A Policy-Based Framework for RBAC

Ricardo Nabhen, Edgard Jamhour, and Carlos Maziero

Pontifícia Universidade Católica do Paraná, PUCPR, PPGIA
Rua Imaculada Conceição 1155, CEP 80215-901, Curitiba, Brazil
{rcnabhen,jamhour,maziero}@ppgia.pucpr.br

Abstract. This paper presents a PCIM-based framework for storing and enforcing RBAC (Role Based Access Control) policies in distributed heterogeneous systems. PCIM (Policy Core Information Model) is an information model proposed by IETF. It defines a vendor independent model for storing network policies that control how to share network resources. PCIM is a generic core model. Application-specific areas must be addressed by extending the policy classes and associations proposed by PCIM. In this context, this paper proposes a PCIM extension, called RBPIM (Role-Based Policy Information Model), in order to represent network access policies based on the RBAC model. A RBPIM implementation framework based on the PDP/PEP (Policy Decision Point/Policy Enforcement Point) approach is also presented and evaluated.

1 Introduction

The Policy Core Information Model (PCIM) is an information model proposed by IETF and DMTF [5]. PCIM is based on the Common Information Model (CIM), proposed by DMTF (Distributed Management Task Force) [4]. The CIM information model permits to represent both, network resources and policies definitions for managing these resources. PCIM can be considered as a sub-set of the CIM model, which comprises only the policy information. PCIM is a generic 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 [12].

This paper describes a PCIM extension for role-based 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]. RBPIM is an open framework for supporting the development of applications that shares a common set of RBAC policies. In the RBPIM approach, the RBAC policies refers to objects represented in a CIM repository. The RBPIM framework is implemented using a PDP/PEP approach [10]. The PDP (Policy Decision Point) is a network policy server responsible for supplying policy information (or policy decisions) for network devices and applications. The PEP (Policy Enforcement Point) is the policy client (usually, a component of the

M. Brunner and A. Keller (Eds.): DSOM 2003, LNCS 2867, pp. 181-193, 2003.

network device/application) responsible for enforcing the policy. 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 reviews some related works. Section 3 presents the RBPIM information model. Section 4 presents the RBPIM framework implemented using the outsourcing model, as defined by the COPS standard [11]. Section 5 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 Related Works

Recent works explore 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 [13]. 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. Policies are written in XACML by policy administrators and made available for PDP servers. The request/response language is used for supporting the communication between PEP clients and PDP servers. A PEP queries a PDP whether a particular access should be allowed using XACML, and the PDP describes answers to those queries also using XACML. The RBPIM framework described in this paper also uses the PDP/PEP approach. However, our approach differs from XACML from several points. First, the RBPIM uses a standard COPS protocol for supporting the PEP/PDP communication. Second, the information model used for describing policies is based on a PCIM extension. Third, RBPIM has been implemented to support 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 [12]. 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 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. In the CADS-2 approach, the permissions are ex-

pressed in terms of "having access to services". A service has subordinated objects called *ServiceAccessPoint* (SAP), which are, in fact, the protected objects in the RBAC model.

The RBIM model described in the section 3 uses the idea of mapping roles to users using Boolean expressions, proposed by the CADS-2 model. Note that this approach offers an additional degree of freedom for creating RBAC policies because the relationship between user and roles can be expressed through Boolean expressions instead of a direct mapping. However, the RBPIM work differs from the CADS-2 model from several points. Many features have been introduced in order to support the other elements of the RBAC model, such as hierarchy of roles, DSD (Dynamic Separation of Duty) and SSD (Static Separation of Duty), defined in the proposed NIST standard [1], but not present in the CADS-2 model. Also, in the RBPIM, Boolean expressions are built using the PCIM extensions proposed in RFC 3460, not available in the original release of the CADS-2 model.

3 RBPIM: The Role-Based Policy Information Model

Fig. 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.

In a *PoliceRule*, 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* (the attributes are not shown in the Fig. 1) helps to create condition expressions. In DNF, conditions within the same group number are ANDed and groups are ORed. In CNF, conditions within the same group are ORed and groups are ANDed.

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. In the PCIMe, *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. 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. Please, refer to the RFC 3460 for more details about this approach.

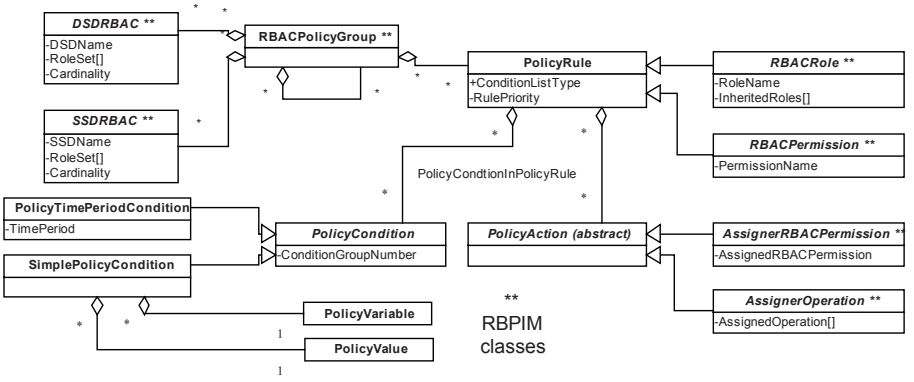

Fig. 1. PCIM/PCIMe class hierarchy and RBPIM extensions

The RBPIM class hierarchy is also shown in the Fig. 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* , not shown in the figure). The *RBACPolicyGroup* class (specialization of *PolicyGroup*) is used to group the information of the constrained RBAC model. As shown in Fig. 1, the approach in the RBPIM model consists in using two specializations of *PoliceRule* for building the RBAC model: *RBACRole* (for representing RBAC roles) and *RBACPermission* (for representing RBAC permissions). *RBACRole* and *RBACPermission* follows the same semantic proposed by the standard class *PoliceRule*, i.e., if the set of the conditions associated to the class are evaluated true, than the corresponding set of actions are executed. This approach offers a flexible method for representing the relationship between user and roles and between roles and permissions using DNF or CNF expressions. *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. The SSDRBAC and DSDRBAC classes permit to describe static and dynamic separation of duty constraints. The RBPIM model defines SSD and DSD with two attributes: a *roleSet[]* that includes two or more roles, and a *cardi-*

nality greater than one indicating the maximum combination of roles in the set a user can be assigned (SSD) or activate within a session (DSD), e.g., for constraining a user to assume the roles "r1" and "r2", one must define a set {r1, r2} with cardinality 2 (the user can assume cardinality-1 roles in the set).

The example in Fig. 2 to illustrates the use of the RBPIM model. The *RBACRole* in the figure was called "role1". The attribute *InheritedRoles* is used for expressing the Hierarchical RBAC, i.e., the role "role 1" inherits the permissions of roles "role2" and "role3". The UA relationship for "role1" is defined as:

IF "*PolicySourceIPv4Variable* MATH 192.168.10.0/24" AND "*Person.BusinessCategory* MATCH CT*" AND "*PolicyTimePeriodCondition* MATCH [20020701,20020831]".

The PA relationship is defined by the reference to the permission object "*App_Directory*", shown in the Fig. 2. This permission defines the operations {*R,W*} are approved when *Directory.Name* MATH "*/etc/application*". Observe how the use of explicit variables permits leveraging the information of existing CIM repositories. As well as PCIM, the RBPIM model is implementation neutral. RBPIM mapping to LDAP schema has been implemented according to the IETF standard PCLS [9]. Please, refer to [9] for a detailed description of the PCIM and LDAP mapping.

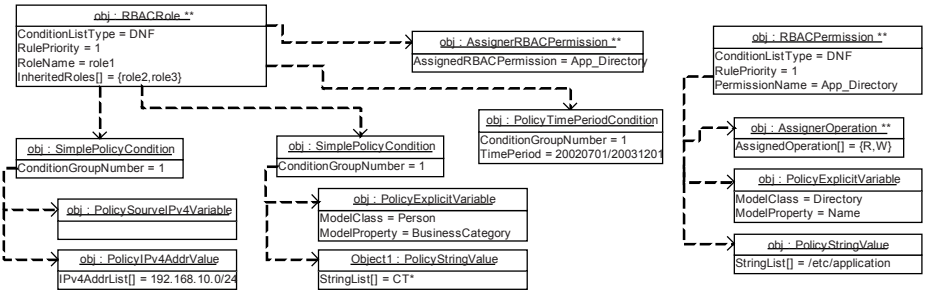

Fig. 2. Object instances of the RBPIM model

4 RBPIM Framework

The RBPIM Framework follows the PDP/PEP approach [10]. The IETF defines that the PEP and the PDP communicates using the COPS (Common Open Policy Service) protocol [11]. The COPS protocol defines two models of operation: outsourcing and provisioning. In the outsourcing model, the PDP receives policy requests from a network node (PEP), and determines whether or not to grant these requests. By the other hand, in the provisioning model, rather than responding to PEP events, the PDP prepares and "pushes" configuration information to the PEP. This takes place as a result of external events (unrelated to the PEP) such as change of applicable policy, time of day, expiration of account quota, or information from third party (non-PEP) signaling. In the provisioning approach, a PEP can answer to events based on the locally stored policy information. Fig. 3 illustrates the main elements in the RBPIM framework. RBPIM framework adopts the PDP/PEP model using a "pure" outsourcing approach, i.e., the PDP carries most of the complexity and the PEP is comparatively light.

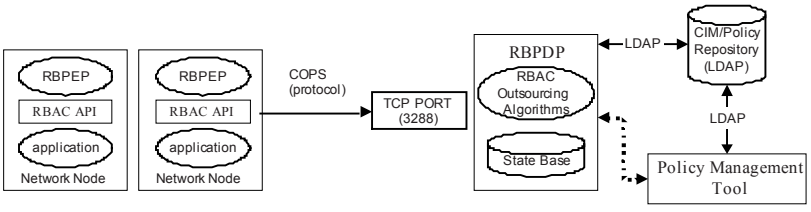

Fig. 3. RBPIM Framework Overview.

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. 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 3). The Policy Management Tool is the interface for updating CIM/Policy repository information and for administrating the PDP service.

The RBPEP is basically a software library that simplifies the task of building "RBAC-aware" applications. It offers a high level programming interface for mapping the RBAC APIs to COPS messages addressed to the RBPDP. The RBAC API's used in the RBPIM framework are based on the RBAC functional specifications described in the proposed NIST standard [1]. The NIST standard defined five supporting system functions[1]: *CreateSession* (*user, session*): creates the *user* session and provides the user with a default set of roles. The *session* identifier is supposed to be generated by the underlying system. *AddActiveRole*(*user, session, role*): adds a *role* as an active role for the current *session*. *DropActiveRole* (*user*, *session*, *role*): deletes a role from the active role set for the current session. *CheckAccess* (*session, operation, object, out: result*): determines if the *session* subject has permission to perform the requested *operation* on an *object*. The *result* is BOOLEAN. *DeleteSession*(*user, session*): deletes a given session with a given owner user. Based on the NIST supporting system functions proposed by NIST, the RBPIM framework defines a set of five API's:

- 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)

As mentioned before, RBPEP maps the RBPEP calls to COPS messages. The COPS for outsourcing model defines several messages, but the most important are REQ (Request) and DEC (Decision), which are used to encapsulate the

[1] In the NIST standard, supporting system functions refer to the process of creating a session, activating roles and checking access permissions.

RBPEP_CreateSession, RBPEP_SelectRoles and RBPEP_CheckAccess API's. The REQ message encapsulates the parameters passed by the PEP to the PDP, and the DEC message encapsulates the messages returned by the PDP to the PEP.

The RBPDP module implements a set of algorithms triggered by the COPS messages sent by the RBPEPs. These algorithms interpret the RBAC policies stored in the CIM/Policy repository and the state information of the RBPEP sessions (stored in a relational state-database), and answer the RBPEP using the COPS protocol. Note that the state-database is a database internal to the *RBPDP* and its information is not described in the RBPIM model. The RBPEP API and the respective PDP algorithms are described next in this section. The most important algorithms implemented by the *RBPDP* are those related to the *RBPEP_CreateSession*, *RBPEP_SelectRoles* and *RBPEP_CheckAccess*. Some obvious error treatment have been omitted in order to simplify the presentation of the algorithms.

RBPEP_Open. The *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.

RBPEP_CreateSession. The *CreateSession* API establishes a user session for the user and returns the set of roles assigned to the user that satisfies the SSD constraints (*roleset*[]). This approach differs from the standard *CreateSession()* function 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 also returns the number of sessions already opened by the user (*usessions*). The application can abort the *CreateSession* process by calling *DeleteSession,* if it does not desire to serve a user with sessions already open. 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. Presently, the approach defined by the RBPIM framework consists in using a *RBACPolicyGroup* object for grouping the RBAC objects. In the CIM/Policy repository, the *RBACPolicyGroup* objects are associated to "organization units" by DIT containment. By using the attribute organizational unit ("OU") in the CIM *Person* object, the algorithm determines the corresponding *RBACPolicyGroup* object associated to the user. The algorithm for the *RBPEP_CreateSession* API is defined as follows:

1. If the session already exists in the state database then returns a <Error> object in the DEC message. Otherwise, go to Step 2.
2. Determine **Ra** as the list of "candidate" roles (*RBACRoles* objects) associated to the *RBACPolicyGroup* object in the organization unit of the user.

3. Determine **Rb** as the list of RBAC roles, subset of **Ra**, which conditions are satisfied by the CIM Person object pointed by *userdn*.
4. Determine **Rc** as the list of RBAC roles, subset of **Rc**, that satisfy the time constraints defined by the *PolicyTimePeriodCondition*.
5. Determine **Rd** as the list of inherited roles indicated by the attribute *InheritedRoles* of all RBACRole objects ∈ **Rb**.
6. Determine **Re** as the disjoint union: **Rc** ∪* **Rd**.
7. Determine **SSD** as the list of SSDRBAC objects associated to the *RBACPolicyGroup* object in the organization unit of the user.
8. Determine **Rf** by removing from **Re** the roles that are constrained by **SSD**. The roles with lowest priority (*RulePriority* attribute inherited by *RBACRole* from *PolicyRule*) are removed first, until the Cardinality attribute of all **SSD** constraints is satisfied.
9. Create in the state database a record with the session, *userdn*, the *roleset[]* defined by **Rf** and *status*=Phase1 and sends a DEC message with the parameters *roleset* and *usessions* encapsulated in <Decision> objects.

RBPEP_SelectRoles. The 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, can be evocated only once in a session. Also, in the RBPIM approach, the standard function *DropActiveRole* 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_SelectRoles* API activate in a session the set of roles defined by the *roleset[]* argument. The *SelectRoles* API will activate the roles only if all roles in *roleset[]* are presented in the session database and all of them are free of DSD constraints. The algorithm for the *RBPEP_SelectRoles* API is defined as follows:

1. If the session already exists in the state database with status=Phase1 go to Step 2. If it doesn't, then returns a <Error> object in the DEC message.
2. Determine **R** as the list of references to roles (RBACRoles) objects associated to the session in the state database.
3. If *roleset*[] ⊄ **R** then sends a DEC message indicating the operation has been denied. Otherwise, go to Step 4.
4. Determine **DSD** as the list of DSDRBAC objects associated to the RBACPolicyGroup object in the organization unit of the user.
5. If *roleset[]* violates the **DSD** constraints then sends a DEC message indicating the operation has been denied. Otherwise, go to Step 6.
6. Update the state database by storing *roleset[]* as the list of active roles in the session and define status=Phase2. Then, sends a DEC message with *result=true* encapsulated in a <*Decision*> object.

RBPEP_CheckAccess. The 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. 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. Explicit variable conditions may define one or more CIM objects. For example, {"*DataFile.Readable=true*", "*DataFile.Name=*.doc*"} will probably define a set of objects instead of a single object. Say Φ as the set of objects defined by the *objectfilter[]* in the *RBPEP_CheckAccess* API. The CIM objects in the Φ can be retrieved by a single LDAP query which filter is based on the *objectfilter[]* conditions. By the other hand, the *RBACPermission* objects associated to the roles activated by the user may also contain conditions based on implicit and explicit variables and, therefore, define another set of CIM objects, say ψ, also retrieved by a single LDAP query. The *RBPEB_CheckAccess* API will return true if $\Phi \subseteq \psi$. Because ψ can be very large, the condition $\Phi \subseteq \psi$ is replaced by the equivalent expression $\Phi \subseteq \theta$, where $\theta = \psi \cap \Phi$. The θ set can also be determined by a single LDAP query, by defining a LDAP filter that combines the conditions presented in the *objectfilter[]* and the *RBACPermission* associated conditions. The implicit variables conditions such as {*"PolicyDestinationIPv4Variable=192.168.2.3*"} are not used for creating the LDAP queries, because implicit variables doest no correspond to objects in the CIM repository. Instead, they are used for eliminating the *RBACPermission* objects that does not satisfy the implicit variables in the *objectfilter[]* vector. The algorithm for the *RBPEP_CheckAccess* API is defined as follows:

1. *Verify* if the session exists in the state database with *status=Phase2*. If it doesn't than returns an <*Error*>. Otherwise, go to Step 2.
2. *Determine* **Ra** as the list of active roles in the session (references to objects *RBACRoles*).
3. *Determine* **Rb** as the subset of **Ra** where the roles satisfy the time constraints (*PolicyTimePeriodCondition* objects*)*.
4. *Determine* **Pa** as the list of permission (*RBACPermission*) objects associated to the roles $\in$ **Rb**.
5. *Determine* **Pb** as the subset of **Pa** that includes only the permission objects which implicit variables conditions satisfy the implicit variables conditions in the *objectfilter[]* vector passed by the RBPEP API.
6. *Determine* **O** as the list of operations (*AssignerOperation*) objects associated to the permission object $\in$ **Pb**.
7. *Determine* **Pc** as the subset of **Pb** that include only the permission objects where the *operation* passed by the RBPEP API $\in$ **O**.

8. *Determine* θ as the list of CIM objects that simultaneously satisfy the conditions defined by the explicit variables in the *objectfilter[]* and the list of conditions (*SimplePolicyCondition*) associated to the permission objects $\in$ **Pc**.
9. *Determine* Φ as the list of CIM objects that satisfy the explicit variables in the *objectfilter[]*.
10. S*ends* a DEC message with *result* = true if $\Phi \subseteq \theta$, otherwise, sends *result*=false.

RBPEP_CloseSession. The *CloseSession* API 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_APIs are 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 Evaluation

There are some important questions to evaluate in the RBPIM model. First, the strategy adopted for representing UA (User Assignment) and PA (Permission Assignment) relationship, based on implicit and explicit variables, is very flexible, but can lead to a long policy decision response time. Other important issue is determining if the outsourcing model is capable of a reasonable response time.

In order to evaluate the performance the RBPIM framework, a Java based RPPDP and a RBPEP scenario simulator was implemented. This prototype is available for download in [14]. In the evaluation scenario, twenty RBPEP clients request the RBPIM policy service provided by a single PDP. Each RBPEP keep a distinct COPS/TCP connection with the PDP. Several user sessions were created in the context of each RBPEP connection. In order to simulate different load scenarios, we have introduced a random delay between each API call evocated by the RBPEP client. By varying the range of the random delay, we have created six load scenarios as shown in Fig. 4. The figure also presents the results obtained using a Pentium IV 1.5 Ghz 256 Mb RAM for hosting the PDP, 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. The result of this simple scenario is presented in Fig. 4. 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.

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 the other RBPIM objects do not affect it. By analyzing the algorithms described in section 4, one could suppose the number of permission objects associated to the roles should affect the *RBPEP_CheckAccess*. 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. The justification for this result is that steps 8 and 9 in the *CheckAccess* algorithm are solved with a single LDAP query that creates a complex filter combining the conditions of all permission objects. Similarly, 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.

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 the other RBPIM objects do not affect it. By analyzing the algorithms described in section 4, one could suppose the number of permission objects associated to the roles should affect the *RBPEP_CheckAccess*. 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. The justification for this result is that steps 8 and 9 in the *CheckAccess* algorithm are solved with a single LDAP query that creates a complex filter combining the conditions of all permission objects. Similarly, 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.

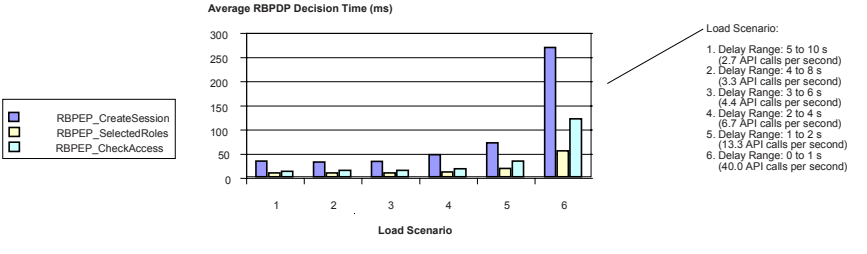

Fig. 4. Average Time RBPDP decision x API calls

6 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, which permits specifying the relationship between users, roles, permissions and 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. 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, (2001) 224-274.

[2] L.S. Bartz: LDAP Schema for Role Based Access Control, IETF Internet Draft, expired, (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 (2003).

[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 2001.

[7] W. Yeong, T. Howes, S. Killie: LightWeight Directory Access Protocol. IETF RFC 1777, March, 1995.

[8] Distributed Management Task Force (DMTF): Guidelines for CIM-to-LDAP Directory Mappings. whitepaper, May 8th, 2000, URL: http://www.dmtf.org

[9] J. Strassner, E. Ellesson, B. Moore, R. Moats: Policy Core LDAP Schema. IETF Internet Draft, January 2002.

[10] R. Yavatkar, D. Pendarakis, R. Guerin: A Framework for Policy-based Admission Control. IETF RFC 2753, January 2000.

[11] 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.

[12] Y. Snir, Y. Ramberg, J. Strassner, R. Cohen, B. Moore: Policy QoS Information Model. IETF internet-draft, November 2001.

[13] OASIS: eXtensible Access Control Markup Language (XACML) -Version 1.03. OASIS Standard, February 2003, URL: http://www.oasis-open.org
[14] RBPIM Project WebSite. URL:http://www.ppgia.pucpr.br/~jamhour/RBPIM, (2003).