# Diplomaterv

**High performance dynamic authorization service for mission critical enterprise environments**

**Nagyteljesítményű dinamikus authorizációs szolgáltatás missziókritikus nagyvállalati környezetekhez**

**Bősze Tibor Attila**
Nappali tagozat
Műszaki informatika szak
V. évfolyam
**2006**

# Nyilatkozat

Alulírott *Bősze Tibor*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

.............................................

Bősze Tibor

# Table of Contents

# Kivonat

Az informatikai rendszerek folyamatos növekedéssel válaszolnak az egyre szigorodó üzleti elvárásokra, így a rendszergazdák és biztonsági felelősök feladata is egyre bonyolódik. A konzisztens biztonsági házirend fenntartása érdekében a kiterjedt infrastruktúra minden egyes komponensén koordinált módon kell adminisztrálniuk a felhasználói fiókokat és jogosultságokat. A folytonos méretbeli és komplexitásbeli növekedés következtében a felhasználó-kezelés hagyományos módja már nem alkalmazható.

A központosított felhasználó-azonosítás („Ki vagy?") és jogosultság-ellenőrzés („Meg szabad ezt tenned?") mára már alappillérei lettek bármelyik komoly nagyvállalati hozzáférés-vezérlési megoldásnak: Az összes felhasználói jogosultság és szerepkör központosított menedzsmentje és a felhasználók rendszerben való ténykedéseinek részletes naplózása kulcsszerepet játszik egy sikeres és biztonságos nagyvállalati környezet kialakításában. Bár a központosítás által leegyszerűsödnek az adminisztrációs feladatok, egy ilyen nélkülözhetetlen és teljesítménykritikus szolgáltatás centralizálása könnyen vezethet csökkent áteresztőképességhez és más rendszerkomponensek rossz kihasználtságához.

A jogosultság-ellenőrző szolgáltatás mindig is kritikus része volt az informatikai rendszereknek, ugyanis e szolgáltatás kiesése az egész rendszer rendelkezésre-állására kihat. Az authorizációs szolgáltatás teljesítménye szintén kihat az üzleti áteresztőképességre – tehát az ügyfelek kiszolgálási képességére, hiszen könnyedén szűk keresztmetszetté válhat a kritikus üzleti folyamatokban.

Ezek az igények egy olyan robosztus, nagyteljesítményű jogosultság-ellenőrző szolgáltatást kívánnak, amely jogosultsági rendszere elég letisztult a jó teljesítmény és kezelhetőség biztosításához, ám elég rugalmas és bővíthető – plug-in-okkal vagy akár külső kiértékelő-modulokkal - a komplex jogosultsági szabályrendszerek kényelmes megfogalmazásához.

Az üzleti elvárások olyan jogosultság-ellenőrző folyamatot igényelnek, ami képes környezetfüggő házirendek kiértékelésére. Ezek a házirendek környezeti változók, futásidejű feltételek alapján szabályozzák a védett erőforrásokhoz való hozzáférést, viszont továbbra is elvárás a magas teljesítmény, hogy a rendszer többi komponense tervezett terhelés mellett futhasson.

Ez a diplomamunka egy olyan jogosultság-ellenőrző szolgáltatás tervezését és implementálását tűzi ki céljául, ami megfelel ezeknek az elvárásoknak. Ez az elméleti oldalon egy olyan új biztonsági modell bemutatását jelenti, ami képes a dinamikus hozzáférési szabályok hatékony megfogalmazására. A biztonsági modell az RBAC modellnek egy olyan kibővítése, amely képes a dinamikus viselkedés - így a kontextus alapú hozzáférési döntések - letisztult és lényegretörő kezelésére.

Maga a megoldás az IBM Tivoli Access Manager robusztus és skálázható rendszerét használja. Ez egy rengeteg hasznos funkcióval és bővítési lehetőséggel rendelkező Common Criteria EAL-3 követelményeknek megfelelő moduláris hozzáférés-vezérlő architektúra. A Tivoli Access Manager RBAC változata számos hasznos kiegészítést

nyújt a hagyományos szerepkör alapú hozzáférés-vezérléshez képest, de a teljesítmény-problémák kiküszöbölése érdekében néhány korlátozást is bevezet.

A dinamizmus kezelésének alapvető ötlete és a Tivoli Access Manager választása után egy további fejlesztési fázis történt. Az implementáció és a biztonsági modell együttes továbbfejlesztése a megoldás finomhangolását tűzte ki céljául – a rugalmasság, a teljesítmény és kezelhetőség szempontjából.

# Abstract

As IT systems extend their infrastructures to meet business needs, system administrators and security masters are faced with having to manage user accounts and access permissions within each of the components[1] in a coordinated manner in order to maintain the integrity of security policy enforcement. Induced by continuous growth of IT infrastructures in both scale and complexity, this legacy approach to user and security management has become inadequate.

Centralizing authentication and authorization has now become the cornerstone of any solid enterprise-wide access management solution: centralized management of each user's access permissions, roles, and detailed audit logs of their actions and activities is fundamental to the success and security of the large-scale enterprise. While it simplifies management, centralizing such indispensable and performance critical services can easily become the reason for degraded throughput accompanied by underutilization of other components.

The authorization service has always been a critical component of IT infrastructures, since the unavailability of the access control service would impact the whole system's availability. The performance of the authorization service also has a direct effect on business throughput as it could easily become a bottleneck in any critical business process.

These needs require the implementation of a robust, high performance authorization service with straightforward authorization logic simple enough to provide qualities in both performance and manageability, yet providing the ability to involve flexible plug-ins or external rule engines to enable the authorization process to satisfy more complex policies.

Business needs require the authorization process to handle security policies that apply to the protected objects based on a variety of runtime conditions, such as context and environment, while still providing the performance to allow the desired utilization of the rest of the system.

This master thesis addresses the challenge of designing and implementing an authorization service meeting these needs. On the theoretical side it introduces a custom security model that supports the definition of dynamic access control rules. The security model extends the de facto industry standard RBAC[2] model to include features capable of handling dynamic behavior like context based access decisions in an intuitive and straightforward way.

The implementation itself utilizes the robust and scalable IBM Tivoli Access Manager infrastructure, a Common Criteria EAL-3 certified modular authorization architecture with many useful features and extension points. The adoption of RBAC used in Tivoli Access Manager provides useful extension to the legacy Role Based Access Control model but also introduces a few constraints to overcome performance issues.

[1] the systems and services to be accessed by the users
[2] acronym for the Role-Based Access Control security model

Given the basic approach to handle dynamism and the choice of Tivoli Access Manager, another development iteration of co-designing the implementation and the security model has been made to fine-tune the solution for flexibility, performance and manageability.

# 1. Introduction

## *1.1. IT security in enterprise environments*

Continuity of operations and correct functioning of information systems is important to most businesses. Threats to computerized information and processes are threats to business quality and effectiveness. The objective of IT security is to reduce significant threats to an acceptable level.

Securing enterprise IT environments is a more challenging task – in both scale and complexity - than protecting a handful of computers. Applications have moved from single systems to a cluster of co-operating modules across different systems interconnected via local networks.

Business continuity - one of the most crucial aspect of the large scale enterprise - might be affected by the corruption of any of those components. Flexibility - the ability to appropriately react to changing business needs – is another indispensable feature in enterprise environments, including - but not limited to - the flexibility of the security services that protect the IT infrastructure.

### 1.1.1. The importance of manageability

The term 'IT security' covers far more than deploying firewalls or antivirus software. Decades ago, corporate security has meant just a touch more than locking a physical door to protect the mainframe – hardware storing confidential business data - from the unauthorized. Even in the past few years, IT security has usually meant just a little more than network security. Configuring firewalls to filter servers' ports and patching applications to circumvent buffer-overflow vulnerabilities have been the typical tasks of a security specialist. Protection against a skilled, malicious computer specialist – a hacker – has been the primary goal when securing IT infrastructures.

However, actual statistics show that the majority of electronic crime is committed by insiders. Data tampering or theft has become a needlessly frequent issue since fired employees often have access to system resources even after the labor relation has ceased. The creation of test accounts provides another common security problem: A local administrator could simply set up user accounts for test purposes and forget to delete them after the test or maintenance task is accomplished. This is an obvious security risk, since these accounts typically consist of easy to guess usernames and weak passwords.

In both cases, if a single, authentic source of identities and access permissions is used by all workstations and servers, the problem can easily be managed by checking the centralized user registry and authorization database for compliance with the security policies. Revoking an employee's login permission – that means disabling all his accounts - is a single command issued only once, at the central repository, and the change is reflected immediately on all the components that authenticate from the shared data source. Improved manageability of user and access data not only improves administrator productivity but also decreases the risk of errors and misconfigurations. Manageability also guarantees flexibility of the security infrastructure by improving the ability to react to changes in a quick and precise way.

### 1.1.2. Performance and availability

Centralizing access control services usually means externalizing authentication and authorization decisions in every single component or subsystem of the IT infrastructure. The services are located on a separate, preferably dedicated physical server, and are accessed over the network by the components that request authentication and authorization decisions.

Network data transfer causes processing overhead and an increase in response time in both endpoints, which could impact the whole systems performance. In addition, in the case of centralized authentication and authorization, the dedicated server needs to be able to serve all the requests of all the components exposed to the users. Whenever an authorization request is issued, the business process – a potentially critical process – is paused until the authorization decision is evaluated and sent back to the requesting component.

Since every server uses a single source of authorization decisions, the dedicated access control server can easily become a bottleneck in the business process, causing other components to lag. From the user’s view, the system is not responding within the expected time, hence, the business process seems to be sluggish. The servers, though, with the exception of the “exhausted” authorization component, are underutilized. The system is not able to serve the needed amount of user requests, the business throughput – and so the profit - decreases. Appropriate performance of the central authorization service is therefore a key prerequisite of business success in a large-scale e-Business environment.

Beside the risk of becoming a bottleneck, a centralized service also features a single point of failure: should the central server fail – e.g. because of a hardware defect – none of the components relying on user authorization is able to complete the business processes, which renders the whole system unavailable. Therefore, a redundant, fault tolerant architecture of such an indispensable service is a must for every mission critical production environment.

The performance and availability of the authorization service has a direct effect on the success of the enterprise.

## 1.2. *Role-Based Access Control*[3]

Role-Based Access Control models have received broad support as a generalized approach to access control, and are well recognized for their advantages in performing large-scale authorization management. In the past decade, vendors have begun implementing RBAC features in their database management, security management,

[3] The term has been introduced in 1992 by Ferraiolo and Kuhn.

and other software products without a general agreement about RBAC components, features and terminology.

The inconsistency caused by the co-existence of several early RBAC models has made the standardization of Role-Based security necessary. The standard consists of the RBAC Reference Model and the RBAC Functional Specification.

The reference model defines the scope of features that comprise the standard and provides a consistent terminology in support of the specification. The RBAC System and Administrative Functional Specification defines functional requirements for administrative operations and queries for the creation, maintenance, and review of RBAC sets and relations, as well as for specifying system level functionality in support of session management and the access control decision process.

Beside the core RBAC components, the standard also provides a standardization for some widely used extensions to the base RBAC concept, which are to be outlined in a separate subsection.

### 1.2.1. Background

The concept of roles has been implemented in software applications for at least 25 years. In the last decade, Role-Based Access Control has emerged as a fully functional model as mature as the concepts of Mandatory Access Control or Discretional Access Control. Now, Role-Based Access Control is the de facto industry standard for large-scale access management.

The roots of Role-Based Access Control include the use of groups in UNIX and other multi-user operating systems to handle access permissions of many "similar" users in a more sophisticated way. This concept of privilege groupings has been reflected in database management systems as well, simplifying administration tasks for sensitive databases.

Later on, advanced concepts like separation of duty[4] as described in former security models[5] have been streamlined and adopted into the RBAC concept to meet business needs and comply with government and corporate security policies, standards and regulations.

The extended concept of RBAC embodies many modern notions in a single access control model, where the particular extensions provide added functionality in terms of roles and role hierarchies, role activation, constraints on user/role membership and role set activation. Some of these concepts are described later on.

The RBAC model has been shown to be "policy-neutral" in the sense that by using role hierarchies and constraints, a wide range of security policies can be expressed. Security administration is also greatly simplified by the use of roles to organize access privileges.

[4] Separation of duty restricts the activation of conflicting roles at the same time
[5] For example Clark and Wilson, 1987

For example, if a user moves to a new function within the organization, the user can simply be assigned to the new role and removed from the old one, whereas in the absence of an RBAC model, the user's old permissions would have to be individually revoked, and new permissions would have to be granted one-by-one. Moreover, without RBAC, enforcing a global change to the corporate policy could potentially mean applying the same modifications to hundreds of user accounts. In addition, administration constraints may need to be enforced to prevent information misuse and prevent fraudulent activities.

A typical authorization constraint, broadly relevant and well recognized, is separation of duties. The intent of separation of duties (SoD) is reducing the risk of fraud by not allowing any individual to have sufficient authority within the system to single-handedly perpetrate fraud. Such constraints can be easily expressed using an RBAC model through SoD constraints on roles, user-role assignments, and role-permission assignments. Furthermore, using constraints on the activation of user assigned roles, users can sign on with the least set of privileges required for any access. In case of inadvertent errors, such least privilege assignments can contain damage.

With the numerous extensions of RBAC like role hierarchies and SoD, it has become a rich and open-ended technology, which ranges from very simple at one extreme, to fairly complex and sophisticated at the other. Treating RBAC as a single model is therefore unrealistic. A single model would either include or exclude too much, and would only represent one point along a wide spectrum of technologies and choices.

The next sections describe the core RBAC model and the fundamental concepts of some widely known extensions.

### 1.2.2. The Core RBAC

Core RBAC embodies the essential aspects of RBAC. It captures the features of traditional group-based access control as implemented in most multi-user operating systems, and as such it is a widely deployed and familiar technology. The features required of Core RBAC are essential for any form of RBAC (discussed in a separate section). These features accommodate the traditional, simple but robust role based access control.

The basic concept of RBAC is that users are assigned to roles, access permissions are granted to roles rather than to single users, and users acquire permissions by being members of roles (see Figure 1: Core RBAC reference model). Core RBAC includes evident requirements that user-role and permission-role assignments can be multi-valued on both sides (many-to-many relation).

Thus a user can be assigned many roles and a single role can have more members. Similarly for permissions, the same permission can be associated with a number of roles and a single role can be assigned many permissions. This increases the manageability of the access control system, as permission grouping eliminates the overhead of modifying permissions for a given set of users one-by-one; therefore, security policy changes can be enforced in more structured, less time consuming manner.

Core RBAC includes requirements for user-role review whereby the roles assigned to a specific user can be determined as well as users assigned to a specific role. Further, it also includes administrative queries to review the permission mapping, that is, permissions granted to a specific role or user; these review functions enable simplified security policy compliance checking, making possible to proactively detect and correct erroneous configuration.

Core RBAC also includes the concept of user sessions, which allows selective activation and deactivation of roles. Finally, Core RBAC requires that users be able to simultaneously exercise permissions of multiple roles. It utilizes an access control decision mechanism that enables the superposition of permissions granted by a set of roles. This precludes limited implementations that restrict users to the activation of only one role at a time and allows for finer grained role definition reflecting the job functionalities or responsibilities the members of the roles have.

A detailed, formalized definition of both the Core RBAC Reference Model and Core RBAC Functional Specification is provided by section "Formalization of the Core RBAC" on page 17.

### 1.2.3. Standard extensions to the Core RBAC

Beside the Core RBAC components, the RBAC standard also provides a standardization for some widely used extensions to the base RBAC concept which provide additional functionality by extending the core model as well as the functional specification. This section outlines the fundamental concepts of some extensions. Formal definition and detailed description of the extensions can be found in the standard.

#### *1.2.3.1. Hierarchical RBAC*

Hierarchical RBAC enables support of role hierarchies. A hierarchy is mathematically a partial order defining an inheritance relation between roles, whereby descendant roles inherit the permissions of ancestors, and ancestor roles acquire the user membership of their descendants.

The RBAC standard uses the term 'seniority' instead of inheritance and defined senior and junior roles where senior roles are specialized roles that extend the junior ones. This terminology may be misleading in a world where object oriented concepts like inheritance are the predominant programming – and also thinking – concepts, so the definitions are adjusted to use the object oriented terminology. Of course the choice of terminology does not impact the mathematical strength of the model. It only provides better understandability.

Hierarchical RBAC addresses the following problem: As the same permission can be granted to distinct roles, roles can have overlapping access privileges. Users that do not have any common role might therefore possess the same permissions. It is customary in many organizations that general permissions - e.g. web or VPN access – are assigned to a large number of users. It would prove inefficient and administratively cumbersome to assign those general permissions to a large number of roles.

On the other hand, a low level of redundancy in role-permission assignment would mean that every user is potentially assigned to a large number of roles. For example, being a manager would automatically mean web, VPN, database access, access privilege to view or edit sensitive data and so on. All these privileges belong to the job functionality of a manager, and as such, they could be granted by being a member of a single role. Having a lot of roles with a small amount of permissions assigned to a single role would not be very informative in terms of organizational structure.

To improve efficiency and provide better support for organizational structure, this RBAC extension includes the concept of role hierarchies. The support of role hierarchies in the form of an arbitrary partial ordering – that means, supporting multiple inheritances among roles – is a desired extension to Core RBAC as it is providing better manageability by giving support for grouping and extending roles.

Multiple inheritance stands for the fact, that a role can have multiple ancestors. The permissions of all ancestors are thereby governed by the descendant role. A change to the general permissions, that means, to some of the ancestor role's permission assignment is thereby applied automatically and transparently to the descendants. Of course this largely improves the manageability and structure of the access control system.

Justification for requiring the transitive, reflexive, and antisymmetric properties of a partial order – in this case, these are the conditions of an ordinary support for multiple inheritance - has been extensively discussed in the mid '90s an there has been a strong consensus on the issue.

Even though, a large number of existing RBAC solutions only support single inheritance. In that case, any role may only have a single ancestor. These solutions are referred to as restricted or limited hierarchical RBAC.

#### *1.2.3.2. Static Separation of Duty*

Separation of duty relations are constraints used to enforce 'conflict of interest' policies. Conflict of interest in a role-based environment may arise as a result of a user gaining access permissions that are granted by conflicting roles.

An example for conflicting roles is the role of the requester and the approver of any critical business process. It is generally not allowed for anyone to be the approver for the process he/she has requested; therefore, these roles can be considered mutually exclusive.

One means of preventing this form of conflict of interest is static separation of duty. Static separation of duty enforces constraints - such as mutual exclusion of certain roles - on the user-to-role assignment.

Static separation of duty constraints consist of a role set and a number, and the policy is enforced by assuring that every user is assigned less roles from the role set, than the given number.

This concept enables the following kinds of policies:

- A user may not be assigned every role in a given role set.
- A user may not fulfill any two (or any given number) roles of a given role set.

From a policy point of view, static separation of duty provides a means of enforcing conflict of interest constraints by separating certain RBAC roles in a static manner. These static constraints are typically implemented to be enforced on administrative operations that have the potential to undermine separation policies by manipulating the user-to-role assignment. In fact, the operation `assingUser(user, role)` should be implemented to respect the separation constraints. Detailed description of the operation is provided by section "Administrative Functions" on page 21.

The enforcement of such a static constraint can easily be implemented in the absence of role hierarchies. Role hierarchies present a potential risk of inconsistencies with respect of duty relations and inheritance; therefore, implementing hierarchy-aware static separation of duty enforcement is a more challenging task. The standard defines the requirements both in the absence and presence of role hierarchies.

Although only policies regarding user-to-role assignment are discussed here, similar static constraints on operations or objects could also be useful in some environments. Formal RBAC models and policies that implement more advanced static separation of duty relations might exist, but these are not present in the standard RBAC specification.

#### *1.2.3.3. Dynamic Separation of Duty*

Dynamic separation of duty relations, similarly to static separation of duty relations, limit the permissions that are available to a user by placing constraints on the roles a user can fulfill. However, dynamic constraints differ from static ones by the context the limitations are imposed in.

Dynamic separation of duty limits the set of available permissions by placing restrictions into the role activation process. This way, the roles that can be activated within or during a user session are the target of the constraints.

As within static separation of duty, the constraints are defined as a set of roles and a natural number specifying, how many roles of the given set may be activated during a session.

Dynamic separation of duty utilizes the role activation feature of the Core RBAC. It extends the support for the principle of least privilege the Core RBAC already provides. Depending on the task a user performs, the session has different levels of access permissions at different times.

The dynamic approach guarantees, that permissions are only available when they are required by a task, and do not persist beyond execution of that task. This aspect of least privilege is also referred to as timely revocation of trust or permissions.

The key advantage of dynamic separation of duty over static separation is that while static separation addresses potential conflict of interest issues at administration time (the time a user is assigned a role), the dynamic approach allows conflicting roles to be assigned a user as long as they are acted on independently.

A simple example illustrates this advantage: Dynamic separation of duty allows a user to fulfill both the role of a requester and an approver, provided that the user acts on two independent processes or cases. He is allowed to be the requester in one business process while being the approver in another process. The separation happens on a per-session basis, that is, the two conflicting roles can not be activated during the same session. (However, a user could launch multiple parallel sessions and misuse a poorly constructed dynamic separation of duty system.)

Dynamic separation of duty generally allows greater efficiency and operational flexibility compared to static separation of duty, but also features performance drawbacks: Enforcement has to happen at runtime, which could degrade operational performance. As stated above, static separation of duty can be implemented by extending administrative functions. This could somewhat increase the execution time of administrative task, but does not effect operational functions.

# 2. Formalization of the Core RBAC

In computer science, formal methods refer to mathematically based techniques for the specification, development and verification of software and hardware systems. The approach is especially important in high-integrity systems, for example where safety or security is important, to help ensure that errors are not introduced into the development process. Formal methods are particularly effective in development at the requirements and specification levels, but can also be used for a completely formal development of an implementation [4].

The formal specification mathematically defines the behavior to be implemented, the requirements an implementation has to fulfill. Due to mathematical methods, a formal specification can precisely express the exact behavior of a system. Formal specifications also enable the creation of proofs of correctness. A specification in any natural language lacks the accuracy to fulfill these needs.

This section provides a formal definition for the Core RBAC, and as such, it might be less enjoyable than the rest of this writing. However, it is indispensable to be aware of the functions and data structures of the Core RBAC in order to fully appreciate the developed security model, design considerations and the implementation itself.

Beside resolving inconsistencies encountered in the RBAC specification proposal, this part also provides some implementation recommendations; however, much of the formalization effort is taken over from the RBAC specification as found in[3].

## *2.1. The Core RBAC Reference Model*

The Core RBAC reference model consists of sets of five basic data elements called users (USERS), roles (ROLES), objects (OBJS), operations (OPS), and permissions (PERMS). The model as a whole is fundamentally defined in terms of individual users being assigned to roles and permissions being granted to roles.

As such, a role is a means for naming and organizing many-to-many relationships among individual users and permissions. In addition, the Core RBAC model includes a set of sessions (SESSIONS) where each session is a mapping between a currently active user and the activated subset of roles that are assigned to the user.

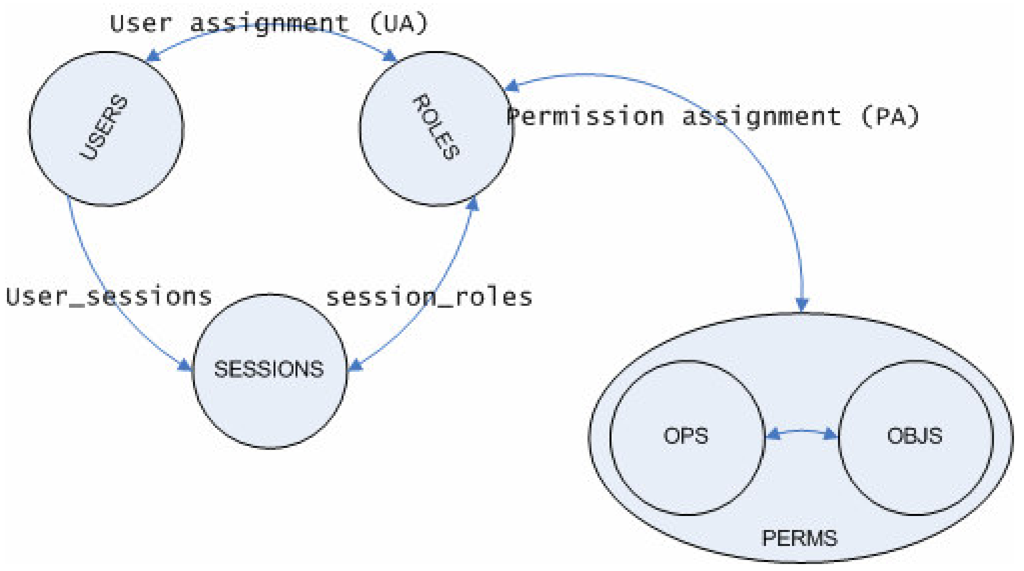


**Figure 1: Core RBAC reference model**

A user is traditionally defined as a human being, but the concept of a user can easily be extended to include machines, networks, or intelligent autonomous agents. A role is a job function in the organization. Within the context of the organization, the roles always have functional semantics associated, regarding the rights and responsibilities conferred on the user assigned to the specific role.

Permission is an approval to perform an operation on a protected object. Consistent with earlier models of access control a protected object is the representation of an entity that, in some manner, contains or receives information. For a system that implements RBAC, the protected objects can represent information containers or exhaustible or expensive resources.

Information containers could be - for example - files or directories in an operating system; columns, rows, tables, and views within a database management system or resource locators (URLs) or file system resources in a web environment. Expensive resources might feature workstations, terminals, printers or other hardware, CPU time, but also bank account or financial transactions.

The set of objects covered by RBAC includes all of the objects listed in the permissions that are assigned to roles.

Similarly, operations are one or more action that can be performed on the protected objects. Operations typically can be thought of as executable images of a program, which upon invocation execute some function on the protected object. For example, within a file system, operations could include read, write and execute; within a database management system, operations might include select, insert, delete, append, and update whereas within a web server serving static contents, operations could be limited to downloading of web resources[6].

[6] In a dynamic web application such as CGI, PHP processors or J2EE application servers, there might be a wide variety of possible operations.

As one can see, the types of operations and protected objects that RBAC controls are dependent on the type of system in which access control is to be implemented.

Without the convenience of separating operations and protected objects there is an enhanced danger that a user may be granted more access to resources than it is minimally needed because of limited control over the “type” or semantics of access permissions that can be associated with users and managed resources. For example, a user may need to list directories and modify existing files, without the permission of creating new files, or one may need to append records to a database table without modifying existing rows. Any increase in the flexibility of controlling access to resources also strengthens the application of the principle of least privilege.

Another premium feature of RBAC is the concept of role relations, around which a role is a semantic construct for formulating a policy. Figure 1 illustrates user assignment (UA) and permission assignment (PA) relations. The arrows indicate the many-to-many relationships already described. This arrangement provides great flexibility and granularity of grant of permissions to roles on the one side and assignment of users to roles on the other.

Any access request in a RBAC implementation is executed within the context of a user session. Each and every such session is a mapping of a single active user to potentially more than one role. Upon logon, a user establishes a session, during which the user activates a subset of roles that is assigned to him or her. Each session is associated with a single user, but a user can be associated with one or more sessions. In this way, multiple parallel sessions of the same user are allowed.

The permissions granted to the user are the superposition of permissions assigned to the roles that are activated across all the user’s sessions. The permissions are traditionally positive, that is, allowing permissions. Negative – or forbidding - permissions are not explicitly included in this model. The model leaves the possibility of incorporating negative permissions open to the RBAC implementation.

In the case of having only positive permissions, the superposition of permissions for the same protected object is the union of allowed operations on that object granted by the permissions.

Is the “read” operation on a certain protected object granted to a user by a permission assigned to one of the user’s roles and the “write” operation on the same object by another permission that is assigned to another active role of the user, the user will be allowed to both read and write the protected object in the same session. In some cases, this is not the desired behavior; this issue is addressed by ‘separation of duty’ concepts.

The co-existence of both positive and negative permissions may involve the development of custom evaluation logic, but the following method seems reasonable: The union of all negative permissions is subtracted form the union of all positive permissions. This way, no restriction can be circumvented. This method only serves demonstration purposes; it might not be suitable for all systems – not even any system.

The above is summarized in the following more compact, more formal definitions:

1. $UA \subseteq USERS \times ROLES$ User assignment is set of user-role pairs, which contains the valid user-to-role mapping.

2. $assigned\_users(r) = \{u \in USERS \mid (u, r) \in UA\}$ That is, a mapping of role r onto a set of users. The expression evaluates to the set of user assigned to a given role.

3. $PERMS \subseteq OPS \times OBJS$ Permission is an operation-permission pair that is in fact an approval to perform an operation upon an object. The official RBAC specification is somewhat inconsistent on this point, as its reference model specification defines the set of permissions as $PERMS = 2^{(OPS \times OBJS)}$, which would mean the set of permissions contains sets of operation-object pairs. Hence, a single permission could contain an arbitrary number of operation-object pairs. The more formal functional specification of RBAC, however, refers to a single permission as a single operation-permission pair. This document, as stated above, adopts the single-pair approach.

4. $PA \subseteq PERMS \times ROLES$ Permission assignment is a set of permission-role pairs that enables many-to-many permission-to-role mapping.

5. $assigned\_permissions(r) = \{p \in PERMS \mid (p, r) \in PA\}$ This expression evaluates to a set containing all the permissions assigned to a given role.

6. $user\_sessions(u) \subseteq SESSIONS$, $session\_user(u) \in USERS$, and $\bigcup_{v \in USERS, u \in USERS, u \neq v} user\_sessions(u) \cap user\_sessions(v) = \emptyset$ Each user can have multiple active sessions, and every session belongs to a single user; two different users can never have a common session.

7. $session\_roles(s) \subseteq \{r \in ROLES \mid (session\_user(s), r) \in UA\}$ The set of activated roles for a session is the subset of all roles assigned to the user that owns the session. This allows the creation of session with a limited subset of permissions for a user, and later, roles can be activated or deactivated on demand.

8. $session\_permissions(s) = \bigcup_{r \in session\_roles(s)} assigned\_permissions(r)$ The permissions granted to a user during a session – more precisely, the permissions of the session - is the union of permissions assigned to all roles associated with the session.

## 2.2. Core RBAC functional specification

The RBAC System and Administrative Functional Specification casts the abstract reference model concepts into functional requirements for administrative operations, runtime management, and administrative review. It outlines the semantics of the various functions that are required by the three parts the specification is divided into:

1. Administrative functions are responsible for creation and maintenance of the components of the RBAC model and their relations. They enable add, modify and delete functionality for managing the entities and relations of the model.

2. Supporting Systems Functions enable the underlying RBAC implementation to utilize the RBAC model constructs during user interaction. It includes runtime management required by RBAC such as session management and support for the authorization decision process.

3. Review Functions enable to view the RBAC entities and relations. They help administrators keep track of changes and effectively reconcile security policies. Besides listing various components the specification also includes optional functions that implement advanced debug or review queries, e.g. listing the allowed operation of a user for a given object.

### 2.2.1. Administrative Functions

Administrative functions enable the creation and maintenance of the RBAC elements USERS, ROLES and PERMS. It also defines functions for administering the main relations of the elements: UA and PA.

As already mentioned the types of operations and protected objects that RBAC controls are highly dependent on the type of environment in which access control is to be implemented. Therefore, management tasks related to creation of operations or protected objects are not part of the generic functional specification.

The RBAC element SESSIONS only has runtime semantics, so creation and modification of sessions is discussed as a part of the system functions. Also, any function related to sessions is not part of the administrative domain.

Short descriptions of the functions are provided in the following list:

- `addUser(u)` The function creates a new user. The function is only valid if the user does not already exist, that is, the user is not already a member of the set USERS. The newly created user does not have any sessions or roles assigned.

- `removeUser(u)` deletes a given existing user. Both the USERS and UA data sets are updated. It is left to the implementation whether active sessions of the user to delete should be forcefully terminated or not. It is acceptable, if the session – and the user – remains active until the session is terminates in a natural manner.

- `addRole(r)` This function created a new role. The role may not be an already existing role. The set ROLES is updated to include the newly created role. Initially, the new role does not have any users or permissions assigned.

- `removeRole(r)` deletes an existing role from the system. Again, it is an implementation decision how to proceed with sessions that are affected by the deletion of the role. The active sessions could be left intact or forcefully terminated. Upon deletion of a role, both ROLES and UA are updated so no query can return a non-existent role – with the possible exception of session queries, as noted above.

- `assignUser(u, r)` The function assigns a given user to a given role. All input data, has to be valid and the user may not already be assigned to the role. The function updates the user-to-role mapping (UA).

- `deassingUser(u, r)` This function removes the given assignment from the user-to-role mapping (UA). The call is valid if all input data is valid and the specified assignment exists. Similarly to other delete function, the handling of open sessions affected by the operation is left to the implementation.

- `grantPermission(obj, op, r)` This function grants a permission to perform a given operation on a given protected object to a role. It updates the permission assignment data set (PA). Again, all input data has to be valid, that is, the role is an element of ROLES and the operation-object pair is a valid permission. The specification leaves problem of open sessions open. However, a system where a user has to restart the session upon every modification might be inadequate in most of the cases.

- `revokePermission(obj, op, r)` The function revokes a valid permission already assigned to given (valid) role by updating the set PA. Again, the specification does not specify a way to handle open sessions, but leaving cached permissions of a session intact is a reasonable security hole, where marathon sessions could benefit from already revoked permissions, because the changes are not reflected immediately.

### 2.2.2. System Functions

Supporting system functions are responsible for the runtime features: session management and the access decision process. Role activation is another advanced runtime feature that allows the establishment of a session with a predefined subset of roles assigned to the session owner. During the session, roles can be activated or deactivated on demand.

These functions, due to their nature, are the most performance sensitive. A large amount of sessions may be established every day, and during every session, a potentially large number of access decisions are evaluated.

The following list provides a short description of system functions along with their formal specification.

- `createSession(u, rs, s)` The function creates a new session `s` for user `u` with an active role set `rs`. The first argument has to be a valid user. The second argument, the active role set has to be a subset of the roles assigned to the user. The third argument is a session identifier that has to be unique. It may not already be present in the set SESSIONS. The session id may be explicitly provided or implicitly generated by the underlying implementation. The formal specification of the function: $createSession(u, rs, s) \lhd$

$$\begin{aligned}
&u \in USERS;\\
&rs \subseteq \{r \in ROLES \mid (u,r) \in UA\};\\
&s \notin SESSIONS;\\
&SESSIONS' = SESSIONS \cup \{s\};\\
&user\_sessions(u)' = user\_sessions(u) \cup \{s\};\\
&session\_roles(s)' = rs;
\end{aligned}$$

  $\rhd$

- `deleteSession(u, s )` This function deletes the session of user `u` with the identifier `s`. The function call is only valid if both the user and the session identifier are valid elements. Additionally, the provided user has to be the owner of the specified session. Upon execution the session is terminated. $deleteSession(u, s) \lhd$

  Formally:

$$\begin{aligned}
&u \in USERS;\\
&s \in SESSIONS;\\
&s \in user\_sessions(u);\\
&SESSIONS' = SESSIONS - \{s\};\\
&user\_sessions(u)' = user\_sessions(u) - \{s\};\\
&session\_roles(s)' = \varnothing;
\end{aligned}$$

  $\rhd$

- `addActiveRole(u, s, r )` The function activates a given role for a specified session. The function is valid if and only if the input data is valid, the user is the owner of the specified session and the role is assigned to the user. Additionally, the specified role may not already be activated in the

session. Formally:
$addActiveRole(u,s,r) \lhd$

$$\begin{aligned}
&u \in USERS;\\
&s \in SESSIONS;\\
&r \in ROLES;\\
&s \in user\_sessions(u);\\
&(u,r) \in UA;\\
&s \notin session\_roles(s);\\
&sessions\_roles(s)' = session\_roles(s) \cup \{r\};
\end{aligned}$$

$\rhd$

- `dropActiveRole(u, s, r)` This function deactivates a given role for a given session. As usual, all input parameters have to be valid. The session has to be owned by the user and the role has to be an active role of the session. The following definition formally describes the function: $dropActiveRole(u,s,r) \lhd$

$$\begin{aligned}
&u \in USERS;\\
&s \in SESSIONS;\\
&r \in ROLES;\\
&s \in user\_sessions(u);\\
&(u,r) \in UA;\\
&s \in session\_roles(s);\\
&sessions\_roles(s)' = session\_roles(s) - \{r\};
\end{aligned}$$

$\rhd$

- `checkAccess(s, obj, op)` This function returns a Boolean value meaning whether the subject of the given session is authorized to perform the given action on the specified protected object. The function is valid, if the session is valid and also both `obj` and `op` are valid elements. The result is true if and only if at least one of the session's active roles grants the permission to perform the specified operation on the protected object.

Formally:

$$\begin{aligned}
&checkAccess(s,obj,op) = (\\
&\quad (s \in SESSIONS) \wedge\\
&\quad (obj \in OBJS) \wedge\\
&\quad (op \in OPS) \wedge\\
&\quad (s \in user\_sessions(u)) \wedge\\
&\quad (\exists r \in ROLES \mid r \in session\_roles(s) \wedge ((op,obj),r) \in PA)\\
&)
\end{aligned}$$

### 2.2.3. Review Functions

Review functions enable the administrators to perform queries on the system for review or reporting purposes. When the RBAC elements and relation instances have been created, it should be possible to view the contents of those relations. For example, from the UA relation, the administrator should have the facility to view all users assigned to a given role as well as to view all the roles assigned to a specified user. These features can be accomplished by a simple query performed on the underlying data set, in this case, the set UA.

In addition, it should also be possible to view the results of the supporting system functions to determine some session-related properties such as the active roles in a given session or the total permission domain for a given session. This requires manipulation of the non-persistent data object such as the sessions or the active role set for a given session.

Further, there is a need to view the resulting policies in a more advanced way, by correlating data structures and following multiple relations. For example, listing all permissions granted to a user of listing all users, whom a specific access level is granted on a given protected object. Advanced queries like that typically require more calculation and are therefore more performance intensive, but they are indispensable for easy policy compliance checking and advanced report generation. Tasks as report generation and compliance checking are predominantly off-line task, and as such, they are less performance critical functions.

There is a wide spectrum of review functions with fundamental – and therefore mandatory - and rather optional queries, which have been designated as optional/advanced function in the standard RBAC specification.

Short descriptions of the functions are provided in the following list:

- `assignedUsers(r)` This mandatory functions returns the set of all users associated with a given role. The input role has to be a valid role. A trivial implementation: $assignedUsers(r) = \{u \in USERS \mid (u,r) \in UA\}$.

- `assignedRoles(u)` This mandatory functions searches the UA set from the user point of view and returns the set of all roles available to a given user. The input user has to be a valid user, an element of the set USERS. It could be implemented analogous to the previous function: $assignedRoles(u) = \{r \in ROLES \mid (u,r) \in UA\}$.

- `rolePermissions(r)` This review function returns all permissions granted to a given role. $rolePermissions(r) = \{(op,obj) \mid ((op,obj),r) \in PA\}$ A query that returns all the roles, which grant a given permission, could be defined in a very similar manner.

- `userPermissions(u)` This advanced review function returns a set of permissions granted to a given user by calculation the union of all permissions granted by all the roles that the given user is assigned. $userPermissions(u) = \{(op,obj) \mid ((op,obj),r) \in PA \wedge (u,r) \in UA\}$ Again, a very similar query can be defined to return all users whom a special permission is granted.

- `roleOperationsOnObject(r,obj)` This function returns the set of operations a given role is allowed to perform on a given object. $roleOperationsOnObject(r,obj) = \{op \mid ((op,obj),r) \in PA\}$

- `userOperationsOnObject(u,obj)` Similarly to the above function, this review query returns the set of operations allowed to perform on a given object. In this case, however, the search criterion is a user rather than a role. For that, all roles the user is allowed to act as have be queried. $userOperationsOnObject(u,obj) = \{op \mid ((op,obj),r) \in PA \wedge (u,r) \in UA\}$

- `sessionRoles(s)` This debug function simply wraps the underlying data structure (`session_roles(s)`) and returns the active roles for a given session to accomplish administrative or debug purposes.

- `sessionPermissions(s)` This debug function is very similar to the `userPermissions(u)` function, but queries the non-persistent `session_roles(s)` data set instead of the UA set. It returns the union of all permissions granted to the roles that are activated in the specified session. All the prerequisite constraints of the underlying data structures have to be met. $sessionPermissions(u) = \{(op,obj) \mid ((op,obj),r) \in PA \wedge r \in session\_roles(s)\}$

Of course there is an infinite set of review functions that can be generated for the RBAC core model; this listing only contains the predominant – and most useful - ones.

# 3. Tivoli Access Manager security model versus RBAC

Tivoli Access Manager is a policy-based access control solution for e-business and enterprise applications. It can be thought of as a collected suite of security management services with a variety of distributed policy enforcing modules and plug-ins for the infrastructure components of enterprise applications. On the other hand it is a unified platform that effectively controls and enforces security policies among heterogeneous application of the large scale enterprise in a consistent and manageable way.

This section only outlines the RBAC concepts implemented in the Tivoli Access Manager; it will either discuss general features, nor non-RBAC-related concepts of it. One of the mostly desired features in enterprise environments, namely Single Sign On, is not mentioned at all; a meticulous discussion of the differences between the NIST Core RBAC and Tivoli Access Manager's RBAC adoption is provided instead.

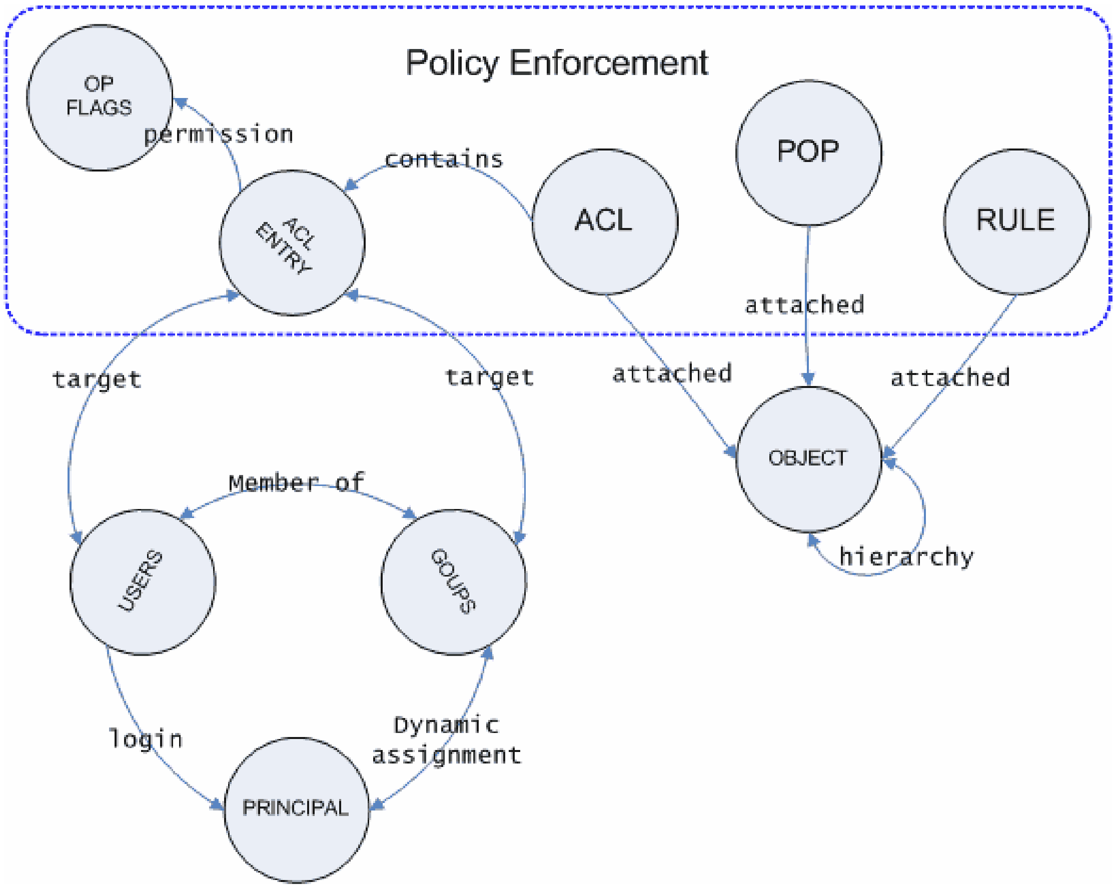


**Figure 2: The Tivoli Access Manager RBAC model**

Figure 4 illustrates the role-based security model implemented in Tivoli Access Manager. The main features and behaviour of the single components are described below in detail.

## 3.1. Roles and Principals

RBAC Roles are implemented as Tivoli Access Manager Groups; the user assignment (UA) data structure is therefore embodied in the group memberships within Tivoli Access Manager. Users and groups as implemented in Tivoli Access Manager are very close to the Core RBAC recommendation; however, there is one key difference: In Tivoli Access Manager, permissions can be assigned to both users and groups. This difference interferes with the permission grouping concept defined in the RBAC specification where permissions can only be granted to roles, not individual users (see Figure 1: Core RBAC reference model). Granting permissions to single users could be a means for handling a small number of exceptions to the global security policy, but doing so is not considered a good practice.

The principal represents an authenticated – that is, a logged in - user in a Tivoli Access Manager environment. This entity is the result of the authentication process and therefore, it can be thought of as the Core RBAC session entity rather than the RBAC user itself. Moreover, as with Core RBAC sessions, every single logon process results a new principal object, which can be acted on independently.

Tivoli Access Manager does not provide out-of-the-box support for dynamic role activation as described in section “System Functions” on page 22. The principal object is initialized to contain all group memberships the according user is assigned, and the set of groups does not change during the session. However, there is a means to programmatically update the group membership at runtime, without affecting parallel sessions of the same user. Effects of updating the in-memory user credential do not persist beyond the session, that is, a principal’s owner can be dynamically added to groups[7] without modifying the central group membership data [6]. On-the-fly role activation can be achieved in a Tivoli Access Manager environment by utilizing the provided authorization API and updating the in-memory credential on demand.

## 3.2. Protected Object Space

The protected object space is a virtual, hierarchical representation of the protected resources. The Tivoli Access Manager protected objects are the according counterpart of the Core RBAC protected objects. A major difference is the support of protected object grouping through the means of protected object hierarchies. Within Tivoli Access Manager, there are basically two types of protected objects: Container objects, which can “hold” or group other protected objects; and leaf objects, which virtually represent the physical or logical protected resources.

[7] In fact, the according API functions create a copy of the original credential that is extended to include the desired groups. This copy can then be used to issue authorization requests with the updated authorization data.

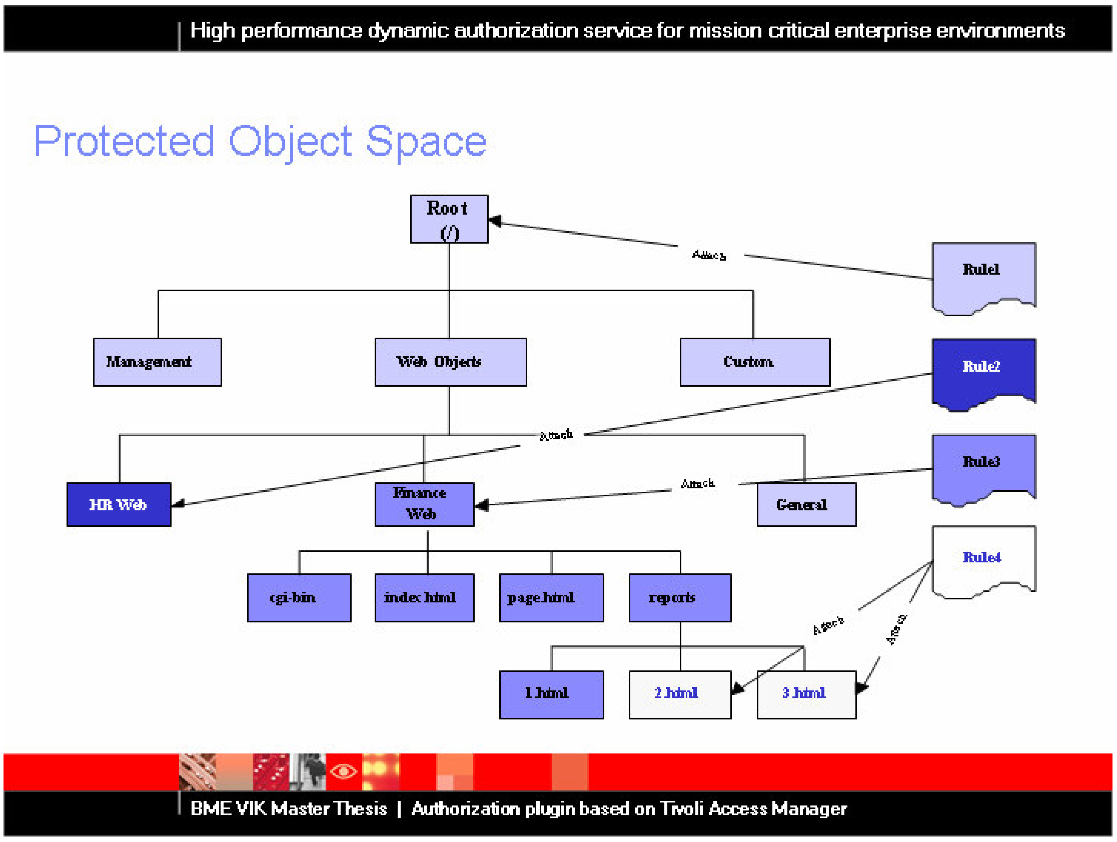


**Figure 3: Protected object space**

Container objects allow all protected objects to be kept organized in a tree structure[8]. The single objects can then be referenced in a way similar to the absolute path of a file within a file system. This feature enables the object space to represent web resources or file system content in a most straightforward manner, but other environments can also take advantage of the organized object structure in terms of manageability, especially when a large amount of protected objects have to be dealt with.

Besides improving manageability, the hierarchical structure of the protected objects offers another key advantage, namely inheritance. Inheritance, in this case, refers to the way a security policy - permission - is applied to the protected objects.

A security policy can be explicitly applied to a protected object or implicitly inherited from objects located above it in the hierarchy. Administrators need to apply an explicit security policy in the protected object space only at the points in the hierarchy where the access permissions must change. Adopting an inherited security scheme can greatly reduce the administration tasks for any environment.

The power of permission inheritance is based on the following concept: Any protected object without an explicitly attached security policy inherits the policy of the nearest container object above it that has an explicitly set security policy. The inheritance chain is broken whenever a protected object has an explicitly attached security policy.

[8] To be more precise, the structure is rather a forest than a tree. A forest is a set of trees. A tree can only have one root, whereas a forest can consist of multiple roots, each holding a single tree.

Security policy inheritance simplifies the task of setting and maintaining access control constraints on a large protected object space. In a typical web-based environment, the security administrator only needs to attach a few security policies at key locations to secure the entire object space (see Figure 3). Therefore, it is called a sparse security policy model.

## 3.3. Policy Enforcement

The Tivoli Access Manager Authorization Service is responsible for evaluating authorization requests. It calculates authorization decisions based on the security policies applied to the requested object. As mentioned before, security policies implement access permissions of the Core RBAC model. The key difference is, however, that Tivoli Access Manager provides three completely different types of security policies, whereas Core RBAC only defines one kind of permission, namely the right to perform a given action on a given object (see Figure 1: Core RBAC reference model).

The three types of policies provided by Tivoli Access Manager are:

1. Access Control Lists (ACLs), which provide the standard, static permission capabilities as defined in the Core RBAC Model

2. Protected Object Policies (POPs) that allow more dynamic behavior, based on object or environment properties

3. Authorization Rules, which enable the definition of flexible, fully dynamic text based access rules

For each request for access to an object inside the protected object space the request will be evaluated against the Access Control List, Protected Object Policy and the Authorization rule attached to the object explicitly or inherited by it. A single object can have none to all three types of security policies attached to it but only one of each type.

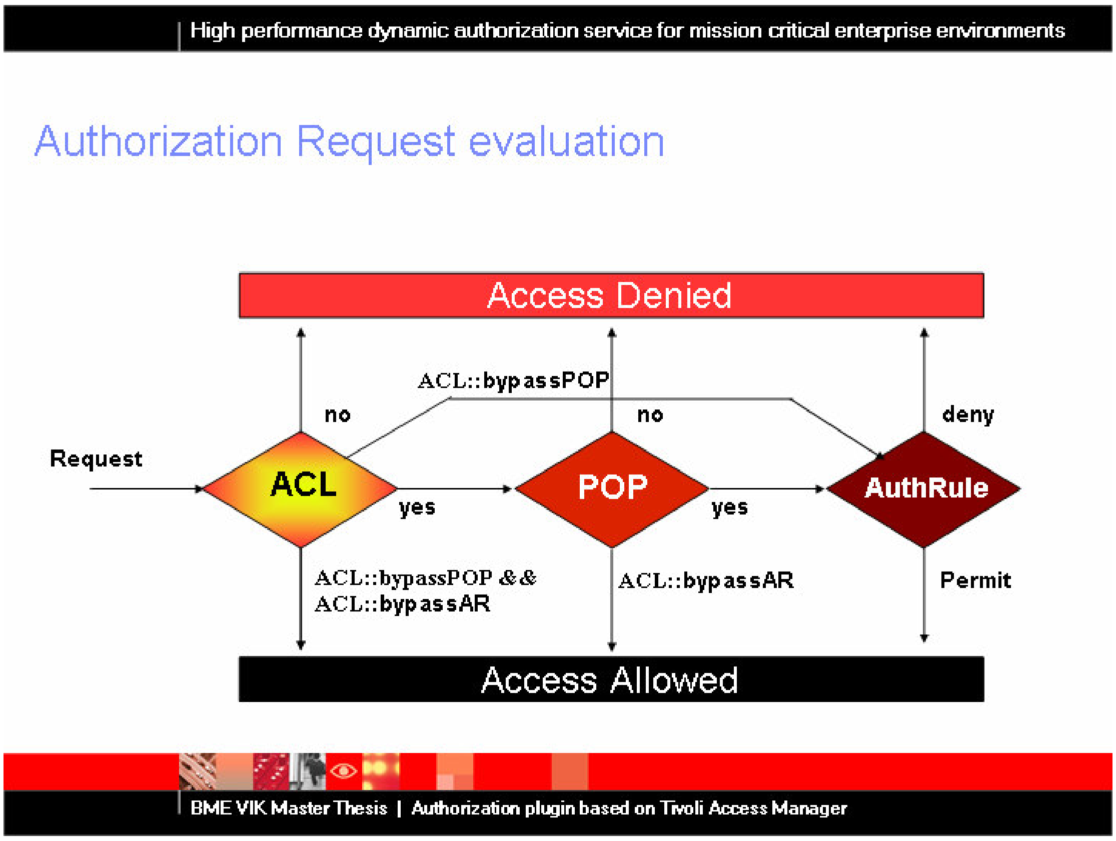

High performance dynamic authorization service for mission critical enterprise environments
Authorization Request evaluation
Access Denied
Request
ACL
no
yes
ACL::bypassPOP
POP
no
yes
AuthRule
deny
Permit
ACL::bypassPOP &&
ACL::bypassAR
ACL::bypassAR
Access Allowed
BME VIK Master Thesis | Authorization plugin based on Tivoli Access Manager

**Figure 4: Authorization request evaluation**

The authorization control decision process is featured in Figure 4. Tivoli Access Manager implements lazy evaluation, that is, the first denial terminates the decision process. For Example, if access is rejected by an Access Control List, no Protected Object Policy or Authorization Rule is evaluated. A denial is immediately returned to the system and the access control decision process is terminated.

A positive – allowing – result of the Access Control List triggers the execution of any attached Protected Object Policy; if the Protected Object Policy also returns a positive result, any attached Authorization Rule is evaluated. From a declarative point-of-view, the authorization decision result can be thought of as the logical product (logical AND) of all three policy instances. The lazy evaluation of the logical product provides serious performance benefits, as Protected Object Policies and Authorization Rules take much more time to evaluate than the fully static Access Control Lists.

As Figure 4 shows, Tivoli Access Manager also provides a means of bypassing Protected Object Policies or Authorization Rules, or even both. This is achieved by special Access Control List flags; more detail on these flags can be found in [7].

### 3.3.1. Access Control Lists

The concept of Access Control Lists is very close to the Core RBAC permission definition. The Access Control List can be attached to a protected object and specifies a predefined set of actions a set of users and groups can perform on the object.

The Access Control List consists of an arbitrary number of so-called ACL Entries. An ACL Entry is divided into two parts:

- The target defines which users are affected by the ACL Entry. The target can be a single user or a group. Tivoli Access Manager implicitly defines two more categories: an entry can also be applied to affect any authenticated users that are not governed by other ACL Entries of the same Access Control List and an ACL Entry to handle unauthenticated users is also provided.

- The ACL Entry permissions define, which actions – operations - are allowed for the entry's target on the object the ACL is attached to. Actions correspond to RBAC operations and are environment specific.

The schematic description of a sample ACL could be:

| | |
|---|---|
| User Jane | append, view, delete, modify |
| User Bob | append, view, delete |
| Group students | append, view |
| Any other authenticated user | view |
| Unauthenticated | none |

#### *3.3.1.1. Action groups*

It is possible, that the same protected resource can be accessed through multiple services, where each service provides a predefined set of possible operations. An evident example for the above is a file that can be accessed as a web resource through a web server but also as a local file from an operating system account.

When managing permissions for multiple environments, grouping of actions seems to be a reasonable need. Within Tivoli Access Manager, administrators can define multiple action groups for organizing their many actions. For example, one action group could include all actions that can be applied to web-based environments while another action group could collect file system related actions.

Tivoli Access Manager supports the creation of up to 32 action groups, where each action group can contain a maximum of 32 actions [7]. Each action group has a unique name, and each action within an action group is identified by a single character. The action group name and action name together identify a single flag in the ACL Entry's permission portion. The flag can take two values: TRUE means that the given action is authorized, FALSE means that the action is not allowed.

A more precise description of the structure of an ACL is shown by the following example:

The actions 'modify' and 'view' represent actions that can be invokes on a web resource, hence, they are contained in a separate action group 'webOPS'. The actions 'append' and 'delete' are operations, which are defined on the same object, but these operation are file system operations, so they are put into a separate action group 'fileOPS'. The ACL of the previous example could be defined as follows:

| | |
|---|---|
| User Jane | `[webOPS]mv[fileOPS]ad` |
| User Bob | `[webOPS]v[fileOPS]ad` |
| Group students | `[webOPS]v[fileOPS]a` |
| Any other authenticated user | `[webOPS]v` |
| Unauthenticated | `<none>` |

### 3.3.2. Protected Object Policies

Access Control Lists provide a static mapping between protected objects and allowed operations for a set of users. They simply provide a yes or no answer to a user request to perform a given operation on a given resource. Unlike ACLs, Protected Object Policies enable policy checking against a predefined set of dynamic conditions.

Protected Object Policies are policy instances that can be attached to protected objects, but unlike Access Control List, they affect all users in the same fashion, without regard to their group membership. The Protected Object Policy specifies additional conditions governing access to the protected object, such as privacy, integrity, auditing, and time-of-day access[9] and so increases the flexibility of security policies.

The additional conditions provided by the Protected Object Policies are basically environment specific properties – like network-address-based access control in a web environment - but also feature general security policy rules. These general object policies include access control based on a time-of-day constraint as well as increased logging capabilities on a per-object basis.

Protected Object Policies enable dynamic access control but are limited to utilize a predefined set of context and environmental conditions. Including a Protected Object Policy into the authorization decision requires runtime evaluation of every defined condition, however, it does not impact the performance of the authorization service as these conditions can be simply calculated from data already available to the authorization system, such as system time or network address. A construct similar to Protected Object Policies is not defined by the Core RBAC model.

### 3.3.3. Authorization Rules

The Authorization Rule is a policy type that provides full flexibility by enabling text-based specification of access control rules. The rule is stored as a text resource within a rule policy object and is attached to a protected object in the same way and with similar constraints as Access Control Lists and Protected Object Policies.

[9] A policy that takes access control decisions based on time constraints. For example, users are only allowed to access a resource on working days between 8am and 6pm.

Authorization Rules allow making access control decisions based on attributes of the requesting principal or the requested protected object attributes as well as attributes originating from the context and environment surrounding the access decision.

For example, rule enable implementing a time-of-day policy that depends on the user or groups the requesting user is member of. Rules can also be used to extend the access control capabilities that Access Control Lists provide by implementing a more advanced policy, such as one based on quotas. While an ACL is only able to grant a permission to write to a resource based on constraints on the requesting user, an Authorization Rule can go further by allowing to determine if a group has exceeded a given quota for a the current week and take an authorization decision accordingly.

Rules are defined as XSL (extensible stylesheet language) transformations to allow maximal flexibility. XSL Transformations are predominantly used to transform XML documents through pattern-matching capabilities, built in text manipulation functions and support for flow control statements like conditional branching and loop control. XSL possesses an inherent ability to analyze and evaluate XML data, which is becoming the standard for data representation in e-business environments. XSL is built on other XML-based standards such as XPath, which is the expression language at the core of an Authorization Rule.

Within a Tivoli Access Manager Authorization Rule, an XSL transformation can reference context and environmental attributes, attributes of the requesting principal and protected object as well as the identifier of the requested operation itself. The textual representation of the attributes can be manipulated and compared by the transformation in an arbitrary way. The only constraint is that the transformation should result in a single line of text representing the access decision in a predefined format.

The following example shows an Authorization Rule that takes the access decision based on the requesting principal's properties such as user identifier, group membership or LDAP distinguished name as well as requested action and a custom environmental attribute.

```
<xsl:choose>

  <!-- Explicitly grant access to user named 'username'  -->

  <xsl:when test="azn_cred_principal_name = 'username'">
    !TRUE!
  </xsl:when>

  <!-- Explicitly deny if the requesting user is 'guest' -->

  <xsl:when test="azn_cred_principal_name = 'guest'">
    !FALSE!
  </xsl:when>

  <!-- Explicitely allow admin by providing LDAP DN      -->

  <xsl:when test="azn_cred_registry_id =
'cn=sec_master,secAuthority=Default'">
    !TRUE!
  </xsl:when>

  <!-- Users who are member of QuotaPrinterGroup but not -->
  <!-- member of NoQuotaGroup may perform action p. They -->
  <!-- may perform action q only if their printQuota is  -->
  <!-- less than 20. -->

  <xsl:when test="azn_cred_groups = 'QuotaPrinterGroup'
    and not (azn_cred_groups = 'NoQuotaGroup')">

    <xsl:if test='contains(azn_engine_requested_actions,"p")
      or contains(azn_engine_requested_actions,"q")
      and printQuota &lt;20'>
      !TRUE!
    </xsl:if>

  </xsl:when>

  <xsl:otherwise>
    !FALSE!
  <xsl:otherwise>

</xsl:choose>
```

The sample rule distinguishes five cases. Upon execution, the user's name is tested against a predefined value. Access is immediately granted upon match, however, if no match is detected, control is passed on to the next case that explicitly denies access for a given user. The third case checks the user's LDAP distinguished name to match a specific value.

The next case is more complex. It provides more flexibility but accordingly, it takes longer to execute. This rule takes the principal's group membership into account, but also involves the requested actions and the environmental variable `printQuota` into the authorization process.

Authorization Rules provide premium flexibility, however, involving them into frequently invoked access control decisions raises serious performance concerns as the transformation has to be processed and executed upon every single request. The performance need of a transformation depends on the XSL code as well as the amount of attributes referenced and the manner pattern matching operations are used.

# 4. Tivoli Access Manager Architecture

The Tivoli Access Manager infrastructure has been designed to provide a robust yet flexible authorization and authentication solution for enterprise and e-business environment. The basic architecture presents a scalable, general purpose security framework rather than a system specialized to target a predefined type of environments, hence, the basic design can be adopted to suit the needs of a number of different types of environments.

The abstract model provided by the Tivoli Access Manager base system is not limited to be used within a given type of environments; the customization and specialization of the actual solution is achieved by the appropriate choice and configuration of further Tivoli Access Manager components, which have been developed for specific environments, such as policy enforcers for SOA compliant systems, web-based e-business environments or operating systems. These specialized components are tailored to support environment specific needs; however, all these modules utilize the same robust Tivoli Access Manager base system.

The Tivoli Access Manager environment is designed to benefit from hardware or software redundancy to provide both high availability capabilities and increased performance. Key components can be replicated to provide a robust load balanced authorization and authentication solution.

This section describes the main components of Tivoli Access Manager. Beside a short technical overview based on [7], the section also introduces possible extension points and uncovers some undocumented details of the authorization service.

## *4.1. Main Components*

Every Tivoli Access Manager system is built around a policy server that enables the creation and maintenance of the abstract security model entities and relations (see Figure 2: The Tivoli Access Manager RBAC model) and one or more resource managers, which enforce the security policies in the target systems.

The Tivoli Access Manager environment includes two main data sources used by the deployed resource managers. The first data source - the centralized user registry - stores the user and group definitions and authentication data. It is used for managing and identifying users in the Tivoli Access Manager environment. The Tivoli Access Manager environment utilizes an LDAP directory server as the user registry.

The second data source is the Authorization Database maintained by the Tivoli Access Manager Policy Server. It stores and provides the security policy enforcement information for every deployed resource manager to enforce security. Resource managers access these two data sources over network channels secured by Secure Socket Layers (SSL).

### 4.1.1. User Registry

The user registry is a central repository, which provides a persistent storage for user identities, authentication data and group definitions as well as the user-group mapping. Further, it serves as the storage of metadata required for additional functionality[10].

The user registry is provided by an LDAP directory server, hence the data is stored in the form of hierarchically organized objects in a tree structure called the Directory Information Tree (DIT). Directory servers are optimized for premium query speed and are typically read more often than updated.

The LDAP distributed architecture supports scalable directory services with server replication capabilities. Server replication improves both the availability and performance of a directory service. The replication is based on a redundant master-subordinate model where the data of the master instance is replicated to several read-only replicas, which the query load is balanced among.

The combination of a master server and multiple replicated servers helps to ensure that directory data is always available when needed. If any server fails, the directory service continues to be available from another replicated server. Tivoli Access Manager supports this replication capability.

### 4.1.2. Policy Server

The Tivoli Access Manager Policy Server maintains the Master Authorization Database, the authentic source of authorization information within the Tivoli Access Manager environment. This component is responsible for administration tasks as the creation and modification of policies and protected objects.

There can only be a single Policy Server within a Tivoli Access Manager environment[11]. This would render a single point of failure only for administrative tasks, as most resource managers can be configured to cache authorization information. With caching enabled, failure of the policy server does not impact the authorization service; it merely disables any modification to the existing security policy. This means for example, that administrators will not be able to make any change to user's permissions, but the users will still be able to log in to the protected applications and carry on with their tasks.

High availability of the administrative functions, if desired, can be achieved by configuring a standby Policy Server. Within this architecture, both the primary and the standby Policy Servers are running providing a hot standby capability, however, only one Policy Server is communicated with by the other Tivoli Access Manager

[10] For example Single Sign On resources defining user and authentication data mapping for the back end applications. This fundamental concept is not discussed further as it is beyond the scope of this document.

[11] To be precise, there can be only one Policy Server for each secure domain. Large scale enterprise environments can be divided into more secure domains, where each domain may have an own object space and user registry.

components. Upon failure of the primary Policy Server, the standby Policy Server takes over the task and becomes visible to the other components of the infrastructure.

#### *4.1.2.1. Structure of the Authorization Database*

The Tivoli Access Manager Authorization Database is a special persistent storage separate from the user registry. It uses a non-standard binary format similar to a generic relational database but specially tailored to store policy data and offer high performance data source to both the authorization and administration services.

The Authorization Database stores the following elements:

- Protected object space, the virtual, hierarchical representation of the protected resources along with their short description.

- The defined action groups and the actions themselves. The actions consist of a single character identifier and a short description. Additionally, an integral value is also provided. This number defines the position – index - of the action in a 32 element array that represents a single action group. This way any permission can be stored as a 4-bytes array for each action group.

- Access Control Lists and Access Control List entries. A textual definition can be attached to the Access Control Lists for maintenance purposes. The permission portion of the Access Control List entries is stored in hexadecimal format. The textual permission format, as shown in Section “Action groups” on page 32 can be calculated by involving the action group and action definitions.

- Protected Object Policies. The policies contain resource manager specific conditions and a textual description.

- Authorization Rules, which consist of an identifying name, a textual description and the XSL transformation as the rule text.

- Mappings between the elements. The mappings define, which policy (Access Control List, Protected Object Policy or Authorization Rule) is attached to which protected objects.

In a file system, Access Control Lists or other permission types are typically implemented as a fixed size binary data in the metadata portion of the file. Setting the same permission for multiple files results the same permission data – the actual binary flags – to be copied to the metadata portion of all the selected files. Storing the same data at multiple locations on the disk decreases the effective utilization and introduces additional manageability issues: A change of policy – for example revoking access privileges of a specific user - can only be enforced by modifying the access privilege for each affected file one-by-one.

In the Authorization Database, Access Control Lists, Protected Object Policies and Authorization Rules are not copied to the protected object they are attached to. These objects are named instances: every one of them is uniquely identified by a textual name. Any protected object, which the given policy is attached to, only stores the identifier of the policy, not a value copy thereof. Hence, changing the permissions granted by an Access Control List that is attached to many protected objects only involved updating a single element. This concept provides premium performance benefits regarding permission modification operations and also keeps a typical Authorization Database much smaller in size.

### 4.1.3. Resource Manager

Within Tivoli Access Manager, the Resource Manager components are responsible for the actual enforcement of the security policies. The Policy server only propagates abstract authorization data (see Figure 2: The Tivoli Access Manager RBAC model) into the Authorization Database, but provides no means to put these access rules in force within the protected applications and systems.

Tivoli Access Manager includes several Resource Manages that build upon the same core infrastructure but each of Resource Manager is specialized to provide access control for a given application.

- Plug-in modules for several web servers enable fine-grained control of the published web content by intercepting and filtering incoming requests based on a centralized security policy. This involves the installation of the plug-in into the target web server. Another component enables the fully transparent control of multiple back-end web servers: this Resource Manager is a high performance, multi threaded reverse proxy that sits in front of the back-end web servers and transparently forwards the HTTP requests according to the access control rules.

- Centralized access control can also be implemented for UNIX-like environments. In the case of UNIX derivatives, the Resource Manager consists of a kernel module that intercepts the predefined operations, and further user-space code that assist in retrieving centralized access control information. Tivoli Access Manager for Operation Systems provides control over the following resources:

    - File system resources.
    - Remote and local network services
    - Login and password management services
    - Services for changing user and group identities, like the ‘su’ and ‘sudo’ commands.
    - Additionally, the Resource Manager enables extensive control over the ‘root’ user account. This account traditionally has unlimited access to system resources in any UNIX-like environment and raises serious security concerns.

- WebSphere Application Servers can utilize the according Access Manager plug-in as a security provider for the deployed web application.

- Another Resource Manager is specially tailored to protect MessageQueue based applications.

### *4.1.3.1. Processing and authorization of a request*

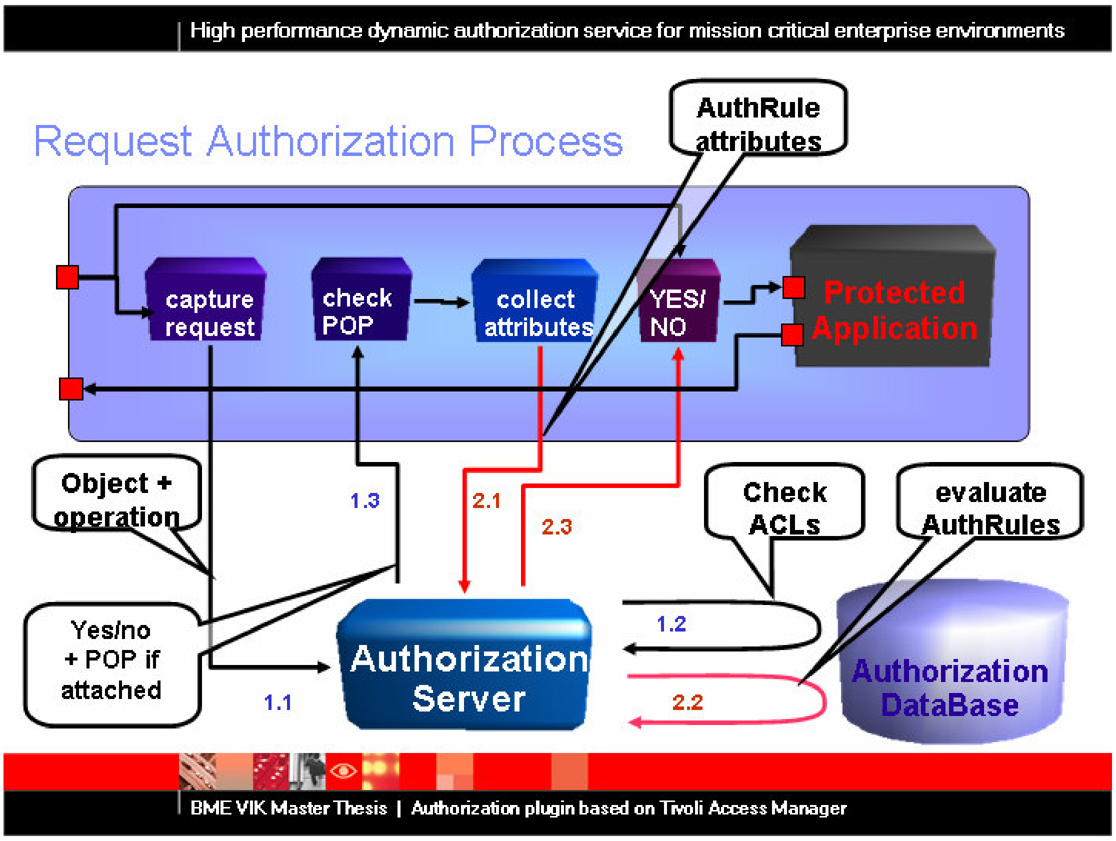


**Figure 5: Request Authorization in Resource Managers**

The many Resource Managers all share a common concept of filtering and approving resource access in the target systems. Figure 5 introduces a schematic overview of the process, the detailed description is provided below.

1. The Resource Manager wraps the secured application and blocks the incoming requests. It then identifies the requested operation and maps it to a predefined Access Manager action. (Actions are the Access Manager counterpart of the RBAC operations, as already mentioned in Section “Access Control Lists” on page 31.)

2. Then, the Authorization Database is queried. The Resource Manager sends the requested object’s name along with the operation to the decision taking component. The request is checked against the Access Control Lists stored in the Authorization Database and the authorization decision is passed back to the Resource Manager.

   a. In case of a positive decision, any Protected Object Policy attached to the requested object is passed back to the Resource Manager along with the "yes" answer. This step is indispensable, since the Protected Object Policy potentially contains environment specific conditions – such as client IP constraints – that the authorization service is unable to check[7].

   b. If access is denied by the Access Control Lists, only the "no" answer is sent back to the Resource Manager since there is no need to evaluate any Protected Object Policy. In this case, the authorization process is immediately terminated. Access is denied.

3. The Resource Manager checks the returned Protected Object Policy. It contains constraints on data already available to the Resource Manager such as a client IP or local time.

   a. Access may be denied by the Protected Object Policy. In this case, the authorization process stops and access is denied.

   b. If access is allowed, the attached Authorization Rule is to be evaluated (if any). The Resource Manager collects the variables that can be referenced from within the XSL transformation and send them to the authorization service.

4. The authorization service receives the attributes provided by the Resource Manager, loads the transformation from the Authorization Database and invokes it with the provided variables. The authorization service then returns the final yes/no answer to the Resource Manager.

   a. Access is granted by the Authorization Rules as well. The Resource Manager forwards the blocked user request to the protected application and returns the response to the user.

   b. A negative answer form the rule evaluator, as usual, denies access to the requested resource.

#### *4.1.3.2. Authorization data caching*

The authorization of a user request involves runtime calculations and communication between multiple components. As the components are typically distributed among more hosts, communication potentially happens over network. In addition to the limited network communication speed, the centralized runtime evaluation could easily become a bottleneck in the authorization process degrading overall system performance.

To circumvent the above performance issues and provide a scalable architecture, Tivoli Access Manager provides mechanisms that enable caching and replication of the Authorization Database.

Although Resource Managers rely on the information stored in the centrally maintained Authorization Database, there is no need to query the central component upon each authorization request.

The information required to make access decisions can be replicated and cached to enable authorization policy to continue to be enforced even if the Tivoli Access Manager Policy Server or the User Registry server becomes inaccessible.

The Tivoli Access Manager Authorization Server is an optional component specially tailored to offload authorization decisions from the Master Authorization Database maintained by the Policy Server to provide for higher availability and increased performance of authorization functions. Each deployed Authorization Server maintains its own replica of the Authorization Database, and provides the functionality to evaluate access controls based on data stored in the replica. (As described in section "Processing and authorization of a request" on page 41, the Authorization Server is capable of evaluating Access Control Lists and executing dynamic Authorization Rules by invoking XSL transformation.)

The Policy Server serves all administration requests and maintains an authentic source of the access decision data. Authorization Servers do not accept administrative requests; they merely provide authorization functionality. Upon policy modification, the Policy Server pushes the updates to all the replicas.

The authorization service along with a database replica can also be embedded directly into a Resource Manager. In this case, the functions of an Authorization Server are contained in the Resource Manager itself. This operation mode is referred to as local cache mode, whereas a standalone Authorization Server would allow multiple Resource Managers to share a single replica, that is, to operate in remote cache mode.

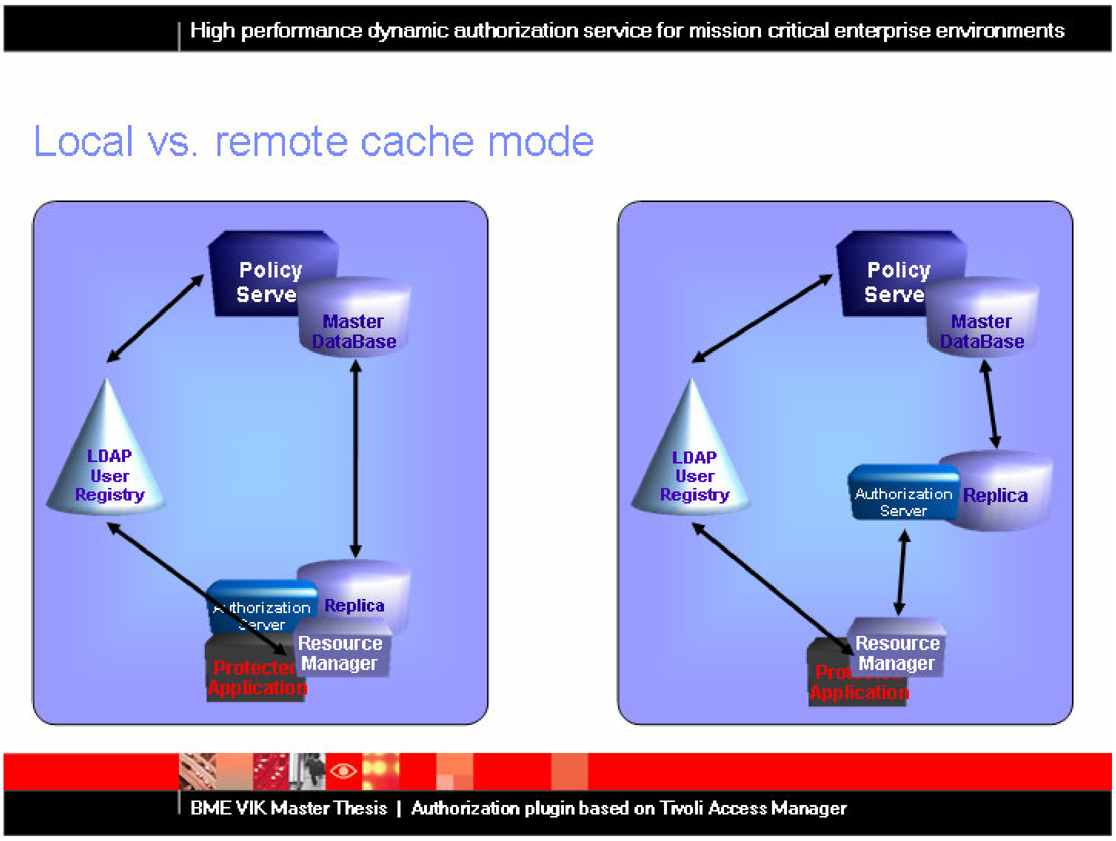


**Figure 6: Local versus remote cache mode**

Both operating modes decrease the load of the single Policy Servers, however, with a large number of replicas typically caused by many Resource Managers running in local cache mode, the load introduced by the synchronization of the many replicas can be significant. Remote cache mode enables cache sharing among multiple resource managers without increasing synchronization overhead.

On the other hand, local cache mode fully eliminates network communication during the authorization process as each involved component is replicated to the affected host. This mode provides better performance in comparison with remote cache mode.

The two cache modes also have different effects on manageability. The duration of the synchronization process – the process when the Policy Server pushes updates to the Authorization Server replicas – determines the time until a modification takes effect. With many replicas, a change to the policy database will take longer to become visible to components that use the replicas.

## 4.2. Extension Points

Tivoli Access Managers provides a complex security solution flexible enough to handle a wide spectrum of security policies. Beside the several out-of-the-box components Tivoli Access Manager also embodies application programming interfaces and extension points to enable communication with, and extension of existing components as well as the creation of new, custom components that utilize the Tivoli Access Manager core infrastructure.

### 4.2.1. Authorization Decision Information retrieval

Dynamic Authorization Rules enable virtually any operation on the provided attributes; hence the authorization process can be extended via these rules to incorporate custom logic. The base system and some resource managers already provide a predefined set of attributes to the rules engine (see section "Authorization Rules" on page 33). Additionally, further attributes can be added in several ways to achieve custom decision logic.

The data and attributes that are used in rule conditions are collectively referred to as access decision information (ADI). The single attributes are name-value pairs. All available attributes form the basis of ADI that can be presented to the authorization engine and referenced from within a dynamic rule.

Custom attributes can be inserted into the user credential as additional entitlement data. Any attribute added to the user credential can be used as ADI in an XSL rule definition, these attributes can be referenced like the predefined set of attributes that is built into the user credential when it is created by the authorization engine (see section "Authorization Rules" on page 33).

Authorization Rules can also involve application context information into the authorization process. Context information includes any information that is not an entitlement but rather specific to the current request or transaction. An example is a credit card limit or transfer amount. This information is passed to the rule engine via a specific parameter of the API call that issues an authorization request. This extension point is typically used by Resource Managers to provide environment specific dynamic attributes to the rule engine.

The final source for retrieving ADI is the dynamic ADI retrieval entitlements service. These entitlement services are designed to retrieve ADI from an external source. Dynamic ADI retrieval services can be developed to retrieve ADI from an enterprise database containing custom business information. The dynamic ADI retrieval service is called to retrieve ADI when the access decision is being made; therefore it has the benefit of being able to retrieve volatile data, such as quotas, at a time when the value is most up-to-date. The concept of always retrieving data pays itself in performance, especially, if communication with remote components – such as enterprise databases – is involved.

The optimal retrieval method for any particular piece of ADI depends largely on the nature of the data itself. Volatile data – data that might change during the lifetime of the user session – has to be retrieved upon each request if using the most current value is a need. Such volatile data can only be provided by a dynamic ADI retrieval service unless the resource manager application already provides it.

Application-specific data that is nonvolatile and not user-specific is usually provided by the resource manager application. Data that is nonvolatile and user-specific is loaded into the user credential when the user is authenticated and is kept with the credential during the whole user session.

The most current values of data provided by the Resource Manager can be retrieved without major overhead although it is dynamic; however, the retrieval itself is wired into the Resource Manager's code and cannot be customized.

### 4.2.2. Application Programming Interfaces

Tivoli Access Manager provides multiple application programming interfaces that enable communication with the Access Manager components from within any developed application. The APIs split the provided functionalities into the following groups:

- The External Authentication Interface enables the development of custom authentication modules. The module is capable of accepting authentication request form the Resource Manager Runtime. The module then performs the custom authentication process and passes the result back to the runtime component. Custom authentication modules can implement any authentication process; however, the deployed Resource Manager must support external authentication modules.

- The Administration API enables programmatic execution of all the functionalities provided by the web based administrative application (web portal manager) or the command-line interface (pdadmin). This API communicates with the Policy Server to modify security policies. This results in updating the Master Authorization Database. The API is designed to increase the manageability by providing a convenient environment for administration tasks. The API ships with Java and C bindings.

- The Authorization API provides a high performance authorization interface. It basically enables standard session management functionalities and sending authorization requests. The functions provided by this API are very close to the Core RBAC System Functions (see section "System Functions" on page 22). Authorization requests consist of a single requested object, a string representing the operations and a reference to the requesting principal. The primary target of this interface is premium performance; hence the authorization function is limited to return only a yes/no answer. Any access to description fields of the protected object or access to the security policies is only provided by the Admin API. The Authorization API's only responsibility is to provide a high performance interface for querying the Authorization Server, whether a given triple of an object, an action string and a requesting principal is authorized.

Using the Authorization API is the only way of programmatically implementing custom authorization logic that has the performance production environments need. Extending Resource Manager provided functionality with loadable modules is a predominant solution that utilizes the robust core Tivoli Access Manager System, given that the Resource Manager supports pluggable modules for the desired function.

Another possibility is the development of a custom Resource Manager. Coding a Resource Manager potentially involves more development, but does not limit flexibility. When using modules, the developer is limited to the customization of the functionality the module is to provide. A custom Resource Manager could implement protection of not supported application of platforms as well as highly customized authorization processes.

### 4.2.3. External Authorization Service

While the APIs enable customized utilization of the Tivoli Access Manager components, there is also a means to externalize a given portion of the authorization process itself.

The External Authorization Service (EAS) interface provides support for application-specific extensions to the Access Manager Authorization Engine. External authorization service plug-ins force authorization decisions to be made outside of Tivoli Access Manager. It could benefit form application specific information not known to Tivoli Access Manager; however, externalizing performance critical processes can highly degrade authorization stability and performance.

An External Authorization Service plug-in is a standalone module that is dynamically loaded into the authorization service. This enables system designers to supplement Access Manager authorization with their own authorization models. The external authorization service allows imposing additional authorization controls and conditions that are dictated by a separate, external, authorization service module.

An EAS is accessed via authorization callouts, which are triggered by the presence of a particular bit in the Access Control Lists that is attached to the requested protected object. The callout is made directly by the Authorization Service.

Registering the service sets a trigger condition for the invocation of the EAS. When the trigger condition is encountered during the evaluation of an authorization request, the external authorization service interface is invoked to make an additional authorization decision.

The decisions returned by multiple External Authorization Services are aggregated with respect to a predefined weighting for each EAS deployed.

The External Authorization Service provides unrestricted flexibility regarding the authorization process; however, by-passing the robust, high performance authorization service provided by Tivoli Access Manager also eliminates the benefits from the stability and scalability of the Authorization Server.

# 5. DynRBAC: a dynamic, role based security model

The Core RBAC model - as the common denominator of almost all up-to-date security models - has evolved to the de facto industry standard in access management and is well recognized for the capability of performing large-scale authorization management.

However, as enterprise business needs tend to require finer grained and more complex security policies, the legacy approach of statically defining permitted object-operation pairs for all the roles has become cumbersome and insufficient for environments where dynamic behavior of the authorization process is desired.

Generally, dynamism of the authorization service refers to the ability thereof to involve volatile data into the access control decision process. On the modeling level, this means that authorization requests for a given object-operation pair issued from within the same session do not necessarily evaluate to the same result.

An obvious example for dynamic behavior is time-of-day login policy. The authorization decision depends on an additional condition external to the Core RBAC authorization system. Since the Core RBAC model does not provide any means to represent the influence of additional (e.g. call context) data within the authorization decision, implementers of RBAC systems are faced with having to workaround[12] this lack to satisfy customer needs. As a consequence, most leading products include custom constraints or policy types external to the Core RBAC model.

These non-standard extensions might be well suitable for different environments but introduce serious concerns regarding the following points:

- Portability of the solution: Portability, in this case, refers to the freedom of choice regarding the actual access manager product to implement the solution with. A portable solution enables the implementer to change the underlying security management product within a secured environment without the need of applying major changes to the security policies themselves. Usage of any custom policy type that might be present in one product while it is absent in all others renders a solution non-portable without major changes.

- The ability of performing formal, model driven verification and validation of the security policy. Custom extensions predominantly lack sufficient formalization (e.g. a formal model), so automated design time policy verification and error checking is hardly possible although a wide spectrum of mature model driven testing technologies are at hand.

- Generalization of the solution. As custom policy types are typically available on a per-product basis, any solution that makes intensive use of a non-standard policy type is bound to the actual access manager product and so to the domain of environments the actual product is able to secure.

[12] The term is used to stress the fact the single extensions are home-grown solutions and are hardly capable of interoperating with each other.

- As a conclusion of the above, usage of non-standard extensions are a well known factor that degrades interoperability in heterogeneous environments.

At the time of this writing, there is no consensus about a generic approach to handle and model dynamic behavior of the authorization decision process. This paper provides a Core RBAC based security model – DynRBAC - that features a generic way of representing dynamic conditions in a natural and straightforward manner.

## *5.1. The DynRBAC model and its basic concepts*

The basic goals are to keep the model simple enough to provide an effective and efficient representation of dynamic constraints yet generic and flexible to satisfy a wide range of needs. Instead of the common but less self-descriptive and less manageable approach of dynamically manipulating user-role memberships the model represents the dynamic behavior of the policy by extending the Core RBAC **`permission`** entities.

**`DynPERMS`**, the dynamic permissions define security constraints that govern the privilege of performing operations on the protected objects. As the Core RBAC permissions, they target roles and are basically operation-object pairs; however, the semantics of **`operations`** in DynRBAC differs from the Core RBAC specification.

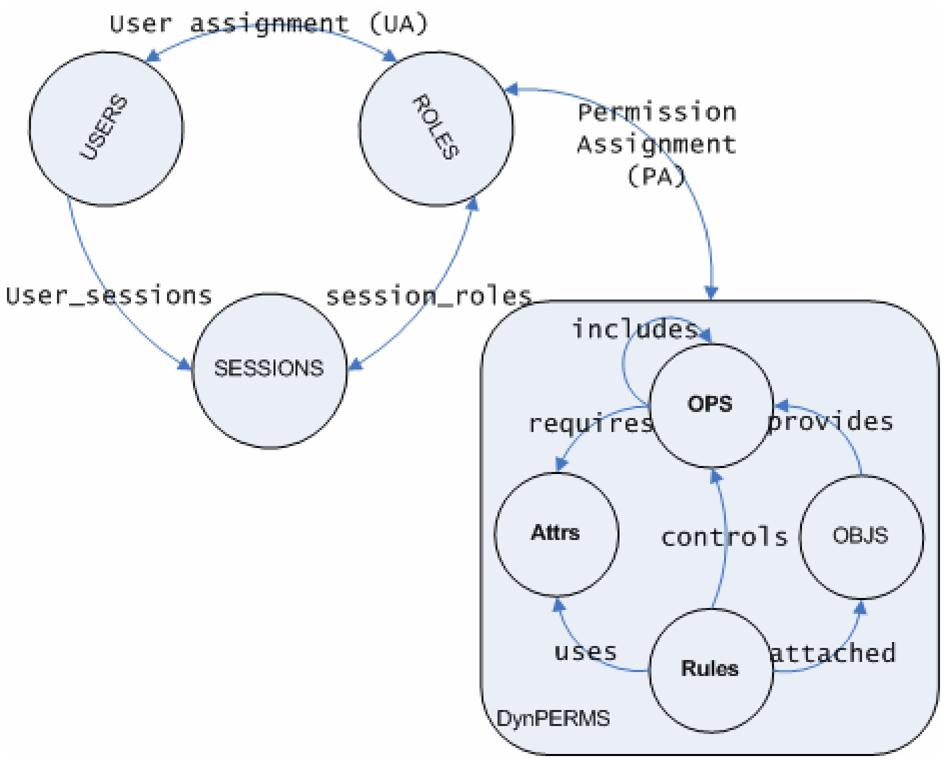


**Figure 7: The DynRBAC model**

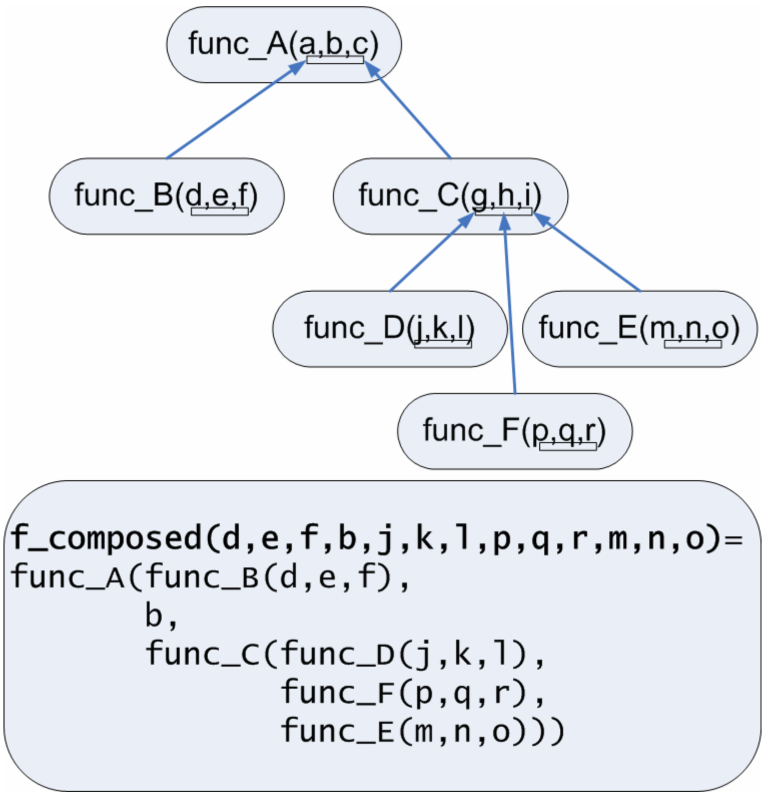
Within DynRBAC, Core RBAC **`operation`** entities (see section The Core RBAC Reference Model on page 17) are extended to accept an arbitrary – however, predefined – set of variables upon invocation. These variables might represent call-context information in the form on name-value pairs. The model does not specify the variable types that can be used; this question is left to the implementation. The name-value pairs are embodied in the **`attribute`** entities of the model.

A custom – object and user dependent - authorization constraint is represented by a **`rule`** entity that is attached to a given object (in **`OBJS`**) and controls user's privilege of performing a given operation operations (in **`OPS`**). **`Rules`** can be thought of as algebraic functions that accept a given set of **`attributes`** as arguments and return a Boolean true/false value representing the access control decision. The set of attributes evaluated by the **`rules`** needs to be available upon invocation of the operation the rule controls, which means that the attached rule can only utilize attributes that are required by the operation that triggers the rule.

The set of required attributes is statically defined for each operation and is independent of the requesting user or the requested object, but the content of the rule – that is, the expression that evaluates the given set of attributes – may be defined individually for each object-role pair. This way, the access control decision depends on the requested operation and the provided attribute set as well as on the requested resource and the user that issues the request; however, the attribute set required for a given operation is invariant: there is no runtime overhead of discovering the required attribute set upon every request. Although the attribute set is defined statically for every operation, the system has to provide a means of handling request with attribute sets inadequate for the requested operation.

A situation where the required attribute set differs from the provided set is recommended to be handled in the following way: In case the required set of attributes is a subset of the provided one, the attached rule can simply be evaluated ignoring the superfluous attributes. Authorization requests where any required attributes are absent feature a more troublesome situation. In the later case, access could be simply denied as all the necessary information has not been provided. A more flexible approach is the use of default values for all the missing attributes; however, this approach might not be suitable for all environments.

The management of dynamic permissions could include redundant tasks as similar or overlapping policies are typically applied to a large number of higher-level operations in a secured environment. To overcome this issue, the model allows for algebraic composition of the operations. Algebraic composition practically means cascading the operations, more precisely composition of the functions $f$ and $g$ is defined as $(f \circ g)(a) \equiv f(g(a))$. This feature enables inclusion of the result of one rule into another rule as input argument. Composition of functions with multiple variables is illustrated in Figure 8.



```
f_composed(d,e,f,b,j,k,l,p,q,r,m,n,o)=
func_A(func_B(d,e,f),
       b,
       func_C(func_D(j,k,l),
              func_F(p,q,r),
              func_E(m,n,o)))
```

**Figure 8: Composing functions with three variables**

With the use of composition capabilities, subroutines can be shared among composite operations and maintained in a manageable manner: Change of policy affecting common functionality shared by a large number of top level operations has only to be applied once by modifying rule governing the given subroutine.

### 5.1.1. A simple example

The following example provides a simplified illustration of the usage of DynRBAC. For the sake of simplicity, only Boolean and integral attributes are used. In addition the scenario is limited to include only one user, one role and a single protected object.

In the example, access to the company's safe is to be controlled. The result of the authorization process depends on call context data. In this example, following environmental conditions are considered relevant for the access control decision process:

- Whether the person attempting to access the safe carries a suitcase. This condition is represented by the Boolean attribute **`suitcase`**, with TRUE meaning that a suitcase is present.

- Whether the person is performing the action at night. The Boolean attribute **`night`** represents the condition.

The access control policy should deny access to the safe at night if the person is carrying a suitcase. Otherwise, access should be authorized.

The following pseudo code expresses the sample policy with a C-like syntax. The set of required attributes for a given operation is first queried by the system. When issuing the authorization request, the underlying system propagates the relevant part of the context – the required attribute set - to the rule evaluator, which then returns the decision result.

```
new operation open(suitcase, night);

//dyn_perm(role,obj,op)
dyn_perm(personell, safe, open) => NOT(suitcase AND night);

//required_attrs(operation)
required_attrs(open) => {suitcase, night};

//check_access(user, operation, context, object)
check_access(person, safe, open, {suitcase=FALSE, night=TRUE})
   => TRUE;
```

The company's safe also provides higher level operations like 'get', 'put', 'check' and 'empty'. All these operations are sequences of lower level actions but have to be authorized with one single request to achieve atomicity[13] for performance reasons. As these operations include opening the safe first, composition can be utilized in the following way:

```
new operation open(suitcase, night);

new operation getFrom(may_open, amount);

dyn_perm(personell,safe,open) => NOT(suitcase AND night);

dyn_perm(personell,safe,getFrom) => may_open AND (1000 >= amount);

//composition
new operation atomicGetFrom(amount, suitcase, night) :=
         getFrom(open(suitcase, night), amount);

check_access(person, safe, atomicGetFrom, {amount=10, suitcase=FALSE,
night=TRUE}) => TRUE;
```

The example shows how subroutines can be shared among multiple higher level operations to provide a more manageable, more structured policy.

[13] Atomicity means that only one authorization request is issued during a complex operation rather than a single request for each one of the building blocks or subroutines of the operation.

## 5.2. Functional Specification

As a Core RBAC extension, DynRBAC adopts a large portion of the functional specification featured in section “Core RBAC functional specification” on page 17. To enable the use of dynamic permissions, modifications have been made to both the administrative and the runtime functions. These modifications include the definition of new functions as well as changes to existing ones.

Only fundamental modifications are describes here, the ones needed to capture and formalize the idea at the kernel of DynRBAC. For space reasons, review functions are not to be described here as they are not essential for the proper operation of the system.

### 5.2.1. Administrative Functions

DynRBAC introduces the following administrative functions:

- The functions `required_attrs(op)` is used to calculate the set of attributes required by a given operation at administration time. In a trivial case of an operation that makes no use of composition, the function returns the attributes that are defined for the given operation. Is composition utilized, the function recursively determines the set of variables used in all the operations that are – directly or indirectly – involved into the composition. The result returned is the union of leaf attributes of all sub-operations and the given operation itself. (Compare to Figure 8). The function is formally defined as:
$$required\_attrs(op) = leaf\_attrs\_in(op) \cup \left( \bigcup_{op2 \in subroutines\_of(op)} required\_attrs(op2) \right)$$

- The function `attach_dynPerm(obj,op,role,rule)` creates a dynamic constraint that applies to the given object-operation-role triple. The input parameter `rule` is a constraint expression that evaluates to a Boolean value. An additional check is performed to ensure that the expression only references attributes that are defined as mandatory for the operation `op`. Finally, the permission assignment (PA) relation is updated to contain the recently created dynamic permission. The formal definition of the function:
$$\begin{aligned} &attach\_dynPerm(obj, op, role, rule) \lhd \\ &\quad obj \in OBJS, \\ &\quad op \in OPS, \\ &\quad role \in ROLES, \\ &\quad used\_attrs(rule) \subseteq required\_attrs(op), \\ &\quad PA' = PA \cup (obj, op, role, rule) \\ &\rhd \end{aligned}$$

- The function `dettach_dynPerm(obj,op,role)` deletes an existing dynamic permission that governs the provided object-operation-role triple.

- Additional functions to create, alter and delete operations with a given mandatory attribute set are also present but not discussed here.

### 5.2.2. System Functions

Modification to the runtime system functions of Core RBAC have also been made to enable dynamic authorization decisions. The functions involved in session management and role activation are not affected by the modifications, but the access control decision process (represented by the function `check_access`) has been adjusted to handle dynamic permissions. In addition, the utility function `satisfies(dynPerm,attrs)` is introduced to support dynamic evaluation.

- The function `satisfies(dynPerm,attrs)` can be thought of as an expression evaluator. It takes a given dynamic permission, substitutes the provided attributes and evaluates the rule. Finally, it returns a true/false value as the result of the rule execution. In the case of composite operations, the function recursively evaluates the rules that govern the subroutines. (Compare to Figure 8) This function is invoked upon every authorization request, so performance of it is crucial for authorization throughput.

- The most fundamental system function, `check_access` has been adjusted to accept an additional input argument, namely the set holding the volatile attributes. The function call is only valid if the provided attribute set is a superset of the mandatory attribute set of the requested operation. Access is granted, if the requesting session is assigned to at least one role so that the role-object-operation triple is guarded by a dynamic permission that – after evaluating the provided attribute set - allows access. Otherwise, access is denied. The following definition formalizes the procedure above:
$$\begin{aligned} &check\_access(s, obj, op, attrs) = \\ &\quad s \in SESSIONS \wedge \\ &\quad obj \in OBJS \wedge \\ &\quad op \in OPS \wedge \\ &\quad attrs \supseteq required\_attrs(op) \wedge \\ &\quad \exists r \in ROLES \left| \begin{array}{c} r \in session\_roles(s) \wedge \\ satisfies(dynPerm\_for(r, obj, op), attrs) \end{array} \right. \end{aligned}$$
$\triangleright$

## 5.3. *Compatibility with the Core RBAC*

The model extension seamlessly integrates into Core RBAC compliant environment on the modeling level. Compatibility on the modeling level has the advantage of almost trivial model transformation between the Core RBAC and DynRBAC models. This feature can be utilized in several manners.

On one hand, backward compatibility enables a Core RBAC policy to integrate into the DynRBAC system without modifications. There is no need to architect a DynRBAC policy from scratch where a fine tuned RBAC policy is already available, since a Core RBAC policy can be imported – transformed, to be precise – into a DynRBAC rule set. On the modeling level, this feature enables transparent transition from an existing RBAC system to DynRBAC.

Forward compatibility, on the other hand, refers to the ability to describe the DynRBAC policy with Core RBAC elements. The transformation of a DynRBAC model instance into Core RBAC allows benefiting from existing Core RBAC verification and validation tools. The possibility of performing formal testing even in the absence of utilities developed for the model is a premium feature especially for new models.

### 5.3.1. Transformation from Core RBAC to DynRBAC

Translating a Core RBAC policy into DynRBAC does not require any structural modification. Users, roles and sessions do not require any adjustment at all, only the permission entities of the two models differ.

Traditional RBAC operations can be represented by DynRBAC operations that do not require any attribute. The set of mandatory attributes for such permissions is the empty set, respectively. The rules of the dynamic permission do not use any attributes, they just immediately evaluate to true. As a consequence, attaching a dynamic permission to an object simply allows the given operation just the way Core RBAC's permission do.

### 5.3.2. Transformation from DynRBAC to Core RBAC

Implementing the behavior of a DynRBAC policy with Core RBAC elements involves no modification of the user, role and session entities and their relations, but requires a method capable of describing the semantics of the dynamic permissions with static building blocks.

The method presented here uses Core RBAC object, operation and permission entities to achieve the desired effect. The main idea is to unfold the dynamic rules and map each possible invocation of a dynamic permission to a separate operation. The representation itself was inspired by the manner HTML FORM data is embodied in the HTTP GET query string. A transformation of the policy described in section 5.1.1 on page 51 is presented below.

```
DynRBAC:

new operation open(suitcase, night);
dyn_perm(personell, safe, open) => NOT(suitcase AND night);

Core RBAC:

grant_permission(personell, safe, open?suitcase=FALSE&night=FALSE);
grant_permission(personell, safe, open?suitcase=FALSE&night=TRUE);
grant_permission(personell, safe, open?suitcase=TRUE&night=FALSE);
revoke_permission(personell, safe, open?suitcase=TRUE&night=TRUE);
```

The number of all Core RBAC operation that have to be defined to mimic the behavior of a single DynRBAC operation is the total number of all possible combinations of the required attribute set of the given operation. In a case of arbitrary attribute types, this is the product of the number of all possible values – mathematically, the size of the domain - of each mandatory attribute for the operation.

$$\prod_{attr \in required\_attrs(op)} | \, domain(attr) \, |$$

Considering only Boolean attributes, this number is $2^{|required\_attrs(op)|}$ as each attribute can only take two values ('true' and 'false').

Independently of the actual attribute types, the number of Core RBAC operations grows exponentially with the number of required attributes for the given DynRBAC operation. This fact raises both performance and scalability objections regarding the implementation of DynRBAC on top of a Core RBAC compliant access management product.

# 6. Design considerations and implementation

The development efforts have started with the primary goal of creating a Tivoli Access Manager based simple proof-of-concept implementation for the DynRBAC model. Due to a couple of months' work, the mission has matured to a project consisting of a stable, highly optimized authorization service accompanied by a convenient thin client administration interface and a J2EE demo application.

Instead of providing full development documentation, the following sections focus on the basic concepts and discuss various alternative approaches. After introducing and reasoning the decisions that have been taken, a description of the developed components is presented. Finally, the process of measuring the performance of the service is outlined; the benchmark results are presented and interpreted.

## *6.1. Designing the DynRBAC authorization service*

The basic goals are performance, scalability and robustness of the authorization service. Tivoli Access Manager is well recognized for providing these features and includes a handful of extension points which could be utilized to implement DynRBAC semantics. It seems reasonable to benefit from this robust and scalable architecture in order to ensure that the resulting authorization solution satisfies the desired needs.

This section guides the reader through the design process of determining the extension point and method ideal to implement DynRBAC with and the overall architecture of the solution. It also explains additional effort that has been made to provide a solution that fits into many real-life situations and environments.

### 6.1.1. Known approaches to handle dynamism

Implementing dynamic access control using Tivoli Access Manager is a challenging task that can be – and has already been - accomplished by applying several approaches; however, existing best practices and recommendations seem to lack either the performance or the robustness mission critical enterprise environments need.

This section discusses the advantages and drawbacks of existing solutions while section 6.1.2 describes a new approach that has obvious advantages in terms of performance and scalability but also introduces some limitations.

#### *6.1.1.1. Dynamic group membership manipulation*

One of the common solutions is rather a security framework than a service. It consists of code that runs outside of Access Manager, possibly in a web application container. Upon invocation, this code accepts various environmental variables and modifies the actual Access Manager user credential (with the Access Manager API) based on the provided variables. The modification typically embodies runtime group membership manipulation. If a given condition is met, the user is dynamically added to a group. This way, the approach dynamically grants permission to the users based on certain conditions[6].

The actual rules are hard-coded into the application; any modification of the policy requires modification to the application code. However, rule evaluation can be separated from the application code by utilizing an external rules engine[6].

In both ways, the application has to be aware of which variables are needed to evaluate a given rule and how the required variables can be gathered. In addition, the external rules engine could render a performance bottleneck in the authorization process and decrease throughput, or, even worse, present security vulnerability that affects the whole systems security.

Another critical point is that sensitive policy data is stored external to Access Manager's persistent storage. This point not only affects performance and robustness of the solution but also its manageability. The administrator has to be familiar with two interfaces – the Access Manager admin interface and the rule engine's interface, since consistently managing the policy from one interface is generally not possible.

#### *6.1.1.2. Developing an External Authorization Service module*

Tivoli Access Manager natively supports externalizing a portion of the authorization process via External Authorization Service modules (see section "External Authorization Service" on page 47). In addition to the standard Access Manager authorization process, a callout is made to query the EAS module, which returns an additional yes/no answer.

The rules logic attribute retrieval can be implemented without any limitation; however, the nature of the module introduces the following limitation: the EAS module can only[14] be implemented in C or C++ as it is dynamically loaded into the multi threaded Access Manager code space[7].

An advantage of the authorization service is that the secured application does not have to know anything about the required attributes or they can be gathered. All processing happened behind the scenes of the Access Manager authorization process. The fact that the application does not have to be aware of which attributes are required is a positive feature, but implementing attribute retrieval in the module can be more costly in terms of performance if no proper caching of the attributes is implemented[6].

As long as the rule evaluation is concerned, this approach has similar drawbacks as the previous one. Hard-coded rules require recompilation upon policy change whereas

[14] Of course it is possible to use any programming language as long as the interfacing between the Access Manager C data types is properly implemented, but this task also requires the extensive knowledge of the C or C++ programming languages.

external rule engines are a potential threat to premium performance and increased security.

#### *6.1.1.3. Access Manager Dynamic Rules*

Tivoli Access Manager also supports dynamic rules that are stored internally, in the policy database (see section “Authorization Rules” on page 33). These rules are XSL transformations that are evaluated upon every single request and take access control decision based on data propagated into the decision process by the underlying resource manager or custom components.

A rule can operate even if the secured application has no knowledge of the required attributes, but in this case, only a limited set of predefined attributes can be referenced. There is a possibility to use external attribute providers; however, this would heavily impact authorization performance[7].

The performance of dynamic rules can be improved by only using attributes that are already present in the access control decision process. This included resource manager provided ones and the ones that are transmitted as an input argument of the API function that issues the authorization request.

Summarizing the above, dynamic rules provide a means of storing custom logic internal to Access Manager and the possibility to manage these rules from within the same administrative interface for consistent overview. This approach enables somewhat better performance since the evaluation is done by Access Manager itself; however, extensive usage of rules that have to be executed upon each request could lead to serious performance degradation and decreased business throughput. Externalized attribute retrieval could additionally slow down the authorization process.

### 6.1.2. Implementing DynRBAC with pre-calculated rules

Almost all of the approaches outlined above externalize the rule evaluation process. Relying on any component that is not as secure, robust and well performing as the Access Manager components raises obvious objections in mission critical environments. Dynamic rules are maintained and evaluated by Tivoli Access Manager internally; however, this method still requires runtime evaluation upon each incoming request. Although redundant deployment of the authorization server can split the load among multiple servers, the extensive usage of Access Manager provided dynamic rules might seriously decrease scalability and performance.

All the above solutions focus on runtime rule evaluation whereas one key benefit of Tivoli Access Manager is its extremely high performance in making access control decisions based on static resources stored in the Authorization Database. When using any of the options listed above, the performance will be impacted with the additional processing of rule evaluation.

Compared to the approaches outlined above, this writing describes a fundamentally different approach of rule based policy enforcement: storing pre-calculated access control rules in the Tivoli Access Manager authorization database. Since the structure of the authorization database can not be altered, existing entities have to be utilized for storing the rules.

#### *6.1.2.1. Storing pre-calculated rules within Access Manager*

Almost all Access Manager entities that are persistently stored in the authorization database have fields – for example description fields – that are capable of holding additional information such as special processing instructions, but these fields are only accessible from the administration API, not from the high performance authorization API. The authorization API merely provides functions to perform access control requests in the form of authorizing a session-operation-object triple; however this task is carried out with extreme performance. The function only returns a true/false value; additional data can not be acquired[15].

The performance reasons described above imply that storage of pre-calculated rules has to be implemented on top of the session-operation-object triple. As already mentioned, dynamically manipulating the session object is not the preferred approach. Encoding the rules in the protected object space – via a Cartesian product of objects and attribute sets, similar to the method described in section 5.3.2 on page 55 - would not provide a scalable solution as the lookup time of objects could be dramatically impacted. The decision has been taken to alter the semantics of the requested operations and utilize the Access Manager provided 'action' and 'action group' entities that typical environments do not make extensive use of. The reasons are discussed below.

Tivoli Access Manager supports up to 32 action groups, each consisting of 32 individual actions allowing to represent a total of 1024 RBAC operations (compare to section "Action groups" on page 32). ACL entries store the set of permitted actions within Access Manager. ACL entries' permission portion is a 1024 long Boolean array that represents the privilege of performing the 1024 actions[16].

The first step of the authorization process is to convert the textual representation of the requested operations to the binary 1024 bit representation. Then, this bit array is masked against the 1024 bits of the ACL entries to decide whether access should be granted by the ACL attached. This task is executed with enormous performance that does not depend on the amount of actions or action groups that are used within the system. Those 1024 bits are compared by a single, atomic instruction.

Storing the results of the rules for all possible input combinations in the ACL entries' permission portion is the adequate approach of enabling dynamic permissions without major performance degradation, since no runtime evaluation is involved as far as the Access Manager authorization process is concerned. The dynamic permissions can be unfolded and stored in the form of access control lists in a manner analogous to the

[15] To be precise, there is means of acquiring various failure reasons and error codes in the form of an attribute list. These messages can contain the failure reason of a dynamic rule.

[16] This is an undocumented Access Manager feature that has been discovered by investigating the authorization database with a binary editor. The native backup and dump utilities of Access Manager also enable to discover some details of the authorization database.

method described in section “Transformation from DynRBAC to Core RBAC” on page 55. Each bit in the ACL entry authorizes the invocation of an operation with given input attribute values (see Figure 9).

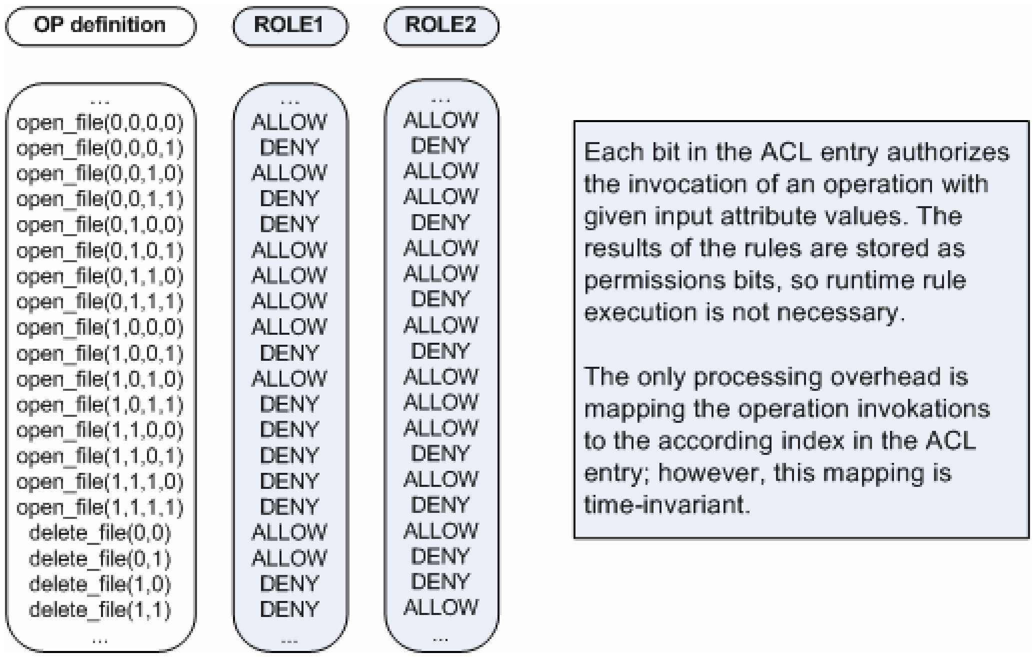

OP definition
ROLE1
ROLE2
open_file(0,0,0,0)
open_file(0,0,0,1)
open_file(0,0,1,0)
open_file(0,0,1,1)
open_file(0,1,0,0)
open_file(0,1,0,1)
open_file(0,1,1,0)
open_file(0,1,1,1)
open_file(1,0,0,0)
open_file(1,0,0,1)
open_file(1,0,1,0)
open_file(1,0,1,1)
open_file(1,1,0,0)
open_file(1,1,0,1)
open_file(1,1,1,0)
open_file(1,1,1,1)
delete_file(0,0)
delete_file(0,1)
delete_file(1,0)
delete_file(1,1)
ROLE1: ALLOW DENY ALLOW DENY DENY ALLOW ALLOW ALLOW ALLOW DENY ALLOW DENY DENY DENY DENY DENY ALLOW ALLOW DENY DENY
ROLE2: ALLOW DENY ALLOW ALLOW DENY ALLOW ALLOW DENY ALLOW DENY ALLOW ALLOW ALLOW DENY ALLOW DENY ALLOW DENY DENY ALLOW
Each bit in the ACL entry authorizes the invocation of an operation with given input attribute values. The results of the rules are stored as permissions bits, so runtime rule execution is not necessary.
The only processing overhead is mapping the operation invokations to the according index in the ACL entry; however, this mapping is time-invariant.

**Figure 9: Storing pre-calculated rules in ACL entries**

However, the advantage of premium performance – caused by the absence of runtime calculation – is accompanied by a serious drawback: the amount of required bits is exponentially proportional with the number of dynamic attributes. A detailed description of the problem is provided in section 5.3.2 on page 55. The 1024 ACL bits provided by Access Manager can be utilized to enable one dynamic operation with 10 input attributes, given that only Boolean attributes are used. This is a serious limitation regarding scalability. However, efforts have been made to overcome this limitation.

The basic idea is to break the operations with many attributes into elementary subroutines. The functional decomposition typically results in low level operations that depend on a smaller amount of environmental attributes. These low level operations allow for more effective utilization of the 1024 bits[17]. Defining only operations with 5 attributes allow for 32 dynamic operations.

The limit of five attributes also has additional advantages. Each pre-calculated rule with five attributes can be stored in a single action group[18]. This allows for solutions where dynamic and traditional policy elements can coexist in a manageable way. In this case, the name of the action group can be used to identify the DynRBAC operation.

[17] Instead of managing one operation with 10 input attributes these bits can also encode two operations with 9 attributes and so on.

[18] 32 bits are requires to store all possible combinations of five Boolean input attributes, and exactly 32 actions can be contained by an action group.

Making use of this approach, all the data is stored in the authorization database – including a descriptive name for the dynamic operation and names for the five Boolean arguments. No configuration data has to be stored outside of the authorization database.

#### *6.1.2.2. Implementing composition of pre-calculated rules*

The approach discussed above provides a high performance solution with all data stored within Access Manager. It enables the rule based authorization of 32 dynamic operations upon each protected object, where every operation accepts five Boolean input attributes. The limitation regarding the input attributes has been made to handle management and scalability issues.

Composition, as described in section "The DynRBAC model and its basic concepts" on page 49, can improve the manageability of the DynRBAC model by enabling subroutine management. As a side effect, utilizing subroutines with less attributes for the definition of high level operations with a wider attribute set becomes possible. This is the inverse process of the functional decomposition outlined above. (Also see Figure 8: Composing functions with three variables.)

On the positive side, composition enables handling more than five attributes per operation although all the low level operations only accept five attributes. However, the nature of composition also introduces the following performance issue: as all the subroutines have to be evaluated one-by-one, the time taken to authorize a composed operation will equal to the time of evaluating all the subroutines. This principle can be easily deduced form Figure 8.

One tweak to the composition evaluation has been made to provide improved performance for composed operations. The tweak exploits the fact, that evaluation speed of a single authorization request is independent of the number of requested operations.

The result of a request embodying more operations is the logical product (logical AND) of the results of the individual operations[19]. However, calculating this result takes the same time as authorizing a single operation. This fact implies that the logical product of multiple dynamic operations can be determined with the same performance as a single permission.

To get the most out of Access Manager in terms of performance, two distinct methods of implementing complex operations are provided:

- Generic composition, as featured in Figure 8, involves the evaluation of the subroutines one-by-one. Based on the tree-like evaluation structure, it is referred to as **tree-composition**. This method is an exact implementation of the mathematical definition presented in section 5.

[19] For example, authorization to read and write a given file at the same time ('rw') will only be granted if the requester has both read ('r') and write ('w') permissions on the underlying resource.

- The high performance but less flexible possibility is christened **operation-chain**. Instead of a complete call tree, it only provides the logical product of the used operations: The complex operation is only permitted if all the used low level operations are granted. The results of the low level operations are always aggregated with the logical AND operation instead of being processed by a top level rule.

Beside increased flexibility, composed operations also render the following issue: Access Manager is only capable of storing the low level operations, the ones that only accept five attributes and do not use composition. Hence, the declaration and definition of composed operations has to be stored external to the authorization database.

The decision has been made to store composition related data in a configuration file along with the code that provides the DynRBAC functionality. This configuration data is not supposed to change, so it has to be processed only one, at system startup.

Beside enabling composition, the configuration file can also be used to store the names of available operations and their attributes to provide more convenient usage of the authorization service. In the absence of this file, the application code that needs to be secure has to be aware of at least the names of available operations. Additionally, the file can also be used to provide default values for the attributes on a per operation basis.

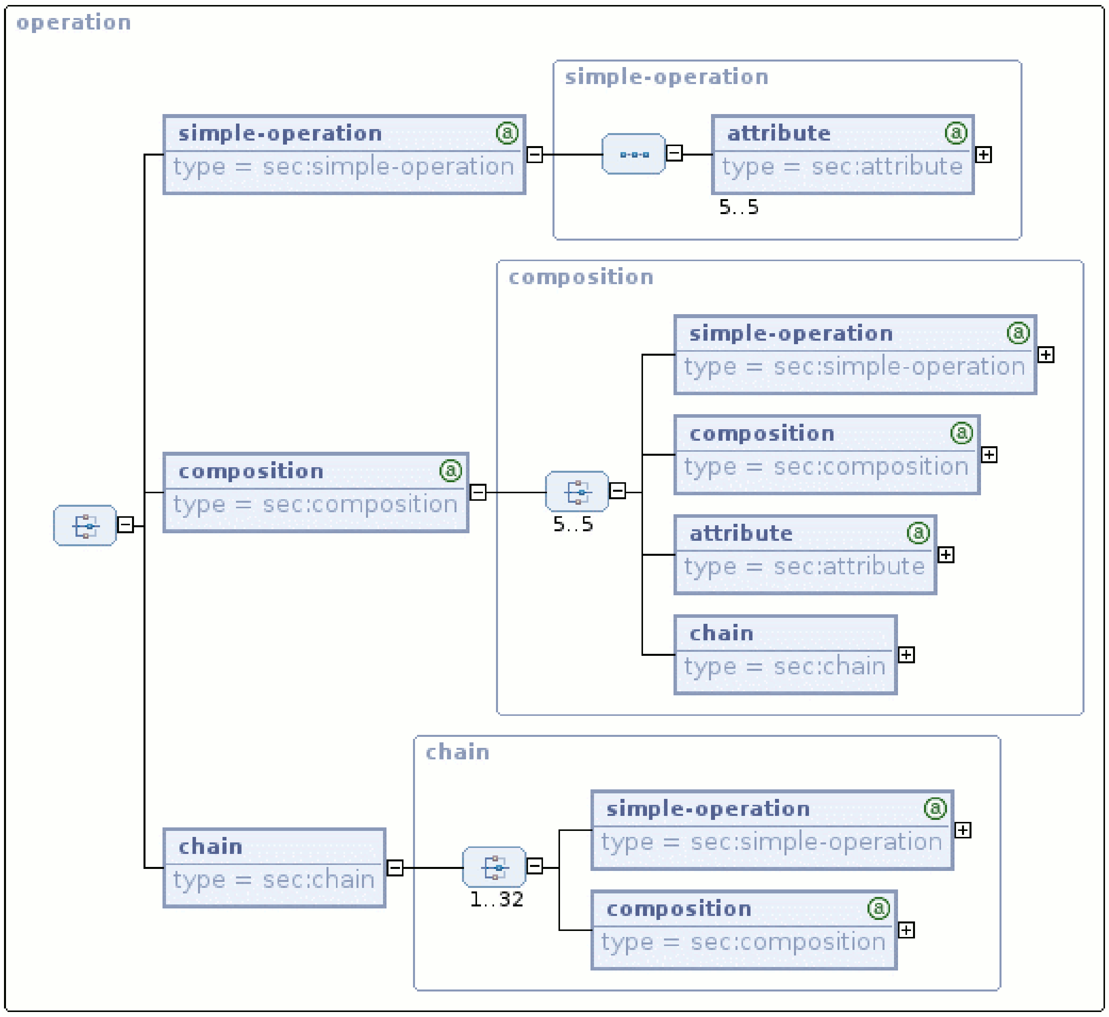


**Figure 10: Configuration schema**

Figure 10 features a graphical representation of the XML schema[20] that holds the authorization metadata structure. Each complex operation is defined by an according `operation` element (see Figure 10) that holds the call-tree or the chain definition along with the attribute names and their default values. The flexible definition also allows utilizing the result of operation chains as input parameter of higher level operations.

Compared to the approach of implementing DynRBAC without composition, with operations accepting five variables (see section 6.1.2.1 on page 60), the extension described above allows for more flexibility; however, not all the configuration data can be maintained in the authorization database. When using a configuration file, some administrative will require modification to the configuration file as well. Additionally, the file has to be protected, since it becomes indispensable for the DynRBAC authorization service.

Supposed the integrity of the configuration file is properly secured, only the maintenance overhead of administering the configuration file remains unsolved.

[20] To be precise, the figure only shows the graph of the operation definition. An arbitrary number of operations can be defined in the XML configuration file.

### 6.1.3. Standalone versus plug-in architecture

The DynRBAC implementation is rather a security service than a framework. A framework merely provides a toolkit to enable the implementation of solutions whereas a service is a centralized provider that can be communicated with from within multiple applications.

Centralizing request evaluation can lead to difficulties if many endpoints share the service provider. To eliminate this issue, the DynRBAC service provider layer has been implemented as a security plug-in that has to be deployed into the secured application's code space. An implementation that embodies a standalone DynRBAC authorization server running on a separate node can also be achieved with little effort.

This section discusses the advantages and drawbacks of the two architectures from various aspects.

- A standalone server that exposes a platform independent communication interface can be utilized by applications independent of whether they utilize the .NET framework, J2EE, of native libraries. However, implementing the communication on the client side is always involved. The plug-in, as implemented, only targets Java applications without additional interfacing; however, promoting the functions to a remote interface can turn the plug-in into a standalone server.

- The fact that the plug-in has to run on the secured host raises a security objective: the risk of exploits through byte-code manipulation that are absent in the case of a standalone server that is only communicated with through network protocols. Serious effort has been made to ensure integrity and avoid vulnerabilities (see section "The security plug-in" on page 66).

- As far as performance is concerned, the plug-in has obvious advantages over the standalone operation mode. The processing overhead of the DynRBAC attribute mapping is performed in the client JVM[21] and does not impact the host main Tivoli Access Manager components run on. This way, the additional load is split up among the secured nodes rather than a central authorization server. This allows for better scalability and performance as well. In addition, unlike the standalone mode, no supplementary network traffic is involved in performing dynamic authorization, since the attributes only undergo in-memory operations on the secured host.

- The more redundant plug-in architecture also has advantages in term of high availability. The failure of the plug-in – possibly caused by hardware or JVM failure – does not affect other plug-ins, which reside in other hosts' JVMs. Since the protected J2EE application and the security plug-in share the JVM, and so the same hardware resources, both the protected and the protecting

[21] JVM is the acronym for "Java Virtual Machine". It is the byte-code interpreter and runtime environment for the platform independent Java framework.

code would become inaccessible due a hardware failure, but other secured application would not be affected.

- Manageability of the security solution is not affected by the redundancy of the plug-in architecture as long as all the configuration data is kept in the Tivoli Access Manager authorization database – that means, if no composition is utilized. In the case of composition-aware DynRBAC plug-ins, the administration of authorization metadata becomes more complex since the XML configuration file has to be maintained for each plug-in. However, this task could be accomplished by automating the process of distributing the updated configuration file to each host the plug-in runs on.

The decision has been taken to create a security plug-in for J2EE applications. However, it should not be much effort to expose the plug-in's interface to other frameworks like .NET. Implementing a Java based standalone server is even less challenging as it only involves promoting the plug-ins method for remote invocation.

## *6.2. Implemented components*

The development process has produced a complete DynRBAC security system for J2EE environments. It includes the high performance security plug-in, a DynRBAC administration interface in the form of a dynamic web application and an additional application for demonstration and testing purposes.

As many implementation details and preferred strategies have already been uncovered, the following sections only provide a short overview of every component, without going into deep detail.

### 6.2.1. The security plug-in

The security plug-in takes the role of a custom resource manager component (see section "Resource Manager" on page 40) and communicates with the Tivoli Access Manager Authorization Server to request Core RBAC compliant authorization. Towards the protected application, it exposes a DynRBAC interface. The application can use the provided service to authenticate its users and to request authorization decisions based on dynamic application-context attributes.

For increased security, the plug-in is specially architected to provide protection even against server side byte-code manipulation and various java-based exploits. Besides applying development practices compliant to high-security java programming recommendations as found in [10][12], additional care has been taken to ensure integrity and confidentiality of the security plug-in.

A common practice of authentication and authorization frameworks is to let the Principle objects to be manages by the secured applications. A poorly constructed application can be exploited to compromise the Principal objects and uncover potentially sensitive data.

The DynRBAC security plug-in targets this issue with a special, thread-safe cache that securely manages the Access Manager provided PDPrincipal objects. Upon successful authentication, the secured application receives a security token. This token contains the user name and a special identifier of the DynRBAC session and can be used to issue authorization requests the following way: using the token's identifier, the plug-in looks up the according Principal object, transforms the DynRBAC request into a Tivoli Access Manager native query and performs authorization.

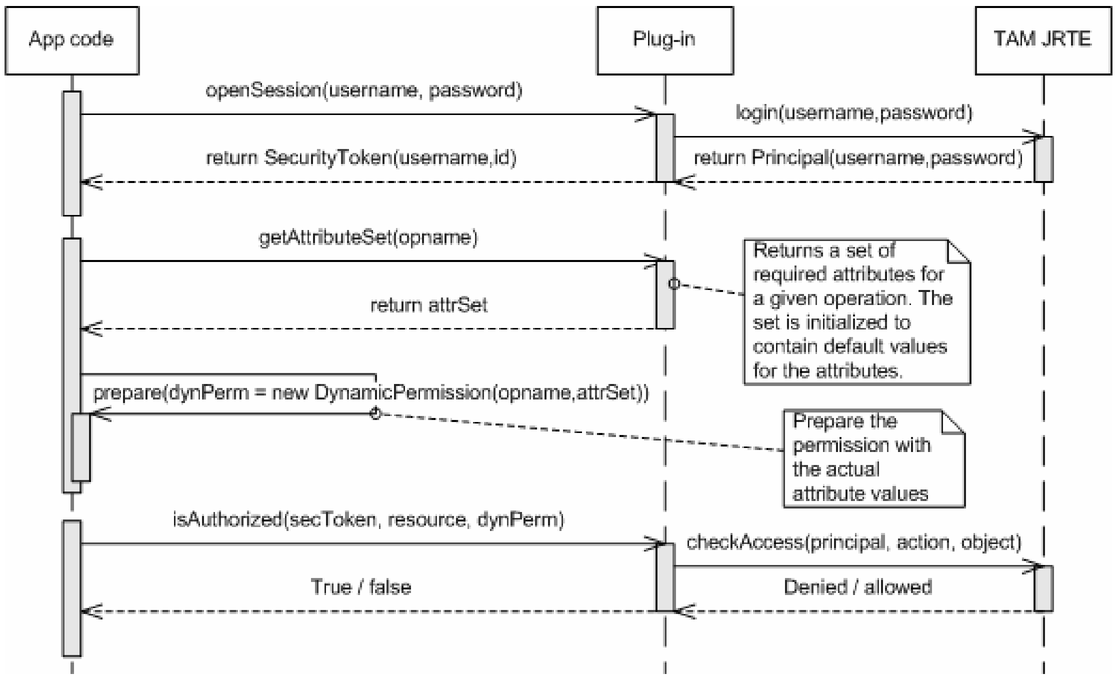

App code
Plug-in
TAM JRTE
openSession(username, password)
login(username,password)
return SecurityToken(username,id)
return Principal(username,password)
getAttributeSet(opname)
return attrSet
Returns a set of required attributes for a given operation. The set is initialized to contain default values for the attributes.
prepare(dynPerm = new DynamicPermission(opname,attrSet))
Prepare the permission with the actual attribute values
isAuthorized(secToken, resource, dynPerm)
checkAccess(principal, action, object)
True / false
Denied / allowed

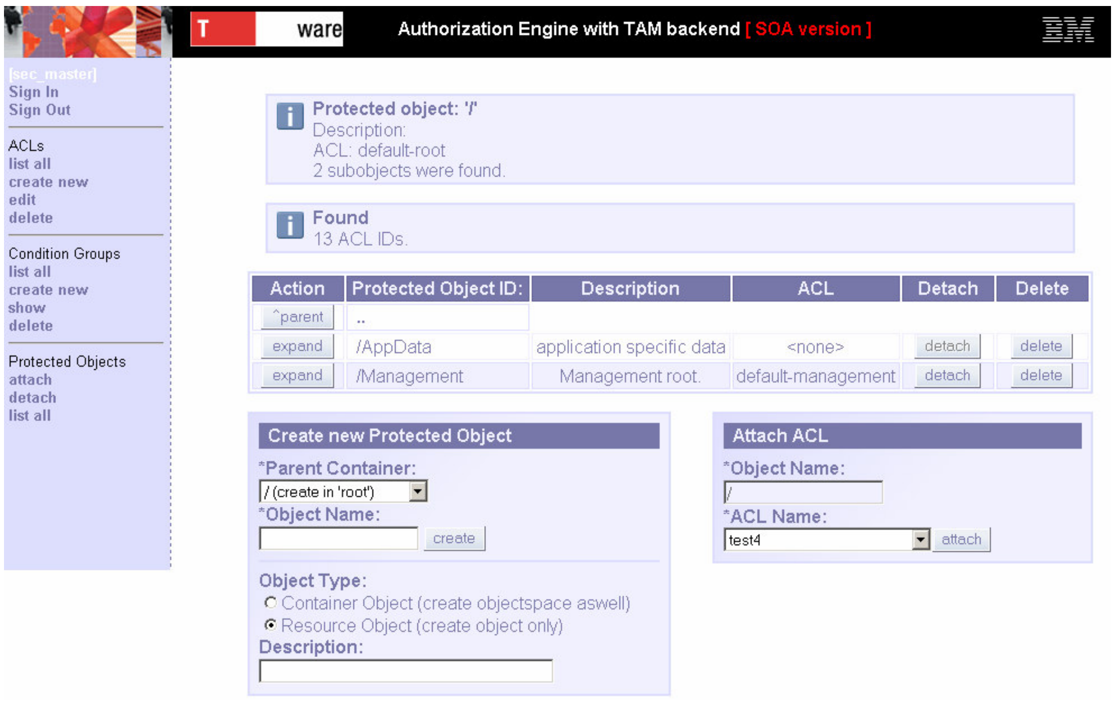
**Figure 11: High level sequence diagram of the security plug-in**

This way, the secured application never access the Principal object, only the security token that does not contain sensitive data, merely the user name and an identifier. Additionally, the implemented cache performs timeout management and allows for increased throughput by enabling multiple execution threads to perform authorization simultaneously.

Communication between the secured application's code, the plug-in and other components is outlined in Figure 11.

### 6.2.2. The administrative interface

A J2EE web application has been developed to enable maintenance of the DynRBAC security policy. It utilizes the Tivoli Access Manager administrative API to manage dynamic permissions and protected objects that are stored in the Tivoli Access Manager authorization database, hence, all communication between the management interface and Tivoli Access Manager happens over a Secure Socket Layer protected TCP channel.



[sec_master]
Sign In
Sign Out

ACLs
list all
create new
edit
delete

Condition Groups
list all
create new
show
delete

Protected Objects
attach
detach
list all

Protected object: '/'
Description:
ACL: default-root
2 subobjects were found.

Found
13 ACL IDs.

| Action | Protected Object ID: | Description | ACL | Detach | Delete |
|---|---|---|---|---|---|
| ^parent | .. | | | | |
| expand | /AppData | application specific data | <none> | detach | delete |
| expand | /Management | Management root. | default-management | detach | delete |

Create new Protected Object

*Parent Container: / (create in 'root')

*Object Name: create

Object Type:
- ○ Container Object (create objectspace aswell)
- ● Resource Object (create object only)

Description:

Attach ACL

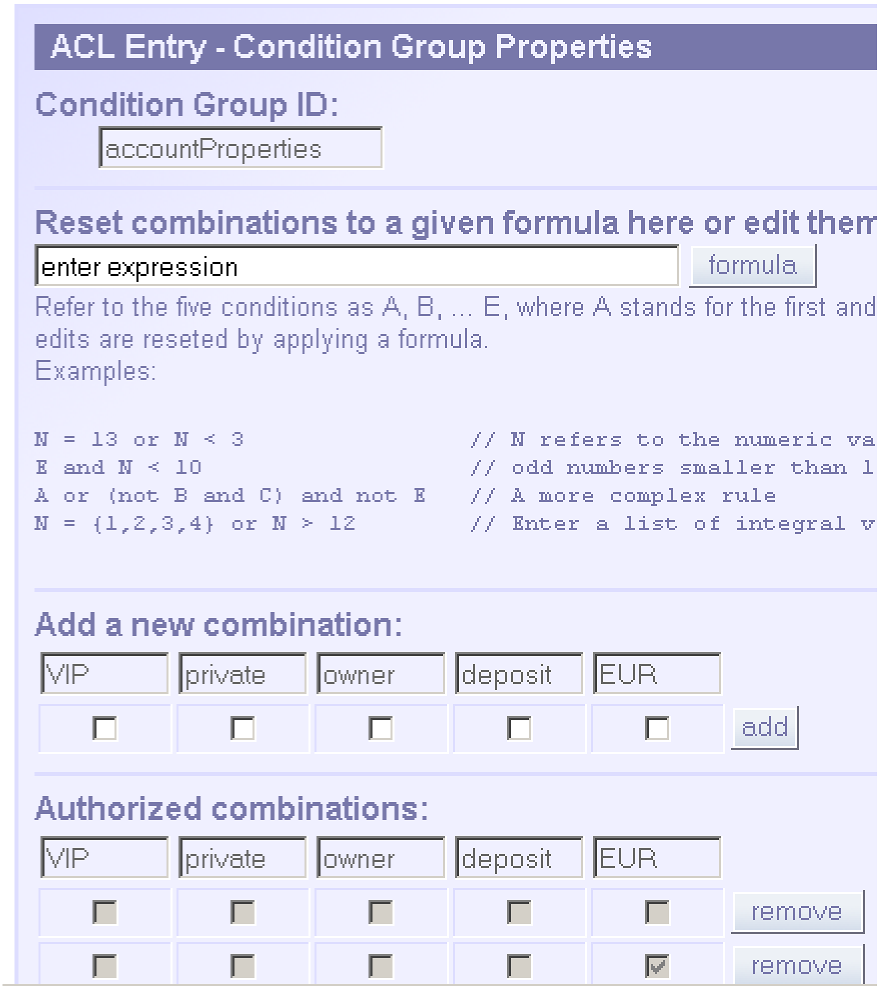
*Object Name: /

*ACL Name: test4 attach

Figure 12: Administrative Interface

The application provides functionality to manage protected objects, create, update and delete dynamic permissions and attach them to the virtual representations of the resources. It also enables the creation and management of authorization-relevant attributes.

The development of the administrative application has also included the implementation of a flexible Boolean expression parser specially tailored to compile dynamic permissions' rules to a pre-calculated format that can be stored in the Tivoli Access Manager ACL entries. Beside providing a text-based rule creation utility, the application also allows for entering or removing any single combination of the five attributes one-by-one, allowing to fine-tune a rule generated from a coarse Boolean expression (see Figure 13: Editing a DynRBAC rule).



ACL Entry - Condition Group Properties

Condition Group ID:

accountProperties

Reset combinations to a given formula here or edit them

enter expression [formula]

Refer to the five conditions as A, B, ... E, where A stands for the first and edits are reseted by applying a formula.
Examples:

```
N = 13 or N < 3                  // N refers to the numeric va
E and N < 10                     // odd numbers smaller than 1
A or (not B and C) and not E     // A more complex rule
N = {1,2,3,4} or N > 12          // Enter a list of integral v
```

Add a new combination:

| | | | | | |
|---|---|---|---|---|---|
| VIP | private | owner | deposit | EUR | |
| ☐ | ☐ | ☐ | ☐ | ☐ | add |

Authorized combinations:

| | | | | | |
|---|---|---|---|---|---|
| VIP | private | owner | deposit | EUR | |
| ☐ | ☐ | ☐ | ☐ | ☐ | remove |
| ☐ | ☐ | ☐ | ☐ | ✓ | remove |

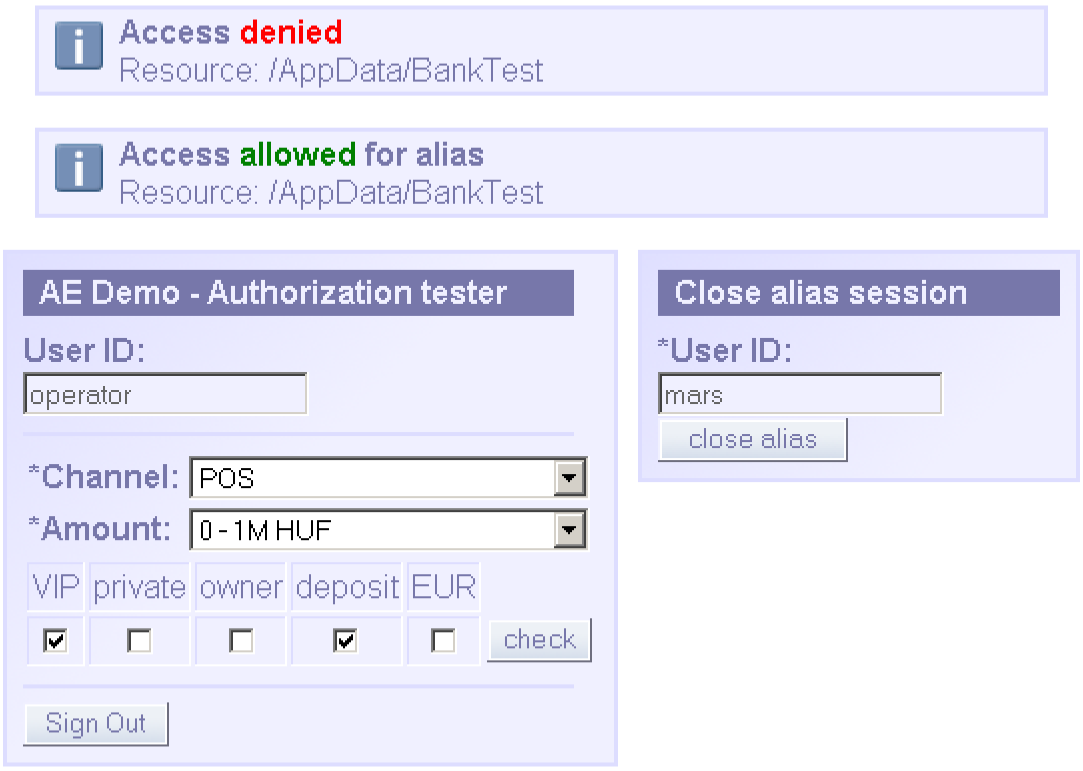
**Figure 13: Editing a DynRBAC rule**

In terms of overall application structure, the administrative interface has been designed to separate the visual representation from the provided functionality. Beside implementing the JavaServer Pages model 2 architecture, a service centric application structure has been developed so the application can easily be extended to support remote invocation of the administrative function – for example with the use of Enterprise Java Beans or Web Services.

### 6.2.3. The demo application

For demonstration purposes, a simple J2EE web application has been developed. It merely consists of a single Java servlet and a JavaServer page; it only provides a login form and a form for performing authorization requests. Upon submitting the authorization form, the application communicates with the DynRBAC security plug-in and displays the authorization decision to the user.

The application demonstrates the usage of DynRBAC within a fictive banking environment to authorize transactions. Each transaction originates form a banking account belonging to a given user. Both the transaction and the account have properties relevant for the authorization decision process. The application interface provides five checkboxes to illustrate the basic usage of the DynRBAC plug-in. These checkboxes control the values of five Boolean attributes associated with binary account properties. The transaction has two properties: the amount to be transferred and the channel the transfer is initiated from (for example a Point of Sale terminal or a cash machine). These properties can take more than two values; the transfer amount is checked to be in one of four predefined intervals and the channel can be selected from a drop-down menu listing eight different types. The permitted states of these non-Boolean values are encoded in Boolean attributes of another dynamic permission, demonstrating that arbitrary data can be represented via Boolean attributes in a more or less efficient manner.


Access denied
Resource: /AppData/BankTest

Access allowed for alias
Resource: /AppData/BankTest

AE Demo - Authorization tester

User ID: operator

*Channel: POS

*Amount: 0 - 1M HUF

| VIP | private | owner | deposit | EUR |
|---|---|---|---|---|
| ✓ | ☐ | ☐ | ✓ | ☐ |

Close alias session

*User ID: mars

**Figure 14: Demo application**

The demo application also has a special feature: it allows users with special privilege to start a second session and perform operations in the name of other users without providing the password of the other user. This functionality allows a special set of users – for example operators – to commit actions in the name of other users. This concept of alias-sessions is also incorporated into the security plug-in but will not be discussed in detail. The basic idea is to set a DynRBAC attribute whenever the operation is being performed from within an alias session.

## 6.3. Performance

High performance and scalability have been the primary goals from the beginning. To measure the effectiveness and efficiency of the implemented authorization service, the plug-in had to undergo a number of performance benchmarks. To provide representative test results a custom benchmark utility had to be developed.

After describing the test environment, the benchmark utility and the applied test method are outlined. Finally, the benchmark results are presented in the form of a diagram derived from the raw data measured. An interpretation and discussion of the results is also provided.

### 6.3.1. The test environment

The test environment has been created in the form of a VMWare image. This hardware virtualization software allows easy development and testing since provides the ability to create and restore snapshots. This feature ensures that the exactly same situation can be measured more than one time.

The virtualization software has run on a notebook with 1.5 GHz CPU and 2 GB of memory. Since the hardware supports CPU frequency scaling, special care has been taken: tests have only been run with the AC adapter plugged in.

The virtual machine has been configured to have access to 1 GB RAM, access to the CPU has not been limited. Therefore, no other activity has been performed on the notebook while tests have been run on the virtual machine.

From the software side, the test environment consists of a single node running Microsoft Windows 2000 Advanced Server. The node has been configured to run the Policy Server and Authorization Server components of Tivoli Access Manager 5.1 as well as the User Registry (Tivoli Directory Server 5.2 with a DB2 8.1 backend) and the WebSphere Application Server 5.2.

This single node deployment outlined above does not allow measuring the performance the components would have in a proper, distributed environment of multiple servers with efficient hardware resources; however, it is well suited to compare the response time and throughput of different dynamic authorization implementations within the same test environment.

### 6.3.2. The benchmark utility

A benchmark utility has been developed in the form of a Java servlet specially tailored to generate large-scale workload for the authorization service. It performs performance tests that measure the time needed to serve a given amount of authorization request. To be more precise, these test cases records the time elapsed between issuing the first and serving the last request.

The test cases are multi-threaded; multiple execution threads are engaged to simulate multiple clients requesting authorization decisions in parallel. The number of simultaneously running execution threads as well as the amount of authorization requests a single thread issues can be configured. Summarizing the above, a test case is a functional unit that accepts a thread count and the amount of requests per thread and returns the time taken to serve all the requests.

The benchmark utility performs multiple test cases with different parameters and records the time of each test case. Starting with 10 parallel threads and a job size of 10 requests per thread, it increases the job size by 10 until a size of 100 is reached. Then it resets the job size to 10, increases the thread count by 10 and starts increasing the number of requests again. When the test case of 100 threads and 100 requests per thread returns, the benchmark utility displays a table holding the recorded times of the test cases.

Three editions of the benchmark utility have been developed: one utilizing the deployed DynRBAC security plug-in, another one that uses the Tivoli Access Manager API to trigger the execution of an Access Manager Authorization Rule[22] (see section "Authorization Rules" on page 33), and a third utility attempting to access a nonexistent protected object through Access Manager API calls.

The main idea has been to compare the performance of the Access Manager native Authorization Rules – the most efficient rule implementation among the approaches discussed in section "Known approaches to handle dynamism" on page 57 – with the performance of the developed DynRBAC plug-in.

The third benchmark has been intended to provide control values. It references a nonexistent protected object; hence, no policy object can be triggered. The authorization database merely runs an object lookup that returns with no matching value and causes the authorization process to terminate immediately, without the evaluation of any policy. All the tasks triggered by this benchmark are also present when running the other two benchmarks utilities, so the third benchmark is expected to provide usable control values.

### 6.3.3. Benchmark results

The benchmark results have shown that the single server deployment is not suitable to provide a scalable solution. The encryption overhead imposed by the suboptimal java implementation of Secure Socket Layer connection has turned out to cause a major load in the case of all three benchmarks, as the many simultaneously running clients all engage encrypted channels.

In a real life situation, load caused by client side encryption is absent on the node where the Access Manager components run. Additionally, key components that have to perform large volumes of encryption are usually equipped with crypto cards, special hardware that accelerates encryption without causing additional CPU load.

To eliminate distortion of the results caused by the encryption overhead, the decision has been taken to subtract the times of the control test cases from both the DynRBAC test cases' and the Authorization Rule test cases' times. As subtracting the same value from both the benchmarks that are to be compared does not impose any 'unfair' side effect, this approach is considered as acceptable to emphasize the difference between the two dynamic solutions to be compared.

[22] For this purpose, an XSL rule has been created that provides a dynamic policy of complexity similar to the rules DynRBAC permissions can describe. The access control decision is made with regards to five attributes is both cases.

Each benchmark has been run at least three times and has returned very similar results. Averages have been calculated to clean the results by decreasing noise; however, the benchmarks could not be run enough times to provide smooth results, the diagrams still contain noticeable noise.

Figure 15 shows the diagrams generated from the results after the control values have been subtracted from both datasets. Besides showing large performance differences the two diagrams also enable comparison of the two solutions in term of scalability.

The authorization rules seem to be much more sensitive to both the number of threads and the size of jobs. This sensitivity is probably caused by the fact that the rule has to be evaluated upon every request, leading to extra CPU cycles. In comparison, access decisions based on pre-calculated rules can be cached as no runtime evaluation is involved on the server side. However, the DynRBAC benchmark also shows minor linearity most likely caused by the process of mapping DynRBAC permissions to Access Manager actions.

The results doubtlessly show that the implemented security plug-in outperforms the Access Manager provided authorization rules in both performance and scalability.

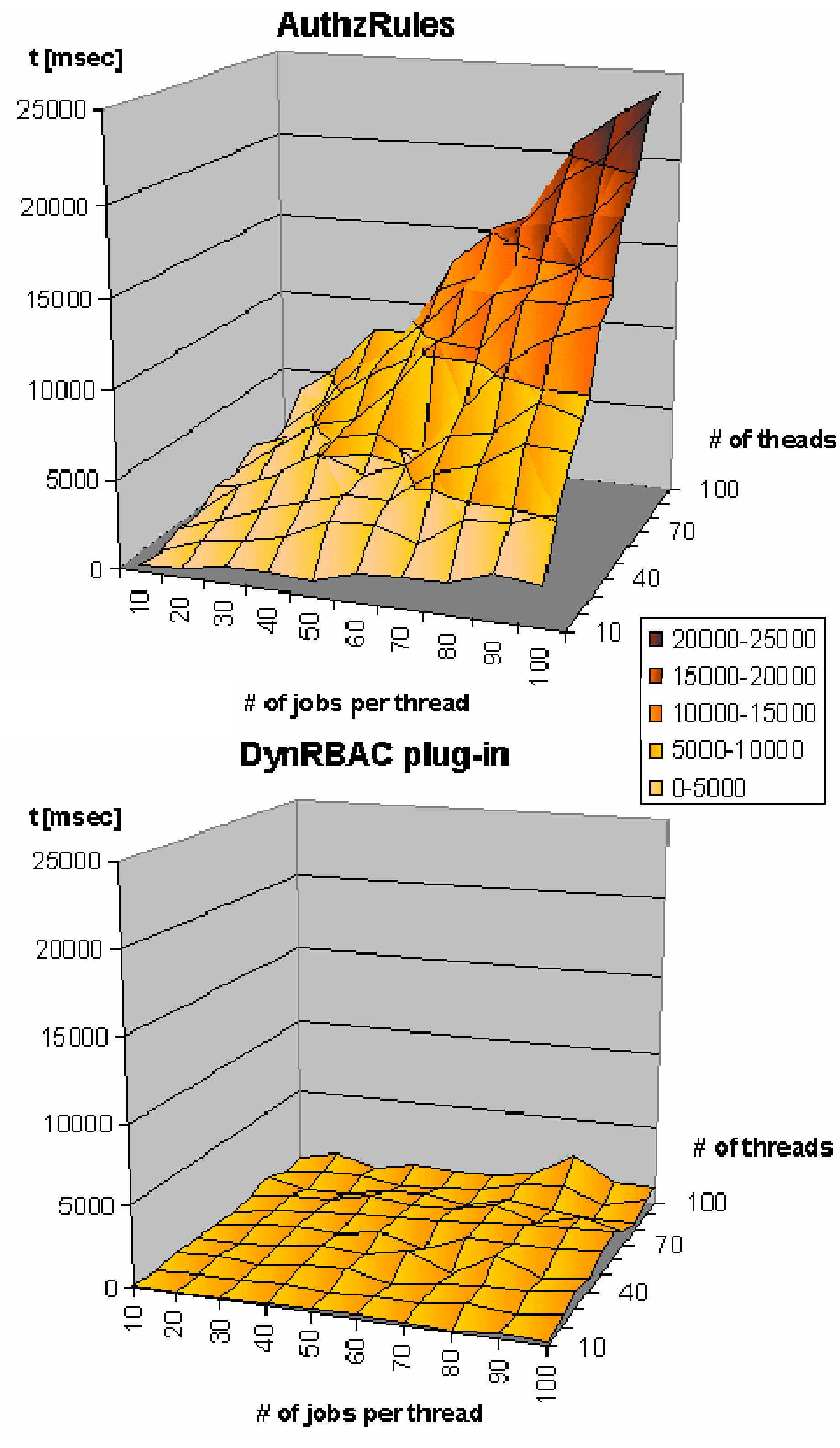


**Figure 15: Benchmark results**

# 7. Conclusion

Performance, scalability and flexibility are indispensable properties where large scale access control is required. Core RBAC is a security model that can be implemented to provide enterprise access control in a well performing manner. Core RBAC also satisfies traditional flexibility requirements, but as business needs require authorization based on dynamic conditions, vendors have to seek a solution outside of Core RBAC.

The lack of a standardized and well recognized method of handling dynamic behavior of security policies has lead to custom extensions provided by vendors of access management products. The absence of proper formalization is a serious drawback is environments where the scale and complexity of the security policy disables manual verification thereof.

As a response to these issues the Core RBAC security model has been extended to provide a generic approach to model dynamism in the authorization decision process. The DynRBAC model has been crafted to enable the straightforward implementation of a high performance authorization service without placing any constraints on the environments it can be applied to.

The decision has been taken to implement a DynRBAC layer on top of the robust and scalable IBM Tivoli Access Manager. The development efforts have started with the primary goal of creating a Tivoli Access Manager based simple proof-of-concept implementation for the DynRBAC model. Due to a couple of months' work, the mission has matured to a project consisting of a stable, highly optimized authorization service accompanied by a convenient thin client administration interface and a J2EE demo application.

Benchmark results have shown the DynRBAC implementation to be both well performing and scalable. The test results have not shown the proportionality other approaches to handle dynamism feature. Pre-calculated rules do dramatically increase performance but simultaneously imply the need to set a constraint onto the number of variables. However, this limitation can be radically loosened at the price of negligible management and operational overhead.

## Acknowledgement

First and foremost, I wish to convey special thanks to Zsolt Kocsis of IBM Hungary who has supported me all the way with inspiring comments and ideas. I am also grateful to him for providing access to the hardware and software resources used in this work. Further thanks go to dr. András Pataricza of the Budapest University of Technical Engineering for the initial guidance and for setting the proper direction for me.

Gracious thanks go further to Balázs Várkonyi of IBM Hungary for his assistance in clarifying typing errors. He has proven to be a reliable and precise proofreader even under serious temporal constraints. Further, I would like to express heartfelt gratitude to my friends and colleagues for their patience and also to other people who have supported me in any manner.

Last but not least, many thanks to my family for their support and patience.

## References

[1] Draft: Role Based Access Control Implementation Standard; January 2006 http://csrc.nist.gov/rbac/draft-rbac-implementation-std-v01.pdf

[2] Enterprise Dynamic Access Control; January 2006 http://csrc.nist.gov/rbac/draft-rbac-implementation-std-v01.pdf

[3] Proposed NIST Standard for Role-Based Access Control; August 2001 http://csrc.nist.gov/rbac/rbacSTD-ACM.pdf

[4] Wikipedia, the free encyclopedia: Formal methods; May 2006 http://en.wikipedia.org/wiki/Formal_methods

[5] Certification Study Guide: IBM Tivoli Access Manager for e-business 6.0; February 2006 http://www.redbooks.ibm.com/redbooks/pdfs/sg247202.pdf

[6] Implementing Dynamic Rules in an Access Manager Environment; July 2002 http://www-128.ibm.com/developerworks/tivoli/security/library/0724/0724_ashley.html

[7] Tivoli Access Manager for e-business Administration Guide; October 2005 http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/topic/com.ibm.itame.doc/am60_admin.pdf

[8] Tivoli Access Manager for e-business Installation Guide; October 2005 http://publib.boulder.ibm.com/infocenter/tivihelp/v2r1/topic/com.ibm.itame.doc/am60_install.pdf

[9] Implement Resource Manager on WebSphere Application Server 5.1 by Two Different Frameworks based on Tivoli Access Manager 5.1; November 2004 http://www-128.ibm.com/developerworks/tivoli/library/t-tamwas/

[10] Developing secure enterprise Java applications; April 2006 http://searchappsecurity.techtarget.com/generic/0,295582,sid92_gci1182919,00.html

[11] Understanding JavaServer Pages Model 2 architecture; December 1999 http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html

[12] Twelve rules for developing more secure Java code; December 1998 http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html