user must explicitly activate the desired roles in a subsequent call to the *RBPEP_SelectRoles* API. This modification avoids the need of the user to drop unnecessarily activated roles in order to satisfy DSD constraints. In order to call the *CreateSession* API, an application must specify the user through a DN (distinguish name) reference to a CIM Person object that represents the user (*userdn*). The RBPIM framework does not interfere in the authentication process. It supposes the application have already authenticated the user and mapped the user login to the corresponding entry in the CIM repository. Because the DSD constraints are imposed only within a session, the *CreateSession* API returns to the application the number of sessions already open by the user (*usessions*). Finally, the *session* parameter is a unique value generated by the RBPEP and returned to the application in order to be used in the subsequent calls.

The *RBPEP_SelectRoles* API activates the set of roles defined by the *roleset[]* parameter. This API evaluates the SSD constraints in order to determine whether the set of roles can be activated or not. If all roles in the set *roleset[]* can be activated, the function returns *result*=TRUE. The *SelectRoles* API, differently from the standard *AddActiveRole* function proposed by the NIST, can be evocated only once in a session. Also, in the RBPIM approach, the standard function *DropActiveRole* proposed by the NIST was not implemented. We have evaluated that allowing a user to drop a role within a session would offer too many possibilities for violating SSD constraints.

The *RBPEP_CheckAccess* API is similar to the standard *CheckAccess* function proposed by the NIST. This API evaluates if the user has the permission for executing the *operation* on the set of objects specified by the filter *objectfilter[]*. The *objectfilter*[] is a vector of expressions of type "*PolicyImplicitVariable=PolicyValue*" or "*PolicyExplictyVariable=PolicyValue*" used for discriminating one or more objects. In the current RBPIM version, the expressions in *objectfilter[]* are ANDed, i.e., only the objects that simultaneously satisfy all the conditions in the vector are considered for authorization checking. For example,

{"*PolicyDestinationIPv4Variable=192.168.2.3*",
"*Directory.Name=/usr/application*"}, specifies the object directory */usr/application* in the host 192.168.2.3. The *objectfilter[]* vector is confronted with the conditions specified by the *RBACPermission* objects in the RBPIM model. If the user has the right to execute the operation on all the objects that satisfy the *objectfilter[]* vector, the function returns *result*=TRUE. The RBPIM framework does not considers relationship between the CIM classes. The explicit variables expressions are evaluated independently, and must belong to the same object class in order to avoid an empty set of objects. To consider association between the CIM classes is a complex issue let for future studies. As an alternative, a condition "DN=value", based on the distinguished-name of an object, can be passed in the object filter to uniquely identify a CIM object, leaving to the application the responsibility of querying the CIM repository. The *RBPEP_CloseSession* terminates the user session, and informs to the PDP that the information about the session in the "state database" is no longer needed. The RBPEP_API is currently implemented in Java, and throws exceptions for informing the applications about the errors returned by the PDP. Examples of exceptions are: "RBPEP_client not supported", "non-existent session", "userdn not valid", etc.

5.3. COPS Messages

The COPS protocol version used in the RBPIM protocol is based on the RFC 2748. This section presents a short summary of the COPS protocol, please, refer to [10] for a more detailed description. Each COPS message consists of a common header followed by a number of typed objects. A field in the common header called "op-code" identifies the type of COPS message being represented. The RFC 2748 defines ten types of COPS messages. In order to understand how these messages are used, it is important to note that the COPS protocol assumes a stateful operation mode. Requests from the PEP are installed or remembered by the remote PDP until they are explicitly deleted. A PEP requests a PDP decision using the REQ (Request) message, and PDP responds to the REQ with a DEC (Decision) message (see Figure 4). The RPT message is used by the PEP to communicate to the PDP its success or failure in carrying out the PDP's decision. The DRQ message is sent by the PEP to remove a decision state from the PDP. A field in the common header called "client-type" identifies the policy client. The interpretation of all encapsulated objects that follow the common header is relative to the "client-type". A PEP sends an OPN (Open) message in order to verify if its specific client-type is supported by the PDP. The PDP responds with a CAT (Client-Accept) message or with a CC (Client-Close) message (the client is rejected). The CAT message specifies a timer in seconds (called KA timer), used for each side validating that the connection is still functioning when there is no other messaging. The PEP sends KA (Keep-Alive) messages to the PDP and the PDP echoes the PEP also using the KA messages. All the RBPEP APIs described in the previous section are mapped to COPS messages. The Figure 4 illustrates the RBPEP API to COPS mapping. The general structure of each COPS messages is also illustrated in the Figure 4.

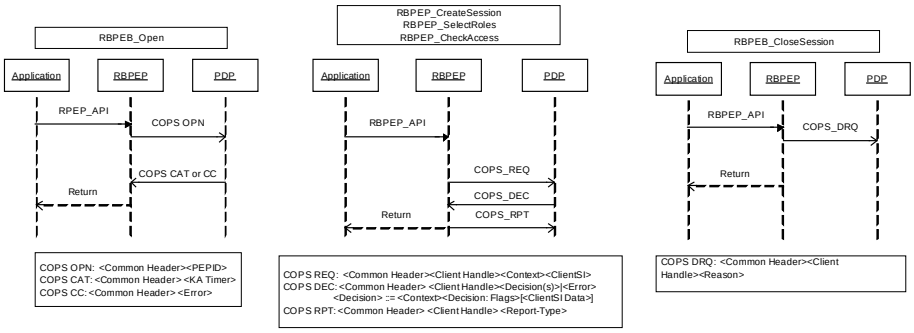

Fig.4. RBPEP API to COPS Mapping

The *RBPEP_Open* API is mapped to the COPS OPN, CAT and CC messages. In all messages, the <*Common Header*> uses the client-type 0x8000 for identifying a RBPEP client to the PDP. This value belongs to the range defined for enterprise specific client-types (0x8000 to 0xFFFF). The OPN message carries the specific object <PEPID> that identifies the RBPEP to the PDP. The <PEPID> is a symbolic string, usually representing the IP or the FQDN of the RBPEP host. If the PDP supports the RBPEP-type client, and the <PEPID> belongs to the list of authorized clients, it returns a CAT message; otherwise, it returns a CC message. The RBPEP will process the API only if it is not already connected to the PDP. The three APIs, *RBPEP_CreateSession*, *RBPEP_SelectRoles* and *RBPEP_CheckAccess* are mapped to the COPS REQ, DEC and RPT messages. In all messages, the object <*Client Handle*> encapsulates the *session* identifier. In the REQ message, the <*Context*> object identifies the API to the PDP and the <*ClientSI*> (Client Specific Information) objects are used to transport the parameters of the API. In the DEC message, the objects <*Decision*> are used to encapsulate the parameters returned by the PDP. In the RPT message, the <*Report-Type*> object carries the information about the success or failure of the RBPEP object implementing the decision delivered by the PDP. Because the RPT message is automatically generated by the RBPEP, the <*Report-Type*> always returns a success status. The *RBPEP_CloseSession* API is mapped to the COPS DRQ message. Like the other messages, the <*Client Handle*> object identifies the session. The <*Reason*> object transport a code that identifies the reason that justifies why the state (session) is being removed. The codes used by the <*Reason*> object are identified by the RFC 2748 [10].

6. Evaluation

In order to evaluate the performance the RBPIM framework, a Java based RBPDP and a RBPEP scenario simulator was implemented (see Figure 5). This prototype is available for download in [13]. In the evaluation scenario, twenty RBPEP clients request the RBPIM policy service provided by a single RBPDP. Each RBPEP keep a distinct COPS/TCP connection with the RBPDP. The RBPEP clients simulate typical access control scenarios created by text input files. Each line of these input files

corresponds to an API call presented in section 5.2. Several user sessions were created in the context of each RBPEP connection. For each connection served, the RBPDP generates an output file containing all COPS messages associated with the correspondent API call in the input file and the elapsed time from the instant of receiving the RBPEP's COPS message to the RBPDP's decision. In order to simulate different load scenarios, we have introduced a random delay between each API call contained in the input files. By varying the range of the random delay, we have created six load scenarios as shown in Figure 6. The load scenario "1" is the lightest scenario and the number "6" is the heaviest one. The former makes the RBPDP to receive 2.7 requests/second (average) and the latter increases this number to 40 requests/second (average). The Figure 6 presents the results obtained with the Java prototype, using a Pentium IV 1.5 Ghz 256 Mb RAM for hosting the RBPDP, and other identical machine for hosting the 20 RBPEP clients.

Initially, we defined a small set with five role objects hierarchically related and six permission objects, corresponding to a small set of departmental policies grouped in a single *RBACPolicyGroup* object. Each role and permission object has been defined considering a small set of three or four conditions combining implicit and explicit variables. Also, three SSD constraints and one DSD constraint were considered. One observes from the results that the *RBPEP_CreateSession* API correspond to the longest decision time. This is justified by the fact that this API prepares the state database by retrieving the list of the roles assigned to the user, free of SSD constrains.

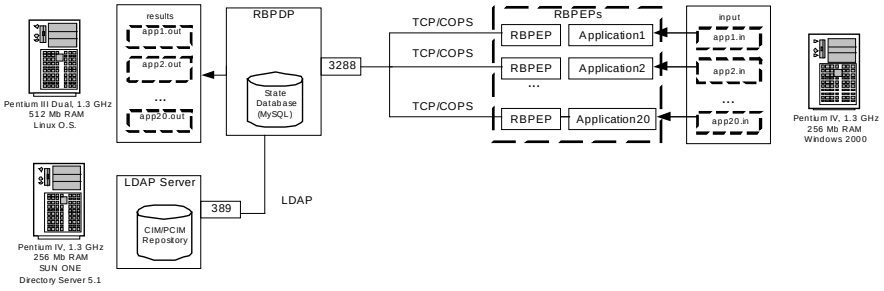

Fig.5. Simulation Scenario

After this initial test, the number of RBPIM objects has been increased. Each RBPIM object affects differently the response time of the *RBPEP_APIs*. Because of the flexibility introduced in the UA relationship by the RBPIM approach, the number of roles objects significantly affects the *RBPEP_CreateSession* API. Increasing the number of roles from five to twenty has almost doubled the average response time. By the other hand, the effect of increasing the number of SSD objects is not important. The response time of other APIs are not affected because the roles assigned to the user are saved in the state database for subsequent calls. The *RBPEP_SelectRoles* is almost imperceptible affected by the number of DSD objects and it is not affected by the other RBPIM objects. The *RBPEP_CheckAccess* should be affected by the number of permission objects associated to the roles. However, our tests shown that increasing the average number of permissions per role from two to ten has no significant effect in the response time. As a final remark, in all APIs, increasing the

number of conditions associated to a role or permission object has no significant effect, because the DNF or CNF conditions are transformed in a single LDAP query.

The results of the evaluation tests show the number of role (*RBACRole*) objects as the most important parameter affecting the response time in the RBPIM framework. The results also show reasonable response times considering the Java implementation and the CPU capacity of the machines used in the simulation. A response time of 50 ms for *RBPEP_CreateSession* (100 ms with twenty roles) in scenario 4 is a reasonable result for an API that is evocated only once in a session. Also, the *RBPEP_CheckAccess* average response time API has presented reasonable results for applications that requires decisions based on user events, and is not significantly affected by the number RBAC policy objects.

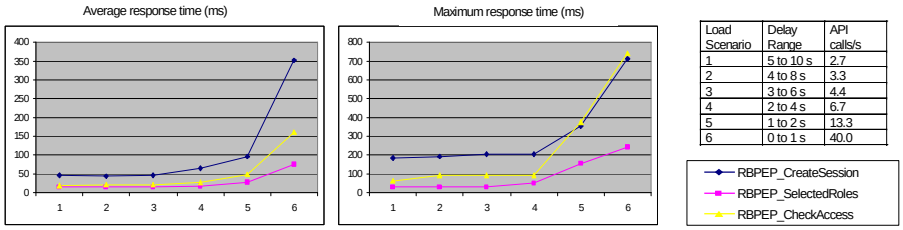

Fig.6. RBPDP decision time x API calls.

7. Conclusion

This paper has presented a complete policy based framework for implementing RBAC policies in heterogeneous and distributed systems. This framework, called RBPIM, has been implementing in accordance with the IETF standards PCIM and COPS, and also, the proposed NIST RBAC standard. The framework proposes a flexible RBAC model by permitting specify the relationship between users, roles, permissions and resource objects by combining Boolean expressions. The performance evaluation of the outsourcing model indicates that this approach is suitable for supporting RBAC applications that requires decisions based on user events. This paper does not discuss the problems that could rise if the PDP breaks. Future works must evaluate alternative solutions for introducing redundancy in the PDP service. Also, additional specifications are required for assuring a secure COPS connection between the PDP and the RBPEPs. These studies will be carried out in parallel with the evaluation of provisioning and hybrid approaches for implementing the RBPIM framework. Also, some important PCIMe modifications must be taken into account in a revised version of the RBPIM information model. Finally, some studies are being developed for evaluating the use of the RBPIM framework for QoS management based on RBAC rules.

References

1. D.F. Ferraiolo, R.S. Sandhu, G. Serban, “A Proposed Standard for Role-Based Access Control”, ACM Transactions on Information System Security, Vol. 4, No. 3, August 2001, pp. 224-274.
2. L.S. Bartz, “LDAP Schema for Role Based Access Control”, IETF Internet Draft, expired, October 1997.
3. L.S. Bartz, “CADS-2 Information Model”, not published, IRS: Internal Revenue Service, 2001.
4. Distributed Management Task Force (DMTF), “Common Information Model (CIM) Specification”, URL: http://www.dmtf.org.
5. B. Moore, E. Elleson, J. Strasser, A. Weterinen, “Policy Core Information Model”, IETF RFC 3060, February 2001.
6. B. Moore, E. Elleson, J. Strasser, A. Weterinen, “Policy Core Information Model Extensions”, IETF RFC 3460, February 2003.
7. W. Yeong, T. Howes, S. Killie, “LightWeight Directory Access Protocol”, IETF RFC 1777, March, 1995.
8. J. Strassner, E. Ellesson, B. Moore, R. Moats, "Policy Core LDAP Schema", IETF Internet Draft, January 2002.
9. R. Yavatkar, D. Pendarakis, R. Guerin, “A Framework for Policy-based Admission Control”, IETF RFC 2753, January 2000.
10. D. Durham, Ed., J. Boyle, R. Cohen, S. Herzog, R. Rajan, A. Sastry, The COPS (Common Open Policy Service) Protocol, IETF RFC 2748, January 2000.
11. Y. Snir, Y. Ramberg, J. Strassner, R. Cohen, B. Moore, "Policy QoS Information Model", IETF internet-draft, November 2001.
12. OASIS, "eXtensible Access Control Markup Language (XACML) -Version 1.03”, OASIS Standard, 18 February 2003, URL: http://www.oasis-open.org
13. RBPIM Project WebSite. http://www.ppgia.pucpr.br/~jamhour/RBPIM