FLORIDA INTERNATIONAL UNIVERSITY
Miami, Florida

ENGINEERING ACCESS CONTROL FOR DISTRIBUTED ENTERPRISE APPLICATIONS

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Konstantin Beznosov

2000

To: Dean Arthur W. Herriott
College of Arts and Sciences

This dissertation, written by Konstantin Beznosov, and entitled Engineering Access Control for Distributed Enterprise Applications, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

_____

Geoffrey Smith

_____

Raimund K. Ege

_____

Ravi S. Sandhu

_____

Yi Deng, Major Professor

Date of Defense: July 18, 2000

The dissertation of Konstantin Beznosov is approved.

_____

Dean Arthur W. Herriott
College of Arts and Sciences

_____

Dean Richard L. Campbell
Division of Graduate Studies

Florida International University, 2000

DEDICATION

To Alla, Vladimir, Valerij, Olga, and Alissa

ACKNOWLEDGMENTS

Kent Wreder directed my original steps towards addressing the problem of access control for distributed enterprise applications. Thanks to his encouragements and support, I had a very quick start. Eric Butler and Eric Navarro introduced me to the work of IT architects and were great colleagues.

My advisor, Yi Deng, was an unending source of useful and pragmatic advice. He helped me to see the problems more broadly and at the same time be more specific. His patience, compassion and belief in me were instrumental in the completion of this journey. Thanks also to him for reading my dissertation and suggesting structural and stylistic changes that made it much more valuable and comprehendible. Yi, thanks for everything.

I would like to thank Yi's students -- Suresh Chegireddy, Banaglore Gururprakash, Luis Espinal, Manish Mahajan and Nathan Vuong -- for their input during our numorous discussions on CAAS. Luis deserves special thanks for he has been a great help to me throughout my research, we did most of CAAS design and implementation together, and I enjoyed working with him very much. SCS Computing Support, directed by Steven Luis, was excellent in providing necessary help with the computing environment during the course of my study and research.

The framework for implementing RBAC using CORBA Security was motivated by the communications with the participants of RBAC workshops of 1997-1999, who also helped me to gain insights of the RBAC model. My understanding of the CORBA authorization

model was largely influenced by active OMG SecSIG members -- Bob Blakley, Bret Hartman, Polar Humenn, and Jishnu Mukerji.

The RAD architecture was invented during the work on the proposal to OMG's Healthcare Resource Access Control RFP and was developed together with John Barkley, Bob Blakley and Carol Burt throughout numerous e-mail rounds, conference calls, and a number of very productive and enjoyable meetings. Without them, the architecture would not exist in its current form. They also taught me a great deal about designing practical solutions and writing specifications. Other members of the OMG's SecSIG and CORBAmed were an extensive source of the comments and feedback on stating the problem and on the architecture.

Finally, special thanks go to my parents and my brother for supporting me with their love and understanding while I was applying to and during my years in graduate school. Thanks to Dina Evans for the support and love she gave me. Olga and Alissa were the source of my inspiration, encouragement and love last year, which was the most intense and demanding period of the work on the dissertation.

Kim Lumpkin from FIU Learning Center provided invaluable help by spending with me many hours on converting my "Russglish" into proper English. Without her guidance the dissertation would be unreadable. Thanks to his very careful reading, Geoffrey Smith pointed at many typos and errors in the text.

My research was funded by the National Science Foundation and mainly by donations from Baptist Health Systems of South Florida.

ABSTRACT OF THE DISSERTATION

ENGINEERING ACCESS CONTROL FOR

DISTRIBUTED ENTERPRISE APPLICATIONS

by

Konstantin Beznosov

Florida International University, 2000

Miami, Florida

Professor Yi Deng, Major Professor

Access control (AC) is a necessary defense against a large variety of security attacks on the resources of distributed enterprise applications. However, to be effective, AC in some application domains has to be fine-grain, support the use of application-specific factors in authorization decisions, as well as consistently and reliably enforce organization-wide authorization policies across enterprise applications. Because the existing middleware technologies do not provide a complete solution, application developers resort to embedding AC functionality in application systems. This coupling of AC functionality with application logic causes significant problems including tremendously difficult, costly and error prone development, integration, and overall ownership of application software. The way AC for application systems is engineered needs to be changed.

In this dissertation, we propose an architectural approach for engineering AC mechanisms to address the above problems. First, we develop a framework for implementing the role-based access control (RBAC) model using AC mechanisms provided by CORBA Security. For those application domains where the granularity of CORBA controls and the expressiveness of RBAC model suffice, our framework addresses the stated problem.

In the second and main part of our approach, we propose an architecture for an authorization service, RAD, to address the problem of controlling access to distributed application resources, when the granularity and support for complex policies by middleware AC mechanisms are inadequate. Applying this architecture, we developed a CORBA-based application authorization service (CAAS). Using CAAS, we studied the main properties of the architecture and showed how they can be substantiated by employing CORBA and Java technologies. Our approach enables a wide-ranging solution for controlling the resources of distributed enterprise applications.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS

Adage	Authorization toolkit for Distributed Applications and Groups
ADS	Authorization Decision Server
AL	Authorization Language
CAAS	CORBA-based Application Authorization Service
CORBA	Common Object Request Broker Architecture
DAC	Discretionary Access Control
DCE	Distributed Computing Environment
EPAC	Extended Privilege Attribute Certificate
FTP	File Transfer Protocol
GSS API	Generic Security Service Application Programming Interface
GAA API	Generic Access Control Application Programming Interface
IIOP	Interoperable Inter-ORB Protocol
JAAS	Java Authentication and Authorization Service
MAC	Mandatory Access Control
ORB	Object Request Broker
PAC	Privilege Attribute Certificate
RAD	Resource Access Decision
RPC	Remote Procedure Call
SESAME	Secure European System for Applications in a Multi-vendor Environment
RBAC	Role-Based Access Control
TCB	Trusted Computing Base

1 Introduction

Software systems today are increasingly integrated and interconnected to achieve organization-wide, agency-wide and industry-wide automation and interoperation. Such integration results in enterprises that consist of autonomous, heterogeneous and distributed systems called enterprise software systems. Applications within each enterprise may be developed independently and based on different design and technology. National defense, industry, commerce and health care are increasingly dependent on the function of these systems [Sumner 1999].

Because of the magnitude and complexity of distributed systems and information resources interconnected by the Internet and/or enterprise networks, designing security mechanisms that protect the systems and resources becomes an increasingly complex and difficult challenge. This is why it is an essential concern to every enterprise [NSF 1999].

The problem of securing information enterprises has been the focus of intensive efforts from industry. As a result, several well-known security system architectures and models for network, operating, DBM, and middleware systems have been developed for constructing scalable and flexible security for distributed environments. This represents significant progress yet it is only the first step for attaining the goal. The issues that remain are the following: handling complex and fine-grained security policies; supporting changes not only in application systems and their underlying platforms, but also in business process and

security policies, as well as in user population and their roles; supporting dynamic configuration of enterprise applications without affecting security integrity; and achieving required performance.

In this dissertation, we consider one particular security functionality -- access control (AC) [Sandhu 1994]. It is a necessary defense against a large variety of security attacks on information enterprise resources. However, the control of access to application resources more and more needs to be fine-grain and support the use of application-specific factors in authorization decisions. It must also consistently and reliably enforce organization-wide authorization policies across enterprise applications.

The existing network, OS, DBMS, and middleware technologies are inadequate for doing such control, and they will never be because they are designed for general purpose usage, and their controls are too coarse and concern only certain resources [CIST-NRC 1999]. Because of this, application developers resort to embedding AC functionality in application systems in order to support complex, fine-grain and context dependent authorization policies.

The coupling of AC functionality with application logic causes significant problems. Enterprise security administrators end up having to configure AC logic on application-by-application basis [Beznosov 1998a]. This application-based multiple point AC makes enterprise security administration tremendously difficult, costly and error prone [Beznosov 1997, Wilson 1997], makes it harder to change security policies and control mechanisms, and makes it difficult to develop, change and dynamically reconfigure application software [Beznosov 1999b, Grimm 1999, Hale 1999].

The way application systems are constructed needs to be changed so that the problem of protecting application resources is addressed and yet the systems can be developed, integrated, and managed in the enterprise computing environment in a cost-effective way.

1.1 Objectives of the Work

In this dissertation, we propose an architectural approach for engineering AC mechanisms capable of addressing the problem of controlling the access to enterprise application resources. The approach is twofold. First, we develop a framework for implementing role-based access control (RBAC) model using AC mechanisms provided by CORBA Security. For those application domains where the granularity of CORBA controls and expressiveness of RBAC model suffice, our framework addresses the stated problem. The second and main part of our approach develops an architecture for an authorization service that addresses the problem of controlling access to distributed application resources, when the granularity and the support for complex policies in middleware AC mechanisms are inadequate and application developers embed additional AC functionality in their systems.

1.2 Summary of the Main Results

Security provided by middleware technologies is important and necessary for protecting distributed applications and their resources. Therefore, it is important to have means for modeling authorization policies using middleware AC mechanisms in order to fully utilize them. We define a configuration of the CORBA protection system state. Using the definition language, we specify an algorithm for authorization decisions in CORBA security. The configuration along with the authorization algorithm mathematically define the state and the behavior of the CORBA Security authorization system.

Using the previously defined configuration of the CORBA protection system, we show how RBAC models could be supported by the CORBA Security service. We provide definitions of $RBAC_0$ and $RBAC_1$ implementations in the language of CORBA Security. Furthermore, we describe what is required from an implementation of the CORBA Security service in order to support $RBAC_0$-$RBAC_3$ models. Our approach allows an implementation compliant with the CORBA Security specification to support RBAC models. This work advances the understanding of the CORBA AC mechanism's capabilities and by this maximizes their utility, which is vital to the use of middleware in protecting application resources.

Our main contribution is the resource access decision service (RAD) -- a novel architectural approach for constructing authorization mechanisms that are functionally adequate for protecting fine-grain application resources using application-specific information in authorization decisions. The approach allows separation of application and authorization logic, which makes application development, deployment, and management more cost-effective. It also enables consistent enforcement of organizational policies across multiple applications. We show its functional capabilities by modeling authorization policies that require the use of such application-specific information as the relationship between the user and the resource owner.

Through the prototype implementation of the CORBA-based application authorization service (CAAS), which was constructed according to RAD architecture, we gained some important insights on the design of RAD-based authorization services. In addition, we showed how the main features of RAD architecture, such as flexibility, configurability, and

extensibility, can be substantiated using standard CORBA middleware and Java programming technologies. Our experience of developing CAAS provides a guideline to the design of RAD-based services.

Using CAAS as a test-bed, we obtained quantitative estimates of CAAS performance for different compositions of its components. We found that depending on the ratio of the application execution time to the number of authorization requests and the performance constraints, one or the other CAAS configuration can deliver the required performance.

1.3 Dissertation Content

In the next chapter, we give background information on the subject of access control in computer systems, explain main concepts and terms, and then introduce the area of application-level access control. Then we state in detail the problem addressed in this dissertation. Finally, we define a framework for evaluating the existing technologies, related work and our approach.

Chapter 3 provides an overview and analyses of the technologies, where we show that the existing middleware technologies are important and necessary for protecting resources of distributed enterprise systems but are not sufficient. The chapter also contains a review of the related work conducted in the research community.

In Chapter 4, we propose CORBA protection system configuration that formally defines the state of the system. Using the definition, we specify an algorithm for making AC decisions in CORBA, and show how RBAC models could be supported using CORBA Security.

Chapter 5 introduces our main contribution -- RAD service architecture. There, we also demonstrate its utility on examples with complex access control policies.

We present a design of CAAS and show how the main features of RAD architecture can be substantiated using CORBA and Java in Chapter 6.

Chapter 7 discusses performance experiments, which we conducted using CAAS, and draws conclusions from the results.

We conclude in Chapter 8 by discussing the achieved results and outlining what should be done next in the problem area.

2 Background and Problem Statement

Before stating the problem addressed by this dissertation in detail, it is necessary to give background information on the subject of access control in computer systems, explain main concepts and terms, and then introduce the area of application-level access control. This chapter's objective is to provide all of the above.

In addition, we define criteria for critiquing existing technologies, related work, and for analyzing the solution we propose. In short, the criteria are the granularity of protected resources, the support for policies specific to the application domain, the variety of information available for making authorization decisions, the use of application-specific information in authorization decisions, the consistency of policies across multiple applications, the support for application and enterprise evolution, and performance and administration scalability.

2.1 Background Information and Terminology

Security of modern software systems is conventionally achieved via *protection* and *assurance*, as shown in Figure 2-1. The former is usually provided by some security subsystems or mechanisms, which are designed to protect the system from specific *threats*. A threat is any potential occurrence that can have an undesirable effect on the assets and resources associated with a computer system [Amoroso 1994]. Protection is based on the

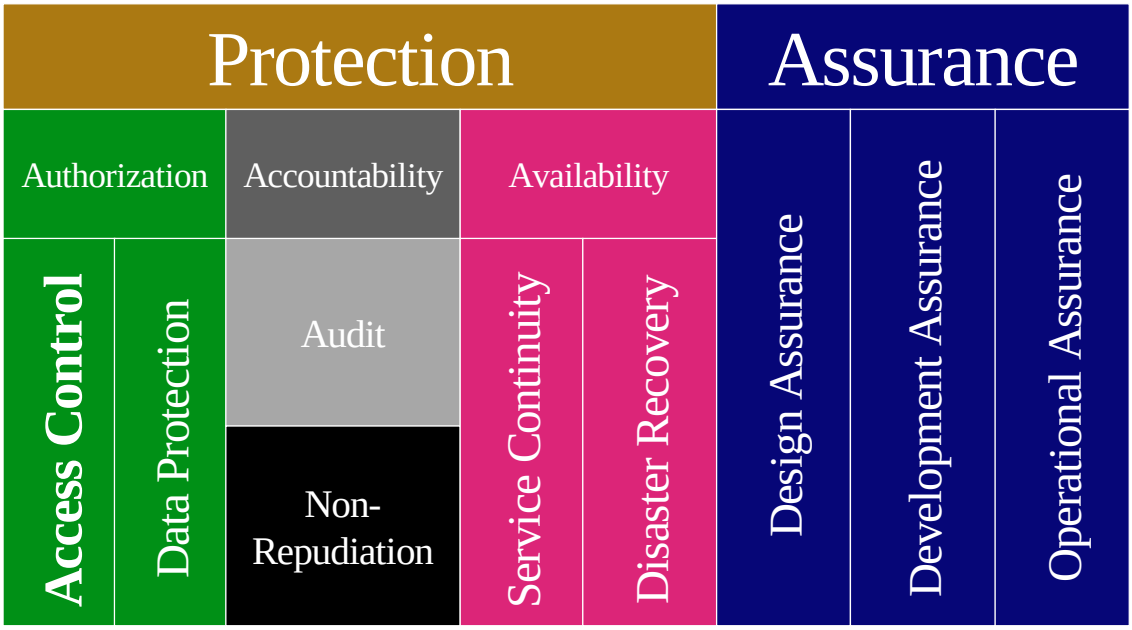


Figure 2-1. Main Concepts of Computer Security

premise that it is possible to list most of the threats which can happen in a computer system, and it is possible to build mechanisms which can prevent the threats [Blakley 1999]. The protection mechanisms can be classified in three groups: accountability, availability and authorization. *Accountability* mechanisms make sure that users (or programs executed on behalf of them) -- conventionally called *subjects* -- are held accountable for their actions towards the system resources and services. Sometimes, subjects are also called *principals*. We will use these two terms interchangeably. *Availability* mechanisms ensure either service continuity or service and resource recovery after interruption. *Authorization* mechanisms ensure that the rules governing the use of system resources and services are enforced. They are further qualified as either access control or data protection ones. *Access control* (AC) mechanisms allow system owner to enforce those rules when rules check and enforcement are possible. The term “authorization” also implies the process of making AC deci-

sions. When checking and/or enforcement of the rules are not possible, data protection mechanisms, such as data encryption, are used.

The structure of traditional AC mechanisms can be viewed using the conceptual model of *reference monitor* [Anderson 1972]. A reference monitor is a part of the security sub-system, responsible for mediating access by subjects to system resources (traditionally called *objects*), as illustrated in Figure 2-2. The mediation consists of making authorization

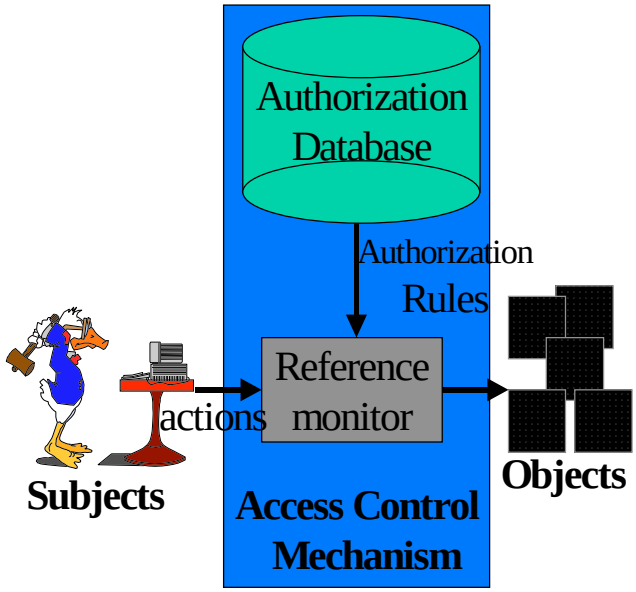


Figure 2-2. Reference Monitor

decisions, by checking access requests against authorization rules from the authorization database -- a storage of such rules -- and enforcing them. A set of the rules is sometimes called a *policy*. Conventionally having subject-action-object structure, authorization rules specify what subject(s) can perform what action(s) on what object(s). Permitted actions are also called *access rights*. Thus a subject has a particular access right to an object if it can perform the action, defined by the right, towards that object. Furthermore, all authorization

rules can be conceptualized into *access matrix* [Lampson 1971], where there is a row for each subject and a column for each object, and each cell specifies access rights granted to the subject for the corresponding object.

In order to make an authorization decision, a reference monitor takes as its inputs authorization rules and three groups of information: 1) about the access request, 2) about the subject who made the request, and 3) about the object to be accessed. It is necessary to discuss what information is in these groups because we will use it for stating the problems, evaluating the existing and analyzing our work.

The information about access request usually carries the request type, for example "read" in a request for reading a file. However, some application domains have a need for AC decisions based on additional attributes of the request. For instance, a banking system might deny a withdrawal request if its amount exceeds a pre-determined threshold.

Information about the subject can be divided in two types -- related and unrelated to security. Originally, only security-related information was used in AC decisions. Controlled by security or user administrators, this information describes subject's identity, group membership, clearance, and other *security attributes*. Some times, we will use term *privilege attributes* to refer to those security attributes that are intended to be used for nothing else but AC.

In some application domains, security-unrelated information about the subject needs to be taken into account. For example, access to rated materials in public libraries could be granted according to the age of the accessing user. Another example is information derived

from the organizational work-flow process. This information is not controlled by security or user administrators and it is not always provided to the reference monitor in the form of subject security attributes. The monitor needs to obtain it via other means. The information about the object to be accessed can also be divided into the related and unrelated to security. An example of an object security attribute is its security level. All this information is used for evaluating authorization rules.

Depending on the capabilities of a particular AC mechanism and the availability of information about the subject, request and object, either only limited or elaborate information can be accessible for making authorization decisions. This information availability will be used as a criterion for evaluating expressiveness (or power) of AC mechanisms.

AC mechanisms are part of most operating, database management (DBM), and middleware systems. They are also present in such control systems as firewalls, and many applications.

2.2 Controlling Access to Application Resources

Application resources can be in the form of data processed by applications, their services (e.g. Telnet [Postel 1983], SMTP [Postel 1982] or WWW servers), particular operations performed on them (e.g. GET access requested from a WWW server via HTTP protocol, operation invocation on a CORBA-based application server), or even menus of the application interface.

Some application resources, such as files, database records, or network sockets, can be protected by an operating, DBM, or middleware system. However, there are resources that

are application-specific and not recognized by anything except the application itself [CIST-NRC 1999], for example the execution of particular parts of the application business logic. In other words, the granularity of application-specific resources is finer than of general-purpose computing systems. Figure 2-3 illustrates the difference in the scope of middleware

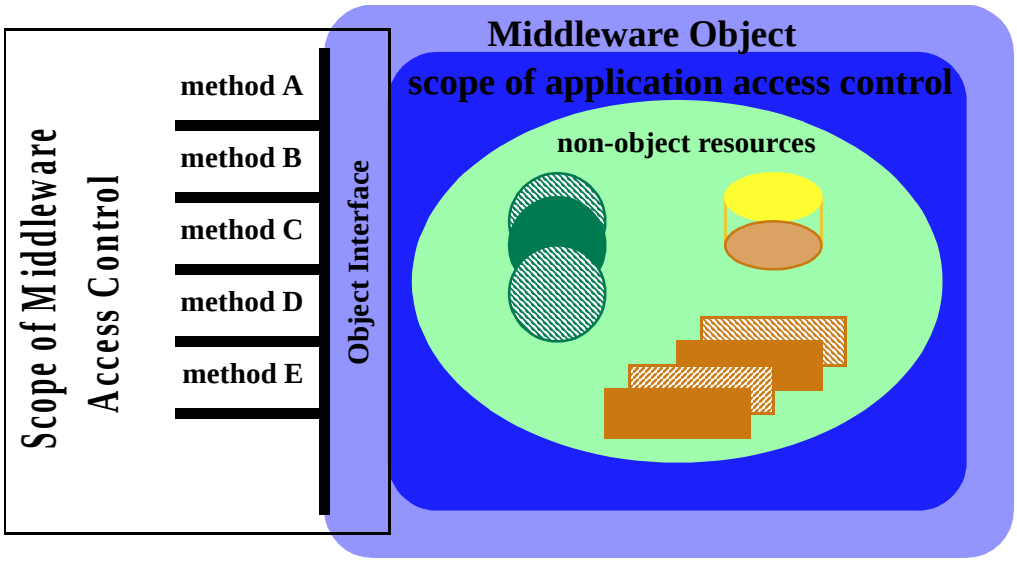


Figure 2-3. Separation of Access Control Scope between Middleware and Application

and application-level AC. This is one essential distinction between application-level and general purpose AC.

Another vital difference is that authorization rules used for application-level AC require the use of such information about access operations, subjects, or objects, that is specific to the application domain or more elaborate (more expressive) than the information used by AC mechanisms of general purpose systems.

In order to meet the requirements, applications commonly have their own AC mechanisms in addition to the use of those provided by the underlying general purpose systems. And this practice is becoming more and more commonplace than exceptional.

2.2.1 Examples

Let us present fragments of actual interfaces brought from formal OMG specification of Person Identification Service (PIDS) [OMG 1998b]. They illustrate the need to exercise AC on the level of method argument and/or return values. Each PIDS-compliant application server must provide access to its functionality and data via interfaces defined in PIDS specification. Let us consider some operations specified by PIDS interfaces.[1]

First we demonstrate the need to control what values of operation arguments can be used by different subjects. The processing of invocations on the operations shown in Figure 2-4 require access control on the level of input argument values. The

```
interface DemographicAccess : IdentificationComponent
 {
   Profile get_profile ( in PersonId person_id,
       in SpecifiedTraits specified_traits);

   void update_traits( in PersonId id, in Profile the_profile );
 };
```

Figure 2-4. PIDS `DemographicAccess` Interface

`get_profile()` operation returns a profile, which is a collection of traits describing a person, or its subset available to the PIDS service. The traits provided as input argument indicate what subset of the profile is required by the client [OMG 1998b]. The `update_traits()` is used to modify an already existing profile by adding new traits and overwriting any values for existing traits that are also passed in. Any traits already stored for that person but not mentioned in the provided profile are left intact [OMG

1. The interfaces are from module org/omg/PersonIdService. We omit some elements of operation definitions, such as exceptions, since their presence does not contribute in the discussion. Also, only relevant operations are reproduced in the definitions.

1998b]. Because different person traits could have different confidentiality level it is realistic to foresee security policies that require PIDS-compliant server to control what subject can access what traits of what person in what mode (e.g. “read” or “modify”).

Second, we show that control over data returned to the client has to be enforced too. Operation `get_all_ids()` in Figure 2-5 returns profiles for all patients the service

```
interface SequentialAccess : IdentificationComponent
 {
  ProfileList get_all_ids( in TraitNameSeq traits_requested,
      in IdStateSeq states_of_interest);
 };
```

Figure 2-5. PIDS `SequentialAccess` Interface

knows about that match one of the provided “states.” The returned profiles contain the traits indicated by the “state” parameter. The service is not supposed to return those profiles, to which the subject does not have access even though it might list them in the operation input.

2.3 Problem Statement

The central problem we address in this dissertation is inadequacy of the architectural solutions for controlling access to enterprise distributed applications and their resources. In order for an AC mechanism to be sufficient it must support functional requirements. In addition, the mechanism architecture must support and be supported by the architecture of the information enterprise where the system is installed. The current solutions are inadequate because they are either functionally deficient in protecting fine-grain application resources according to the application-specific policies or do not support the objectives of the information enterprise architectures, or both.

In this section we define the problem. First, we expand on the subject of information enterprise architecture problems and their causes and show what architectural properties a system must have in order to support the enterprise architecture. Then, we zoom into the discussion of the requirements for controlling access to application resources at the system scope. Finally, we substantiate the general discussion with real-life example of a health care enterprise and describe concrete issues with AC in it. We complete the problem definition with a summary.

2.3.1 Information Enterprise Perspective

It is necessary to place the problem of engineering access control to application resources in a larger context in order to discuss the requirements and the validity of the work. Such a context is the architecture of information enterprises (IE) because distributed applications are parts of them, and the goals of engineering distributed applications should support the goals of IEs.

We must clarify the notion of an enterprise before discussing the problems that have to be addressed at this level. An intuitive perception of an IE tells us that it is a system of information systems. Such a description, although correct, is far from rigorous. We will use the following more precise definition of an enterprise as “an organizational scope upon which a common set of information technology policies can be imposed” [Mowbray 1997]. The technological scope of IE is defined by the following hierarchy: object, module, collection of modules, framework, program, application, system, department, enterprise, conglomerate enterprise, industry enterprises, and global infrastructure.

We discuss extensively the problems of information enterprise architectures (IEA) [Beznosov 2000]. Here we merely summarize the main results for the sake of briefness. The major problems encountered in the IEA construction are low semantic compatibility of resulted systems, high re-alignment and maintenance cost, and its exponential increase to the increase in the number of deployed applications [Zachman 1997]. In addition, enterprise modeling takes too long and becomes outdated too soon [Fowler 1997]. The main causes of the problems are the lack of efficient solutions to manage changes accumulated across an enterprise; the lack of an efficient and precise way to describe, analyze, and communicate the architecture; architectural mismatch; poor abstraction; and poor support for legacy, component-based and multi-paradigm systems. The main constraints are the amount and nature of change on the enterprise level [DeBoever 1997, Mowbray 1997], and the necessity to reuse the existing information infrastructure [DeBoever 1997, Fowler 1997].

Clearly, the main goal of an enterprise, which must be supported by constituent applications, as any other informational construction, is to satisfy its functional and non-functional requirements. For an enterprise, the former is the business work-flow it is to support. Today, business work-flow changes more and more rapidly. The rate of change has grown from a full cycle period of approximately 7 years in the 1970s and 1980s to 12-18 months in the 1990s [DeBoever 1997]. Essentially, the non-functional goal is not only to align the enterprise with the business work-flow but also “to have such an enterprise that will allow quick re-aligning when the business work-flow changes” [DeBoever 1997]. Another important goal for an enterprise is to allow the gradual migration towards new technologies with the retirement of old ones as well as the evolution of systems comprising the enter-

prise. We define a *well constructed* IE as one that fully supports business work-flow and allows sufficiently quick re-alignment according to the work-flow changes while requiring only a reasonable amount of resources to maintain and manage the enterprise. In each case, the notion of *quick* and *reasonable* has to be determined.

Therefore, we suggest that the architecture of a system or a service functioning in the enterprise environment must aim to 1) reduce the amount of change associated with it and other systems, 2) reduce the cost associated with maintaining and re-aligning it and other systems, and 3) enable solutions that scale well with the increase in the number of deployed applications.

For instance, solutions currently available in the industry control access to application protected resources at several points, as Figure 2-6 shows. They are network (e.g. firewall), middleware, database and operating system controls. Making all these controls to work in concert and consistently enforce enterprise-wide access control policies is a daunting task, when there are hundreds of application and supporting systems (e.g. operating systems). Such solutions considerably increase the amount of change associated with administering authorization policies and applications, increase the maintenance and re-alignment costs, and do not scale well with the increase of the number of applications.

2.3.2 System Perspective

The main problem at the system level is that middleware AC mechanisms do not protect fine-grain resources or they provide limited capabilities for handling complex policies, which is required in some application domains, e.g. health care [Beznosov 1997, Wilson 1997]. In addition, there is a need for domain-specific factors (e.g. relationship between the

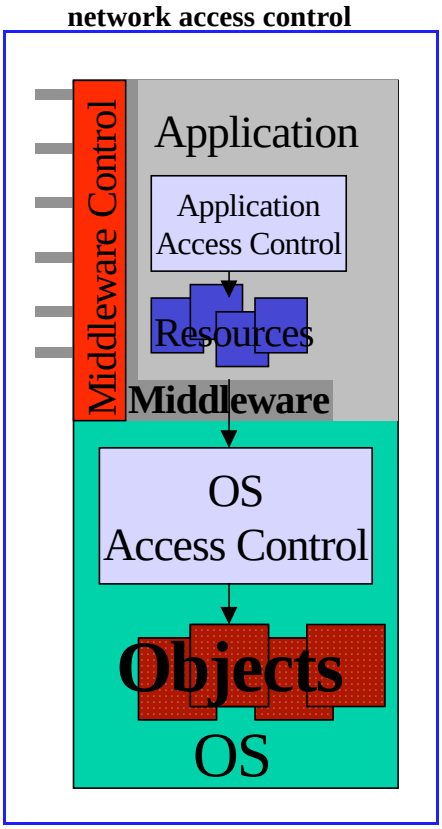


Figure 2-6. Points of Access Control

user and the patient [Barkley 1999], emergency context) to be used in access control policies. This complexity and granularity level often force application designers to embed domain-specific authorization logic inside their applications. Some even document patterns of designing "application security" [Yoder 1997]. As a result, this increases the complexity of software design and makes it difficult to ensure system integrity and quality. It also significantly increases the difficulty and cost of system administration and management.

2.3.3 Problems with Access Control in a Health Care Enterprise

In the previous sections, we outlined general problems that architectures of information enterprises face, suggested what architectural properties enterprise systems and services should have. We also showed what functional requirements mechanisms controlling

access to application resources should have. Now we substantiate our discussion with particular problems in AC for the information enterprise of a health care organization -- Baptist Health Systems of South Florida (BHS). BHS is the largest non-for-profit healthcare organization in South Florida, which is comprised of six major hospitals and clinics. Due to the technical and historical reasons BHS information enterprise is referred as Computerized Patient Record (CPR). We will use this name through out the section. Parts of the section are based on the materials from [Beznosov 1998a, Beznosov 1998b].

2.3.3.1 Introduction

CPR enterprise is and will be a heterogeneous environment for long time if not forever. Legacy computing technologies and architectures, such as stovepipe systems [Mowbray 1995], are going to co-exist with new component-based systems as well as with new middleware and other technologies such as CORBA and its services, common and vertical domain facilities. The enterprise will always have to accommodate emerging technologies with old disappearing ones. The main goal for CPR security architecture is to provide a security environment where the view of an enterprise user will be consistent across all its components, and AC decisions will be made according to one set of enterprise-specific policies. We list the main issues that make this goal difficult to achieve and maintain. We present our vision on how a CPR enterprise architecture can be designed so that the described problems can be addressed in the realm of existing constraints. The problems discussed here are based on the experience from the ongoing project of designing CPR security architecture at BHS.[1]

1. More information about BHS can be found at htttp://www.baptisthealth.net

In order to facilitate understanding of the issues and constraints, we first provide background information on CPR enterprise and describe its specifics next.

2.3.3.2 CPR Enterprise

CPR is a long-term initiative at BHS. Wreder et al. [Wreder 1998] describe its ultimate goal as to provide the mechanism to capture, manage and present information required throughout the continuum of care in a manner that optimizes the business process by taking advantage of distributed object computing technologies. BHS's CPR can be viewed as a set of object services and clients distributed across a healthcare enterprise. Since all clinical and some business services are eventually expected to be integrated into the CPR infrastructure, it is considered as an enterprise itself. CPR architecture is being constructed utilizing the Object Management Architecture described in [Soley 1996]. CORBA-compliant ORBs constitute a distribution backbone for CPR components.

All deployed application systems are selected according to the criteria of the best fit for a particular business process they serve and according to the mandatory requirement to comply with CPR architecture. Particularly, application systems and services are required to provide CORBA-compliant interfaces to their main functionality and to use services available within CPR enterprise to avoid redundancy. For example, any application system and service that has a notion of patient is required to utilize a CORBA-compliant Patient Identification Service (PIDS) [OMG 1998b] and expose any data related to clinical observations via interfaces compliant with Clinical Observation Access Service (COAS) [OMG 1997] standard from the OMG. The very first CORBA-based CPR service was deployed at BHS in February 1998. The service provides access to clinical transcription records. BHS

is in the process of deploying a Master Patient Index service that will provide PIDS among other services. An anatomic pathology system that will be using PIDS and will also provide access to its data via COAS-compliant interfaces is expected to be deployed as well.

Even though all new components deployed in CPR enterprise are based on CORBA technology, there are legacy systems that have to be integrated in CPR architecture at some point. Also, some new non-CORBA-compliant services will be deployed within CPR enterprise. Such systems and services have to be integrated in CPR enterprise including its security infrastructure. We will discuss the issues of designing CPR security architecture in the next sections.

CPR enterprise has its own features that affect its architecture. Some are common to any enterprise, some are specific to health care, and others are BHS distinctive. They serve as constraints to the enterprise, including its security infrastructure. We identified the following significant features:

- As we discussed in Section 2.3.1, like with any other IE, due to the increasing rate of information enterprises growth and the replacement of conventional monolithic, multi-purpose solutions by component-based specialized ones, the amount of maintenance and administration is rapidly increasing. The increase of enterprise size and complexity exacerbates all other factors.
- CPR business processes change much faster than they used to do. This forces CPR enterprise configuration to be adjusted at the same rate. For a security architecture, this means decentralized administration and extensive delegation of administration privileges, as well as more frequent AC policy changes driven by business processes.

- Many different application systems are used across the enterprise. Y2K inventory revealed about 200 different applications from word processor to multi-million dollar clinical systems.
- Some products come from narrow niches with few vendors, which eliminates fair competition of products forcing customers to select sub-optimal solutions or contract conditions.
- Heterogeneous operating system (OS) environments serve different needs of different departments. The heterogeneity is also due to tight coupling between applications and the underlying OS. Major clinical applications are available only on particular OS or even hardware platforms.
- Vendors are oriented towards numerous more conservative customers. Those customers are usually technically less educated and, as a consequence, concerned only with the functional properties of the products.
- Outside visitors have the potential for physical access to desktops and network infrastructure. Unlike financial or manufacturing enterprises, in health care organizations patients, and often their guests, have access to most facilities thus making it almost impossible to introduce the notion of trust boundary dividing the facilities.
- Different departments have different levels of urgency and different requirements for confidentiality and service availability. This makes security protection of the same high strength unjustifiable and some times even conflicting with the support of business practice specific to the particular department.

- The information technology (IT) department cannot afford in-house development due to the lack of resources and qualified staff, which mandates the use of COTS applications, external consulting, integration and outsourcing services.

2.3.3.3 Security Architecture Issues

Given the general constraints imposed on CPR, we discuss the issues related directly to its security infrastructure, and first present four groups of issues related to the CPR security architecture. We limit our discussion to the issues only directly related to access control. Its full version can be found in [Beznosov 1998a, Beznosov 1998b]. To ease the understanding of how the groups relate to each other, we place them on a discrete 2-dimensional space depicted in Figure 2-7. The horizontal dimension identifies if the issue can be

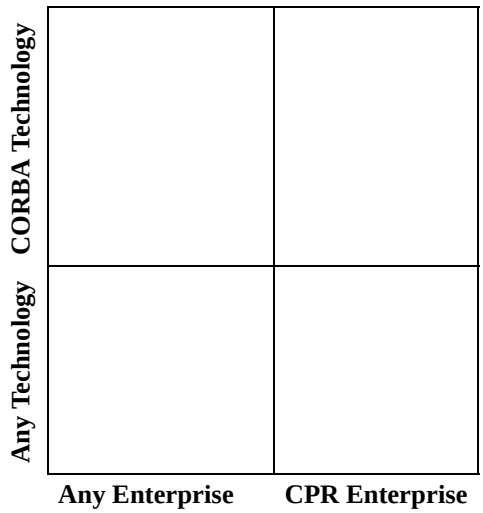


Figure 2-7. CPR Security Issues Space

found generally in any information enterprise or only in a CPR enterprise. The vertical dimension identifies if the issue is related to any technology or it is specific to CORBA-based enterprises.

General issues are propagated into more specialized areas. For example, those problems that exist in any information enterprise are propagated also into a CPR enterprise. To illustrate it, we present the same issue space in the propagation pyramid shown on Figure 2-8. More general problems and requirements at the foundation of the pyramid, if not

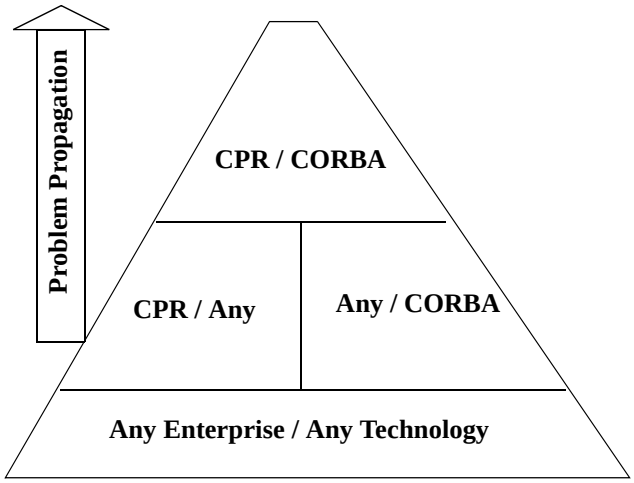


Figure 2-8. Propagation of Problems

addressed, would propagate to the upper layers. We discuss the problems in groups moving from generic to specific.

Any Enterprise Based on Any Distributed Computing Technology

Coupled AC logic. Conventional applications have their own AC decision logic tightly coupled with an application itself. Enterprise security administrators end up having to configure such logic on application-by-application basis, which brings tremendous administration overhead and highly increases chances of human error.

Decisions about which users can have what access to what assets of the information enterprise should ideally depend only on the following factors: user privilege attributes,

enterprise security policies, business workflow and its constraints. All these items are properties of a particular enterprise and not of any application in it. Also, AC models must have a common denominator to map enterprise security policies and business workflow constraints uniformly into particular authorization rules. Therefore, all access decisions should be foreign to an application service and native to the enterprise security infrastructure as well as the enterprise business workflow.

No standard administration interface. Among those applications that have their own AC mechanisms, each has its own proprietary interface to administrate the mechanism. This makes it impossible to administrate AC and other security mechanisms for multiple applications using a single administration environment. Therefore, the automation of AC administration is a very resource-consuming and error-prone task.

Inconsistent AC models. Due to the replication of security information over applications and coupling of authorization and application logic, multiple inconsistent AC models co-exist in the same information enterprise. In this case, it is highly difficult to insure consistency of AC rules across the enterprise. Most of the time, security administrators end up having no guarantee, whatsoever, that authorization rules and, especially, changes to them are consistent across all application systems and comply with organizational policies.

CPR Enterprise Based on Any Distributed Computing Technology

YES/NO AC. It is hard to draw exact borders between what a care giver, as a user of medical systems, is supposed to have access to and what he/she is not. Some scenarios are clear (e.g., a registration clerk trying to change lab test results of a patient) and some are not (e.g., emergency room physician browsing encounter history of a patient). There is a

need for, so called, “soft AC” when a principal is granted access; however, audit and (maybe even) non-repudiation “alarms” go off for later investigation. Meanwhile, the user is warned that they are accessing information they are not supposed to. Such “soft AC” notion is missing from most models including CORBA Security. Additional abstraction is needed in security administration solutions to accommodate “soft AC.”

Vanilla security administration. A low-level generic security administration model, where authorization (and other) rules are expressed in terms of security attributes of subjects and (maybe groups of) objects/interfaces, does not support needed abstractions specific to the business process. Domain-specific AC languages that abstract the access model to the level of business model are necessary.

CPR Enterprise Based on CORBA Technology

Heavy security policy domains. Ideally, the notion of security policy domains should be used actively in order to leverage AC mechanisms of CORBA Security service. All information about a patient can be represented as a collection of objects that belongs to the same AC policy domain. When a new patient is registered and his/her record is created, all data about the patient is accumulated in the objects belonging to the patient’s domain and AC (as well as other security) policies are instantiated appropriately. Consider a common situation when a healthcare enterprise serves thousands of patients. We do not have empirical knowledge but it seems that the current security technologies (e.g. SESAME and Kerberos) used for CORBA Security implementation would not scale to scenarios with thousands of security policy domains.

Coarse-grain AC. Preliminary modeling of a CPR AC [Beznosov 1997, Wilson 1997] shows that the basic CORBA Security AC model does not take into account such important for a healthcare enterprise factors of authorization decisions as the content of requests and replies, and the context of client/server interactions.

2.3.3.4 Goals for CPR Architecture

Not all the issues are as urgent in the short term period or as important in the long term period as others. Some of them are highly critical for CPR enterprise success. Below, we state the goals that we believe will impact significantly the way CPR enterprise security architecture will evolve.

Long Term Most Important Goals

Achieving long term goals will enable integration of applications with CPR security infrastructure. The goals are difficult to implement quickly because they require re-structuring of the infrastructure and re-design of the applications. However, once realized they will enable creation of a *well constructed* CPR enterprise which will support organizational work-flow and its changes. Besides generic goals, CPR security infrastructure has several specific long term objectives in the area of AC.

Enterprise-wide logically single repository of user security attributes is paramount to any well structured organizational security infrastructure. It will provide a single view of a user no matter what underlying security technology and applications are used. When a user initializes a session, information from the repository is used for their identification and authentication. The main advantage is, however, the existence of logically single location for security information related to the user which allows inherent coherence of any changes

to it. Such a repository will allow significant reduction of enterprise-wide user security information administration.

Second goal is the realization of fine-grain uniform access controls across all applications. Otherwise complete CPR automation will risk a health care organization to face liabilities of various degree. For instance, breaching the regulations on patient information privacy and confidentiality [DHHS 1999], which are part of health insurance portability and accountability act (HIPAA) [USA 1996] imposed on US health care industry, would jeopardize the company ability to compete on the market and could bring legal actions against its administration. The keys here are the granularity and uniformity of AC. Without needed granularity, service-based health care applications would not provide protection necessary for controlling access on the need-to-know basis. On the other hand a lack of uniformity would introduce inconsistencies in AC enforcement thus considerably decreasing its usability, manageability, and maintainability.

Another important goal is the use of domain-specific high-level abstraction for administering security in general and AC in particular. We describe below those factors that should be used to make elaborate authorization decisions in order to comply with patient information discloser requirements.

Affiliation -- what subsidiary of the health care system a particular care giver works for or is a partner with. Due to frequent mergers and to the fact that many physicians consult in several hospitals, this factor affects authorization decisions.

Role -- what role the user is assigned to in the current session. This factor is important to use because the same user can act in different roles performing his or her responsibilities and because RBAC decreases security administration overhead. However, modeling of health care AC policies shows us that the type of relationship between the user and the patient is also used very extensively in making authorization decisions.

Relationship -- what is the relationship between the user and the patient whose records are to be accessed. Today health care practise increasingly employs shared care approach in which the patient is managed by a team of care professionals each specializing in one aspect of care [Grimson 2000]. Some types of relationships that need to be managed in the healthcare context are: patient's primary care provider; admitting, attending, referring, or consulting physician of a particular patient; part of the patient care team; healthcare staff explicitly assigned to take care of the patient; patient's immediate family; patient's legal counsel or guard; personal pastoral care provider.

Location -- where the user is accessing information services from. Location information is used in several types of authorization policies. One type is represented by the following example of an AC policy: a nurse should have access to medical records of a patient if the nurse is currently working on the same “floor” as the patient. Another type uses location to identify the trust domain where the user is accessing information services from. A reasonable policy would deny access to any sensitive information for anyone accessing it from untrusted areas or via unprotected communication channels. Location can also be used to derive the emergency level of access. A policy can allow read access to all patient

information of all patients for any user assigned to the role physician and accessing the information from an emergency room.

Time -- when access is requested. The time factor is useful for authorization rules on users assigned to shift-related positions such as nurses and for task-based AC [Thomas 1994] when access to patient records is granted for the task duration to the users responsible for accomplishing it.

All this information is essential in order to make authorization decisions at health care enterprises. To achieve integral use of the described factors, an effective domain-specific authorization language that would incorporate the concepts of role, affiliation, location, relationship and time is needed.

Putting all these goals together, we believe that if the security infrastructure of a health care enterprise can be designed in such a way that AC can be enforced at fine level of granularity, in a uniform way across the enterprise, and a domain-specific high-level authorization language is used, then CPR security infrastructure can be well structured.

We used CPR security architecture at BHS as a concrete example for illustrating the context in which the problem of engineering access control in distributed applications is stated. Familiarity with the context will help to the understanding of the problem requirements and its solution proposed in this dissertation. We outlined the main issues in constructing CPR security architecture. In addition, we grouped them into four categories according to the type of information enterprise (general or healthcare) they can appear in, and the type of distributed computing technology they characterize (any or CORBA-spe-

cific). Finally, we defined most important long and short term goals for CPR security infrastructure.

2.3.4 Summary

The issues central to the problem of controlling access to the resources of distributed enterprise applications are of two types: functional and architectural. The functional are 1) the granularity of protected resources, 2) the enforcement of policies specific to the business domain of the enterprise, and 3) the decisions based on elaborate security-related and unrelated information about the accessing subject, the access operation and the object. Last but not least, ways must be provided to ensure the consistency of policy enforcement across multiple applications.

Architecturally attractive solutions must effectively support the evolution of enterprise systems, i.e. changes to existing applications, their insertion or deletion, changes in business processes and security policies, changes in hardware/software platforms, etc. These qualities need to be achieved at reasonable cost during the development, operation, and evolution of application systems and the enterprise they comprise. Above all, the solution shall scale well with the number of applications.

2.4 Evaluation Criteria

Before proposing our solution to the problem, we will review the state of the practise and research in the next chapter. In this section, we define a framework containing criteria for evaluating the existing technologies and related work, as well as for analyzing our approach.

Any existing or proposed solution should be evaluated on the basis of its adequacy in addressing the problem. Therefore, the problem statement is the main source for the criteria. Particularly, how well does it address the following main issues?

1. **Granularity of protected resources.** If a technology or solution does not allow authorization decisions on fine-grain resources, then it cannot be used for protecting application resources. We will use the following granularity hierarchy: application, interface, method, arbitrary resource.

2. **Support for policies specific to the organization application domain.** There is a wide range of supported AC models and policies, as it will be shown in Chapter 3. At one end there are AC mechanisms that support only one model (and the corresponding policies), for example lattice-based mandatory AC (MAC) [Bell 1975]. At the other end are solutions that allow implementation of any authorization logic and their support for policies is limited only by the interface to the logic. In general, the more AC policy types a mechanism can support the easier it is to configure for required organizational policies. When applying this criterion we will look at the range of supported AC models.

3. **The variety of information available for making authorization decisions.** As we discussed above, authorization decisions are made by evaluating rules with the use of information about the subject and object, as well as the operations to be performed by the former on the latter. The available information is limited. For example some technologies allow obtaining only authenticated identity of the subject but not the informa-

tion about group membership or activated roles, which ultimately limits the functional capabilities of the AC mechanism based on such a technology. We will look into what information is available and what information is used in authorization decisions.

4. **The use of application-specific information.** The use of information which is application-specific and becomes available only while the application processes the client request is critical for some application domains (e.g. health care). If a solution does not allow the use of such information, then full automation of protecting application resources would not be possible.

5. **Support for consistency of policies across multiple applications.** It was discussed earlier that in the enterprise environment, the issue of consistent policy enforcement is a critical one. We will consider the support for enterprise-wide consistent AC policy enforcement while examining the available and proposed approaches.

6. **Support for insertion and deletion of applications, changes in policies and the computing environment.** No matter how functionally perfect the support for the AC of application resources is, if it is highly ineffective to accommodate all these changes, then it is of no good in enterprise settings. Most available approaches support the changes to some degree. We will evaluate how good the support is. Unfortunately, there are not any objective quantitative criteria for determining the level of support. This is why we compare the solutions with each other in regards to this criterion.

7. **Solution scalability.** Performance and administration scalability highly affects the approach utility. Regardless of all other merits, if an approach does not scale well it can not be more then just an academic exercise. Since there is not any benchmark available

for evaluating the scalability of AC solutions, we will use common knowledge to reason about the scalability. For instance, when it is possible, we will examine the amount of data that needs to be modified, in order to accommodate a policy change. Another commonly known measure that we will use is the communication complexity, which is still regarded as the major factor in the performance of distributed systems.

3 Related Work

This chapter provides a survey and analyses about available solutions in the area of this dissertation, i.e. controlling access to the resources of distributed enterprise applications. In sections 3.1 and 3.2, we survey the existing work in detail, then summarize the discussion in Section 3.3.

The idea of treating authorization logic as an independent component of software systems is not new. An abstract model of a reference monitor [Anderson 1972] is a classical example of authorization decisions being made and enforced outside of applications. The concept has being employed in the AC design of operating systems from the early days of computer security. Most operating systems implement authorization logic in the security part of their kernels [Benantar 1996, Curry 1992, DEC 1989, Gligor 1986, Grampp 1984, Heydon 1994, Hommes 1990, Karger 1991, Luckenbaugh 1986, McCauley 1979, McInerney 1999, Mullender 1990, Pfleeger 1989, Quarterman 1985, Saltzer 1974, Walker 1980].

Among special-purpose ad-on security software packages, Computer Associates' Access Control Facility 2 (CA-ACF2) [CA 1998a] and CA-Top Secret [CA 1998b], as well as IBM's Resource Access Control Facility (RACF) [Benantar 1996, IBM 1976] are the most known ones. RACF is a security system for MVS and VM operating systems. It acts as a central control point that mediates access to various system resources by authenticated users. The operating system's resource managers send user requests to RACF for valida-

tion. Computer Associates' packages are integrated in operating systems and work in a way similar to RACF. As a matter of fact, MVS installations have the option to use CA-Top Secret or CA-ACF2 as their choice of access control software package [Benantar 1996], which underlines the separation of AC mechanisms from application and even operating system functions.

3.1 Access Control for Distributed Applications: State of the Practice

In this section, we review the capabilities and discuss what the main-stream technologies provide for engineering of AC in distributed software applications. We evaluate their fitness by applying the criteria described in Section 2.4. Ideally, all security functionality should be engineered outside of an application system, therefore making it, so called, "security unaware." This is why we also examine if the distributed security technologies can enforce AC externally to the application.

In general, there are two types of technologies used for securing distributed software systems. One type is the technologies that merely provide party authentication, communication protection, and AC independently of the underlying communication layers. They are Kerberos [IETF 1993, Neuman 1994a], GAA API [Ryutov 2000a], and SESAME [Kaijser 1998, Parker 1995]. Application developers deliver inter-application communications by other means (e.g. ONC RPC [Bloomer 1992]). This enables the use and mix of any desired communication protocols and media. However, developers are overburdened with the efforts to integrate security with the underlying communication technology.

Another type is middleware technologies, such as CORBA [OMG 1996b], DCE [Gittler 1995], Java [Lai 1999], and DCOM [Microsoft 1998], that provide an underlying communication infrastructure along with the security subsystem, thus enjoying reasonable integration of both and much more seamless use of the former by developers. Moreover, some of them (CORBA and DCOM) enable basic AC completely outside of an application system because access decision and enforcement occur before the remote call is dispatched to the application.

3.1.1 Java Authentication and Authorization Service

The release of the Java 2 platform introduced a new security architecture [Gong 1997], which uses a security policy to decide about granting access permissions to the running code. It uses factors relevant to the code for authorization decisions, such as where the code is coming from and whether it is digitally signed and, if so, by whom. Such a code-source-centric style of AC is very different from user-centric authorization policies supported by conventional computing environments. Java has recently become widely used in enterprise application systems where different users run the same code. The Java Authentication and Authorization Service (JAAS) [Lai 1999] is designed to provide a framework and standard programming interface for authenticating users and for assigning privileges to users. Using JAAS together with Java 2, an application can provide code-source-centric, user-centric, or a combination of both types of authorization.

In Java 2 and JAAS (we will refer to the combination as “JAAS”), AC is enforced by the security subsystem only on Java Virtual Machine (JVM) protected resources, such as files, sockets, etc. Java objects or other application resources are not protected, so AC has

to be implemented by an application itself. Application developers can program against the same authorization API as the one used for the rest of Java 2 run-time if they employ JAAS authorization.

For an application to use the JAAS authorization mechanism, it needs to 1) construct an instance of class `java.security.Permission` representing the protected resource(s) in question, 2) locate global instance of `Policy` object, 3) obtain permissions granted to the code and the subject via `Policy::getPermissions()`,[1] and 4) determine if the returned collection of granted permissions contains the required one.

JAAS supports any level of resource granularity because it specifies a flexible mechanism for defining application-specific protected resources. This is done via access rights which are *permissions* in the terminology of Java security architecture. Java permissions are classes with the common ancestor `java.security.Permission`. Depending on the semantics of a permission, a group of resources could be associated with it. For example `java.io.SocketPermission` is associated with all port numbers in the example policy in Figure 3-1. There are several pre-defined permissions. They are `file,`

```
//JAAS principal-based policy
grant
      Codebase "http://bar.com",
      Signedby "bar",
      Principal bar.Principal "duke"
      {
            permission java.io.FilePermission "/cdrom/duke", "read";
            permission java.io.SocketPermission "*", "connect";
      }
```

Figure 3-1. Example of JAAS Policy Entry (adopted from [Lai 1999])

1. An application can obtain permissions for processing a client request only once as long as the subject privilege attributes, code base, and code signer do not change.

`socket`, `property`, `runtime`, `AWT`, `net`, `reflect`, `serializable`, and `security`. New subclasses of `Permission` can be defined in order to implement new types of permissions, including those which are application-specific.

JAAS model defines a generic concept of authorization engine via abstract class `java.security.Policy`, implementations of which are responsible for determining what permissions are granted to the code source executing on behalf of the given subject. The main method of this class, `getPermissions(subject,codesource)`, returns a collection of permissions granted to the subject with privilege attributes presented in argument `subject`, and executing code that came from `codesource`. A subclass of `Policy` can implement a different authorization policy, which should comply with the class definition. Therefore, the main constraint on such an implementation would be the syntax of `getPermissions()` method. JAAS provides a default subclass `Policy-File`, which supports authorization decisions according to the source code base, the identity of the code signer, and the value of privilege attributes possessed by the subject. These all are used to determine permissions for a particular resource. The flexibility of JAAS comes from the property that the authorization logic can be implemented in various ways without deviating from the JAAS AC model. Since `Policy` is an abstract class and its main method `getPermissions()` could be implemented in many different ways, JAAS does not constrain implementers to any particular authorization model, which enables support for policies specific to the organization or to the application domain. It is up to the implementers of `Policy` instances to achieve performance and administration scalability.

JAAS has a generic and extensible support for different authorization factors. Privilege attributes are not limited to the predefined ones. New attributes can be easily defined via new Java classes. Moreover, JAAS supports the composition of privilege attributes into hierarchies, which is important for implementing AC models with relationships between attributes, for example role-based AC (RBAC) with role hierarchies [Sandhu 1996]. On the other hand, even semantically the same attributes, if they are implemented as different classes, are considered dissimilar by JAAS, which introduces a basis for confusion. A notion of an attribute type, as in CORBA or SESAME, would sufficiently address the problem.

JAAS architecture does not explicitly support the consistency of authorization decisions across multiple applications because `Policy` instances used for authorization decisions must be local to the application. However, this does not preclude an implementation of `Policy` to delegate authorization decisions to a remote service.

JAAS architecture is relatively adaptable to the changes in applications, authorization policies, and computing environment. Changes to the policies can be accommodated via the replacement of the `Policy` object. Java's dynamic loading mechanism allows the addition and removal of applications as well as adaptation to various changes in the computing environment.

Because JAAS architecture is defined as a set of several Java abstract classes and interfaces, allowing the implementations of very different scalability, we can only analyze the scalability constrained by the interfaces to its components. We found that the semantics of `Policy::getPermissions()`, which returns the amount of data proportional to the

number of all permissions in the system granted to the subject, can cause performance scalability problems for policies of some types. Consider for example an implementation of `Policy`, which supports, or maps to, an owner-based discretionary AC (DAC) policy [NCSC 1987], similar to UNIX file permissions. In such systems, there is a set of permission bits declaring access rights for the owner, the members of the primary and other groups of each file. Thus `getPermissions()` should return permissions to all files that the user has. This is justified only when an application needs to make many authorization decisions for the same subject running the same code. However, when only one authorization decision is needed in order to process a request or when the code base or signer change, returning all permissions granted to the subject seems inefficient. A more scalable solution would be a one in which the result of the authorization decision is returned instead.

3.1.2 Distributed Computing Environment

The Open Software Foundation's (OSF) Distributed Computing Environment (DCE) [Kong 1995], is an underlying RPC infrastructure and a collection of integrated services that support the distribution of applications on multiple machines. The functionalities provided by DCE security services include: user authentication, secure data communications to protect data communicated by an application to other applications over DCE infrastructure, and authorization for applications [Gittler 1995].

DCE Security is based on Kerberos [IETF 1993, Neuman 1994a], which performs authentication of users and applications based on cryptographic keys so that communicating parties can trust the identity of each other. DCE augments Kerberos with a way to trans-

fer additional privilege attributes to a server that may choose to perform AC based on those attributes.

The service does not control access to applications or their resources, and DCE applications are expected to enforce and provide administrative access to authorization policies on their own. To do so, an application has to implement AC functionality, including an access control list (ACL) manager and an ACL storage, as shown in Figure 3-2. In order

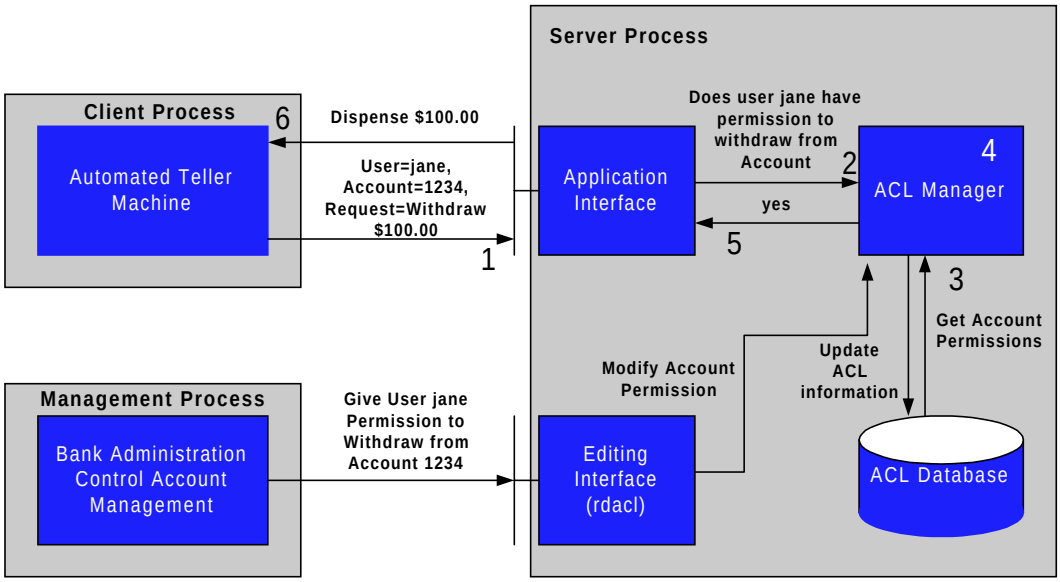


Figure 3-2. Authorization Process and ACL Management in DCE-based Application Systems (from [Caswell 1995])

for an application to use DCE security service for AC, it needs 1) to determine the DCE object ID (OID) of the resource in question, and 2) to obtain authorization decision from its ACL manager using the OID.

If an application uses the DCE ACL model for authorization, it associates an ACL with a protected resource via OIDs, which are used by the ACL manager to determine right ACLs. The exact definition of “resource” is entirely at the discretion of the application. For

example, an object could be an item of stored data (such as a file), or could be a purely computational operation (such as matrix inversion). Thus, the concept of OIDs enables any granularity of protected resources.

DCE ACLs support a limited number of privilege attribute types -- only identities of the user, who is the resource owner, the owner group, and other group(s). There are also distinctions between:

- “local” and “foreign” (from another DCE cell) subjects,
- those acting as delegates and primary invokers, and
- entries that specify specific and default policy, i.e. in the absence of any other applicable ACL entry (ACLE).

DCE ACL language is also considerably limited allowing security administrators to either explicitly grant or deny rights to the subject based only on its identity or group attributes. The language capability to support policies specific to application or organization remains to be seen.

The following simple example from [Caswell 1995] demonstrates (Figure 3-2) how AC is expected to be implemented by an application system. User `jane` makes a request to withdraw $100.00 from her account number `1234` (step 1). The application interface passes this information to the ACL manager asking for an authorization decision (step 2). The ACL manager retrieves the authorization policy for account `1234` from the ACL database (3) and applies the policy to derive the answer (4). If user `jane` is authorized, the withdrawal is performed (5 and 6).

In order for an ACL associated with application resources to be administered, DCE applications are expected to provide a means for it. They can implement DCE standard ACL administration interface (`rdacl`). When Jane's account is first set up, a bank employee would use an administrative tool to give user `jane` the permission to withdraw money from account `1234`. The editing interface enables the ACL manager to change the policy. An ACL manager changes a policy by retrieving the current policy, modifying it, and writing it back to the ACL database. `Rdacl` interface seems to be the only means of ensuring the consistency of authorization policies across application boundaries unless access to the ACL database is implemented as a global service. In the latter case, policy and application changes could also be accommodated by the DCE environment easier than in the basic configuration shown in Figure 3-2.

As seen from the discussion above, DCE security service provides rudimentary help to applications to make AC decisions, and it enforces no AC externally to an application. Comparatively to its predecessor, Kerberos, it advances privilege attribute management by enabling attribute types other than subject identity in EPACs. However, the expressiveness of DCE ACL language is fairly limited, and we could not determine how application-specific factors could be used in authorization decisions if the mechanisms of DCE ACLs are utilized. It seems that the increase of the application client or server population would not drastically affect overall DCE-based enterprise performance because AC decisions are made using local data. However, administration scalability is poor because policy changes have to be reflected in the ACL database of every application unless the database is centralized. Then the performance scalability would suffer.

3.1.3 Microsoft Distributed Component Object Model

The Distributed Component Object Model (DCOM) [Grimes 1997, Rubin 1999] is a middleware technology from Microsoft, which extends the Component Object Model (COM) to support communication among COM objects on different computers running some flavor of MS Windows OS. A schematic representation of DCOM middleware is shown in Figure 3-3. DCOM protocol, known as "Object RPC" or ORPC, extends the stan-

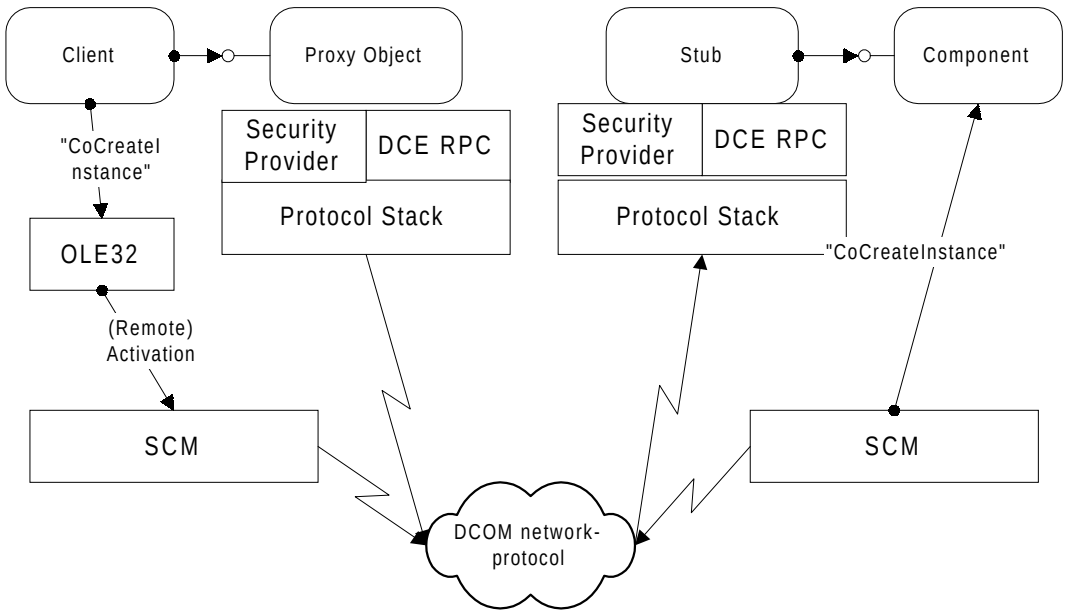


Figure 3-3. DCOM Middleware (from [Microsoft 1998])

dard DCE RPC protocol. At the wire level, ORPC uses standard DCE RPC packets, with additional DCOM-specific information.

Since DCOM RPC is a derivation of DCE RPC, it is not surprising that its security model resembles DCE security. ACLs, with the language similar to the one in DCE, are used to code authorization policies. In DCOM, they are named Discretionary ACLs

(DACL) to signify the default right of the object owner to modify DACL entries. DACLs can be configured using DCOMCNFG configuration tool or programmatically using the Windows NT registry and Win32 security functions. However, these do not change the essence of the model. What does, though, is the capability of enforcing policies outside of a DCOM object, and the presence of a hierarchy of policies. This is a considerable advantage over the DCE AC model, where no control is enforced by the security environment, and an application has to implement its own.

DCOM provides two choices for controlling access to applications and their resources [Eddon 1999]. With “declarative security,” DCOM can enforce AC without any cooperation on behalf of the object or the object's caller; the policies for an application can be externally configured and enforced. The declarative security policies can be divided into default policies and component-specific ones. A default policy specifies the default launch and access settings for all components running on the local machine that do not override these settings. Component security settings can be used to provide security for a specific component, thereby overriding the default security settings.

With another, “programmatic security,” DCOM exposes its security infrastructure to the developer via security APIs[1] so that both clients and objects can enforce their own application-specific authorization policies in regards to resources of any granularity, and using any information as input for the decisions. Programmatic security can be used to override both default and component security settings in the registry. Figure 3-4 shows the hierarchy of DCOM authorization policies: 1) policy encoded in the behavior of the com-

1. For example, calling subject identity can be obtained using methods `IObjectContext::IsCallerInRole()` and `ISecurityProperty::GetCallerSID()`.

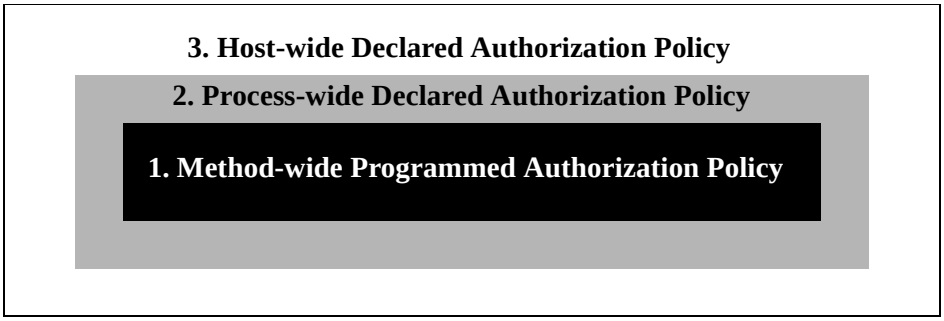


Figure 3-4. The Hierarchy of DCOM Authorization Policies and their Scope

ponent implementation, 2) the declarative process-specific, and 3) the declarative host-specific policies. Policies 2 and 3 are enforced before the call is dispatched to the object method. In this hierarchy, the inner policies override the outer ones in the following way: before the invocation reaches the method implementation, statements, if any, from process-wide policy override corresponding statements in the host-wide policy. If the invocation is allowed, then it will be dispatched to the method implementation, which will be able to exercise its own AC policy (policy 3), if any.

A significant hindrance to the authorization model is the granularity of "component-specific" declarative policy (policy type 2 shown in Figure 3-4). The granularity is per OS process, and there is no distinction among different object methods. That is, the policy uses the same DACL to control access to all objects and methods on those objects that a system process implements. However, if an application needs to have finer level of granularity and still use the DACL mechanism, it can achieve it, though with more effort and not transparently to the application logic. An application can associate its fine-grain resource with a protected resource of the operating system, such as an MS Windows registry key, assume the identity of the subject, and try to access the OS resource. If this access fails, the assumed subject did not have permission to access the resource.

Process-wide and host-wide policies (types 2 and 3) implicitly introduce the notion of access policy domains for DCOM objects. Unfortunately, the partitioning of objects can be only according to their locations and not according to their sensitivity or the value of other parameters. The limitation of authorization policy domains to the host boundaries restricts the administration scalability of DCOM-based distributed applications because it has to be performed individually on each host or even for each process.

As we have shown, no application-specific information can be used or application-specific policies are enforced when declarative AC is exercised. Declarative authorization policies and their changes have to be administered on a machine-by-machine basis, which hinders administration scalability and rules out automatic policy consistency across application boundaries unless applications are located on the same host.

3.1.4 SESAME

Secure European System for Applications in a Multi-vendor Environment (SESAME) is an European research and development project, which was started in late the 1980s. It is also the name of the technology that came out of that project. This technology [Kaijser 1998, Parker 1995] defines components of a security architecture providing the underlying bedrock upon which full managed security products[1] can be built using the following services defined by the architecture: authentication, authorization, confidentiality, integrity and audit.

1. Examples of such products are ICL's Access Manager [McMahon 1995] and Bull SA's Integrated System Management AccessMaster [BullSoft 1995].

The work of SESAME components (shown in Figure 3-5) could be described in the following way. The user logs in the SESAME environment by interacting with a user spon-

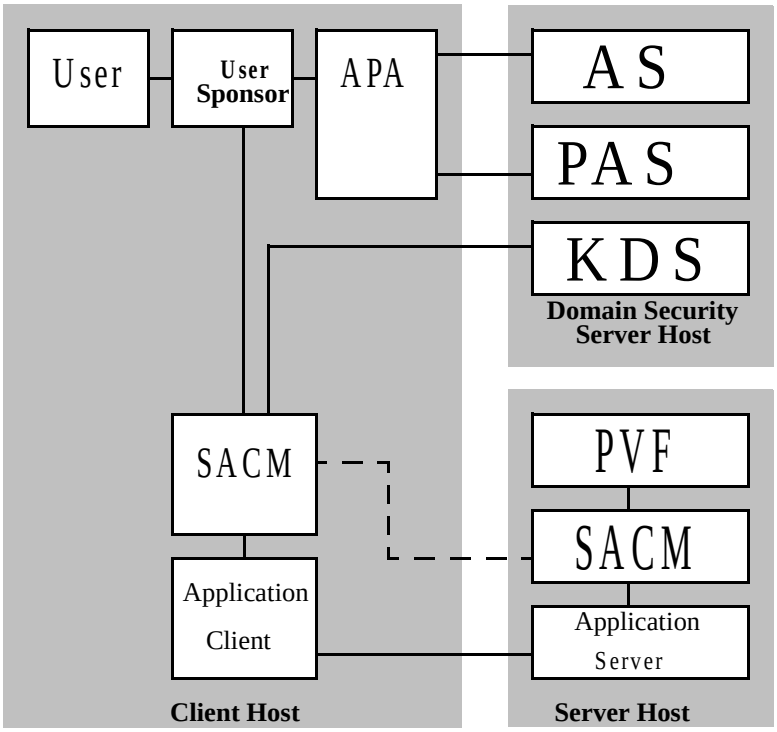


Figure 3-5. SESAME Components

sor (US) client, which then contacts the authentication server (AS) via the authentication privilege attribute (APA) client. The US authenticates itself to the AS, and then contacts the privilege attribute server (PAS) and receives from it a privilege attribute certificate (PAC) containing the subject privileges used for AC decisions. The user is now authenticated and has a PAC, which can be used when starting application clients. The PAC allows a user to access applications on a computer, which knows nothing about the user, but can verify the user privileges from the PAC. If the user wants to start an application, the US contacts the secure association context manager (SACM) for the application client. The client SACM then contacts the server SACM and they exchange subjects' credentials.

Next, server SACM contacts the PAC validation facility (PVF) to validate the subject's PAC. Finally, the user can start the application client and exchange data with the server.

A PAC can be used more than once at more than one target application. It is digitally signed to prevent it being undetectably tampered with. Privilege attributes can have any of the following syntax representations: access identity, role(s), primary group, and secondary group(s).

SESAME technology is not a middleware. Rather it is an architecture for security services. It does not provide a means for communication such as ORB bus in CORBA, or RPC layer in DCE or DCOM. Thus it cannot control pre/post invocation events. This is why AC and other security functionality has to be specifically invoked by an application system. This prevents SESAME from providing AC external to an application, as in DCOM "declarative security" or CORBA. On the other hand, authorization logic is provided to an application by SESAME-compliant infrastructure, as opposed to DCE where an application even has to implement ACL storage as well as run-time and management functionality.

Authorization decisions in SESAME are made by the SACM of the target application, which is responsible for receiving Generic Security Service (GSS) token and passing it to the SACM using GSS API [Linn 1997]. The target SACM passes the incoming security information to the PVF for analysis and validation. If all is valid, the SACM receives an integrity and confidentiality dialog keys from the PVF for protecting exchanges between the client and the target application. But if the PAC checks made by the PVF fail, the security context is not made available to the application.

Even if the checks succeed, besides routine checks of matching initiator and PAC identities, the SACM performs an additional AC check according to authorization rules [Parker 1995] represented as a set of access control entries (ACE) compliant with POSIX.6 [IEEE]. An entry can specify a certain application or “all applications,” to be accessible or not accessible by either an identity, a role, a group or “all initiators.”

The smallest unit of AC check in SESAME is an application system. Therefore, either access is granted to the whole system or any access is denied at all. For distributed applications, which commonly expose their functionality via several operations with different AC requirements, such a level of AC granularity is frequently insufficient. Consequently an application system has to implement additional functionality in order to exercise per-operation AC. Also, the architecture lacks the capability of applying one authorization policy to several applications thus requiring each application to be configured individually to support the policy.

The concept of domain in SESAME pertains to various authorities that manage keys, identities and privilege attributes. SESAME domains affect AC in the way that the same user can be granted different identity and privilege attributes in different domains, and the attributes can be mapped with restriction [Ashley 1997] thus influencing decisions made by the target SACM. Identity and privilege attribute domains make user security administration more scalable for large or multi-organizational environments.

The lack of external AC, coarse granularity of authorization decisions, and the need to administer the policies on application-by-application basis make SESAME less attractive then JAAS, DCE, DCOM or CORBA technologies for AC in enterprise distributed appli-

cations. However, SESAME can be deployed over most communication technologies, and is known for its advanced model of privilege attributes management and propagation, which is best suitable for large multi-domain heterogeneous environments [Ashley 1997]. These make it indispensable for building heterogeneous, multi-vendor, high-performance distributed application systems that require the use of different communication layers, and authorization based on privilege attributes other than user identity.

3.1.5 CORBA Security

The Common Object Request Broker Architecture (CORBA) technology, including CORBA Security Service, provides a general-purpose infrastructure for developing and deploying distributed object-based systems in a broad range of specialized application domains. All entities in the CORBA computing model are identified with interfaces defined in the OMG Interface Definition Language (IDL) [OMG 1999a]. A CORBA interface is a collection of three things: operations, attributes, and exceptions. An implementation of a CORBA interface is called a CORBA object. Hence, we use “CORBA object” or just “object” to mean “implementation of a CORBA interface,” where it does not cause confusion. Object functionality is exposed to other CORBA-based applications only through the corresponding interfaces. Objects have object references by which they can be referenced. An object reference is a handle through which one requests operations on the corresponding object.

3.1.5.1 Security Model Overview

CORBA Security (CS) standard [OMG 1996b] defines the following functionalities visible to application developers and security administrators: identification and authentica-

tion, authorization and AC, auditing, message integrity and confidentiality protection, authentication of clients and target objects, optional non-repudiation, administration of security policies and related information. One of CS objectives is to be totally unobtrusive to application developers. Security-unaware objects should be able to run securely on a secure ORB without any active involvement on the site of application objects. In the meantime, it must be possible for security-aware objects to exercise stricter security policies than the ones enforced by CS. In the CS model, all object invocations are mediated by the appropriate security functions in order to enforce various security policies such as AC.

Every user authenticates when he/she logs into the CS environment. The main result of authentication is a set of security-related data -- `Credentials`. The information in `Credentials` constitute the identity of the new subject, which initiates requests on CORBA objects on behalf of the user. Authenticated security attributes are part of the information stored in the `Credentials` object and are used for the purpose of enforcing various security policies. Because CS defines advanced concepts of privilege attributes, similar to SESAME, it enables AC policies based on roles, groups, clearance, and any other security-related attributes of subjects.

CS architecture achieves performance and administration scalability by the means of policies and policy domains, where any security policy is associated with a policy domain (or just “domain”). Policies of more than one type could be associated with the same domain and each object can belong to more than one policy domain. Domains could be organized in federations, hierarchies or be completely unrelated. AC decisions could be specific for each object, if the object is located in a separate domain, or a large group of

objects could be associated with one policy domain. This means that the model scales (in terms of performance as well as administration) very well without loosing fine granularity. Unlike DCOM, CORBA objects residing on different computers can be associated with the same domains.

As in DCOM Security, AC can be enforced completely outside of an application system because the enforcement occurs at the ORB level. Everything, including obtaining information necessary for making authorization decisions, is done before the method invocation is dispatched to the target object. As Figure 3-6 shows, policy enforcement code is

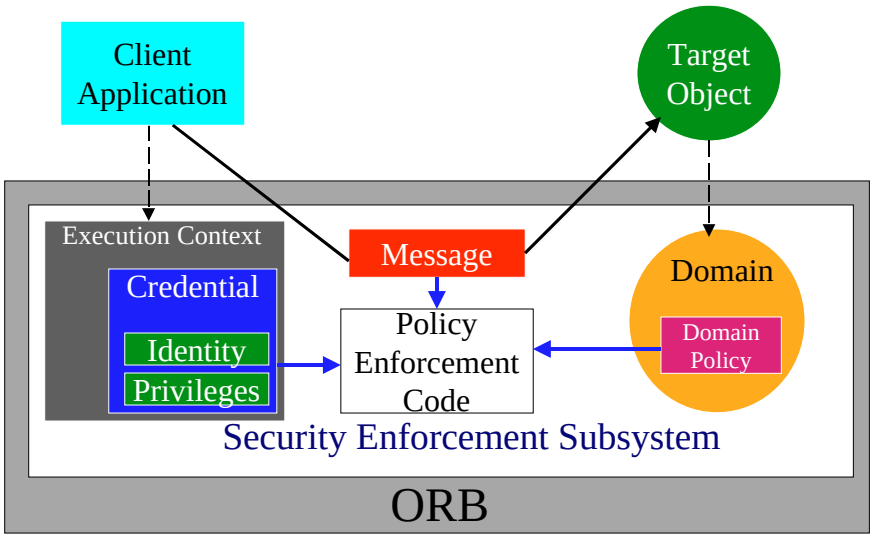


Figure 3-6. Enforcement of Policies in CORBA Security (from [Blakley 1999])

executed inside of CS enforcement sub-system, when a message from client application to a target object is passed through the ORB. Executed at the client ORB as well as at the target ORB, the enforcement code uses three sources of information for making decisions before it enforces them. First is the policy of the domain(s) to which the target belongs. Second is the information from credentials of the client. In case of AC policy enforcement, these are client privilege attributes (such as access identity, group membership, role and

clearance). The third source of information is the message itself which, in case of AC enforcement, is a request to invoke an operation on the target object.

CS controls access by clients to object methods. Objects, in their turn, are placed in AC policy domains, which allow the same policy to govern access to the methods of all the domain members. CS allows stating AC policies in terms of subject and object security attributes as well as operations implemented by those objects. Operations are grouped via rights required for invoking them. The rights granted to a subject according to its privilege attributes should match the required rights of the operation. AC policies control what subjects can invoke what operations on what objects in the domain the policies are defined on. The expressive power of CORBA AC mechanisms was analyzed by [Karjoth 1998], where it was shown to support lattice-based mandatory AC (MAC) [Bell 1975]. We discuss in greater detail the CS authorization model in Chapter 4. We also show there that it is possible to configure CORBA AC mechanisms to support role-based access control (RBAC) models, which means that DAC models can be also supported, as Sandhu and Munawer show in [Sandhu 1998a].

User grouping via privilege attributes, object grouping via policy domains, and method grouping via the concept of required rights enable high scalability of CS administration, which is an important factor in object-oriented enterprise distributed environments. Still there will be applications, in which additional AC has to be exercised (a so called *security-aware* application). A security-aware application can do so with the help of CORBA Security interfaces. For enforcing conventional AC policies, an application system needs to know who, wants to access, what protected resource, and in what way. CORBA Security

provides to an application a means to find out “who.” Interface `SecurityLevel1::Current`, available to an application, defines method `get_attributes()` for obtaining subject security attributes.

3.1.6 Generic Authorization and Access Control API

Generic Authorization and Access Control API (GAA API) is published as an IETF Internet draft authored by Ryutov and Neuman in [Ryutov 2000a]. It defines a framework for application authorization aiming to address the lack of standard authorization API for applications using GSS API. Kerberos [IETF 1993, Neuman 1994a] was the first security technology providing GSS API functionality, and it did influence the model behind GSS API. Kerberos had only rudimentary support for AC in networked applications: if a client did not have an authenticated ticket for a particular network server, then it could not establish a connection with it thus being denied access.

The GAA API model is based on the assumption that the distributed nature of Internet-based computing requires interactions between entities across autonomous and mutually-suspicious security domains. The authors also put in the front corner a requirement for a mechanism which provides authorization decisions on fine-grained resources for a wide range of systems.

The framework consists of two major parts: a programming interface for obtaining authorization decisions by application systems, and a “universal” language for AC policy representation, Extended ACL (EACL) [Ryutov 2000c], which is an extension of the traditional ACL model. The subject of our discussion is the API itself, which goals are 1) to support the needs of most applications, thus allowing application developers to refrain from

designing their own authorization mechanisms, 2) to allow better integration of multiple mechanisms with application servers (for example, GSS API [Linn 1993] and GAA API can be integrated to provide authentication of an invoking subject and authorization decisions).

GAA API does not enforce AC externally to an application. Instead, it provides authorization decisions which can be described as follows [Ryutov 2000a]. An authentication service performs authentication of users and supplies limited credentials, in the form of GAA API security context, to the application via authentication API, as shown in steps 1 and 2 in Figure 3-7. Then, the application calls GAA API routines (steps 3, 4, and 5) to

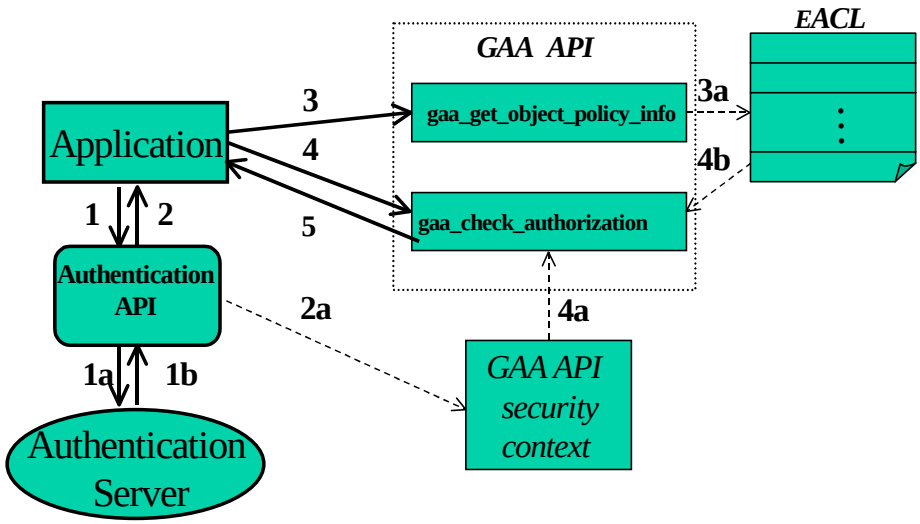


Figure 3-7. Sequence of Events in GAA API Model

check authorization against the policies. The API routines obtain subject identity from authenticated credentials of the client (step 4a) as well as policies (steps 3a and 4b) from local files, distributed authorization servers, or by some other means. They combine local and distributed authorization policies and information. For example, it is possible in the GAA API model to combine global EACL with the machine or application-specific list,

which enables the use of application-specific policies. The way the combination happens is not defined in [Ryutov 2000a] and depends on the concrete implementation of GAA API, which means that changes in the computing environment and the policies behind the programming API are not supported in a standard way and are implementation-specific.

For the purpose of our discussion, the most important API function is `gaa_check_authorization()`, which provides applications with authorization decisions, or indicates if additional checks are required, in regards to the requested operation(s). Its inputs are 1) a handle[1] to the data structure containing rules governing access to the resource in question, 2) security context containing privilege attributes of the accessing subject, 3) operations for authorization, and 4) parameters for a parameterized operation. The output consists of short, yes/no/maybe, and *detailed* answers. Specifying additional conditions which have to be met or time limits of the decision, the concept of detailed answer is unique to GAA API and provides capabilities required in many application domains. It is a data structure, used only when the short answer is "maybe," that contains a time window, during which the answer is valid, and a list of zero or more rights granted or denied to perform requested operations. Each right can be accompanied by the corresponding conditions, if any. Each condition is marked as evaluated or not evaluated. An evaluated condition could be also marked as met, not met or "further evaluation or enforcement is required." This tells the application which policies must be enforced.

The application must understand the conditions that are returned unevaluated, or it must reject the request from the client. If understood, the application checks the conditions

1. It is supposed to be obtained prior to the invocation via function gaa_get_object_policy_info() described below.

against the information about the request, the protected resource, or environmental conditions to determine whether the conditions are met. The enforcement of the returned conditions is up to the application. An example of condition enforcement is the use of CPU utilization. It could be specified in the policy that processing of the client request can be performed as long as the CPU is utilized less than 20%. Such a requirement could not be enforced by an authorization service. In the GAA API model, it would be passed to an application as a condition expected to be further enforced. Some other examples of conditions are printer load, provision of payment for access to the resource, and location of the subject [Ryutov 2000a]. As it can be seen, these authorization conditions give substantial flexibility for enforcing application-specific policies. Still, it remains to be seen if the concept will not cause tight semantic coupling of authorization service implementation with the application systems it serves.

The detailed answer may also mean that authorization is not completed yet, and additional privilege attributes are required. The application requires them from the client because GAA API attempts to build an authorization model that would fit into the existing, and, we believe, outdated[1] model of GSS API implemented first by Kerberos. It reuses Kerberos's authentication model, in which only authenticated subject identity is provided. This is why the GAA API model assumes group membership service, the definition of which it left beyond the scope of the model. A group server furnishing group membership information is the only way by which subject privilege attributes can be obtained. In order to do it, the client should request group (non)membership certificates from the server

1. We believe GSS API is outdated because some other architectures, such as SESAME, attempted to extend it. This indicates that the API does not satisfy the needs any more.

explicitly. The server is not part of the specification [Ryutov 2000a], although it was introduced earlier by Neuman in [Neuman 1993] and is described in Section 3.2.3.2 as part of the discussion of the authorization service from the University of Texas at Austin. Because the client is asked to provide group certificates after it already made an application request, it is possible to use application and even request-specific information for authorization decisions, which gives advantage to GAA API over other authorization solutions.

However, the use of a group server has significant drawbacks. First a communication scalability problem is created because some policies might require an undetermined number of interactions with the server causing possibly remote communications, which are usually expensive, unless the server and the client are co-located. But such a co-location means that the number of group server instances should be proportional to the number of clients, which is an obvious performance, maintenance and administration scalability problem. Second, it is very inefficient to obtain subject privilege attributes over and over even when they are the same during user session.

Another drawback of the approach is that the service, or at least its proxy, should be co-located in the same process because the only language binding available as of May 2000 is defined in C language [Ryutov 2000b]. The main advantage of the API over the other reviewed models is the support for a very flexible and powerful concept of additional conditions that should be enforced by the application or met by the client.

GSS API provides very generic low-level abstraction, the use of which by application developers requires significant integration efforts. This prompted new generations of secu-

rity technologies for distributed application systems such as CORBA and DCOM, in which an application can be developed without any notion of underlying security, including AC, unless it requires the enforcement of complex policies. However, if an application does use bare GSS API and it requires the authorization on fine grain resources or enforcement of complex AC policies, then GAA API, not the EACL, meets most authorization needs of such applications.

3.2 Access Control for Distributed Applications: State of Research

There are three main research directions in addressing the problem of controlling access to the resources of distributed enterprise application systems. They are policy agents, interface proxies and interceptors, and enterprise-wide authorization servers. In this section, we describe and critique each of them.

3.2.1 Policy Agents

By the term “policy agents,” we refer to a direction in the area of AC distributed applications, the approaches of which suggest the enforcement of AC policies by the means of native mechanisms available locally in the computing infrastructure of each application system, as shown in Figure 3-8. They could be the OS AC or add-on packages, AC provided by the middleware (Section 3.1), by DBMS security layers, or even by AC mechanisms of the application integrated environment. The main feature of these approaches is achieving the consistency of authorization policies across application boundaries by the means of centralized AC management via translation of authorization rules into languages supported by local mechanisms, and the distribution of the rules across application systems.

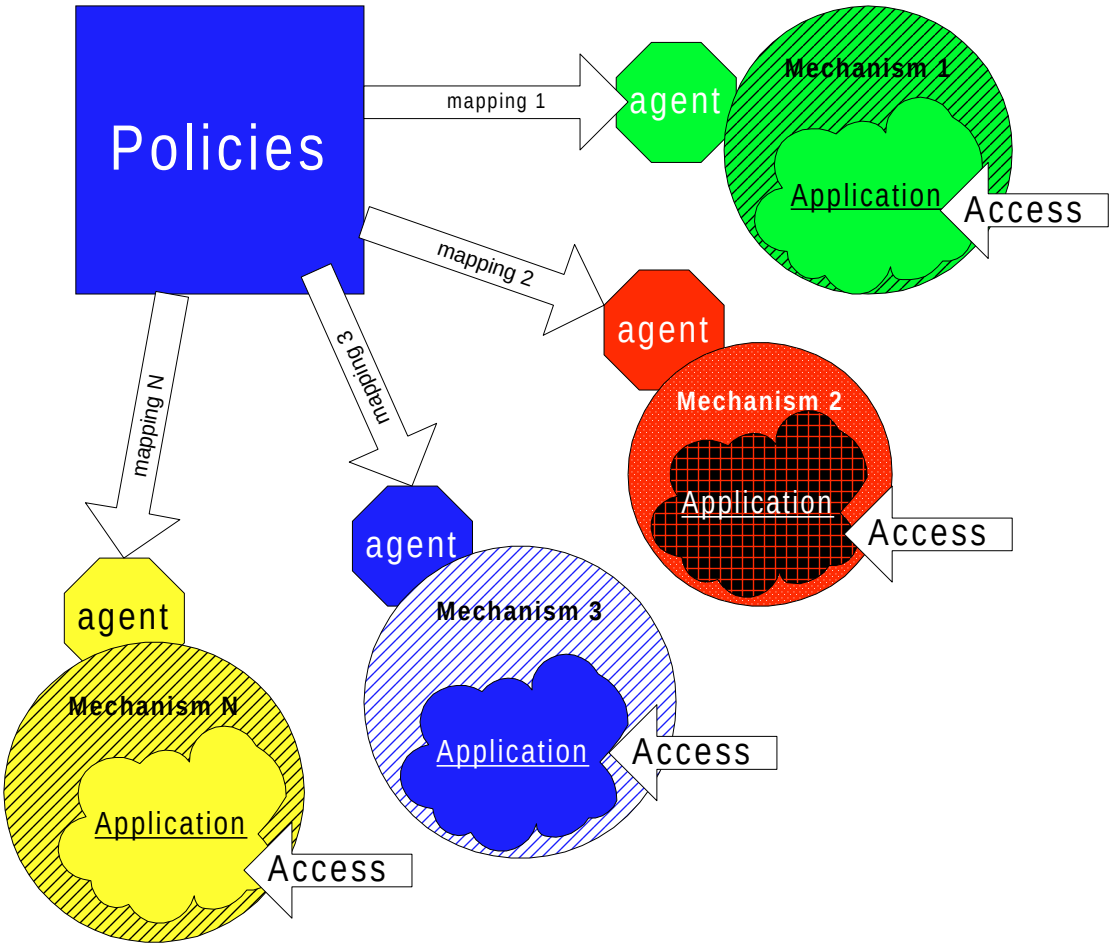


Figure 3-8. Policy Agents

This is achieved with the help of policy agents. The distributed management architecture based on such agents provides the infrastructure necessary to map domain-wide authorization rules into rules specific to particular mechanisms.

All approaches under this direction have the following advantages:

- Inherent fault tolerance. If a mechanism responsible for AC decisions and enforcement fails, only the application system protected by the mechanism becomes affected, while all other systems remain protected.

- For an intruder to gain unauthorized access to all protected resources in a policy domain, either all AC mechanisms or the policy management, mapping and distribution infrastructure have to be subverted. Since the latter can be implemented using off-line techniques, it can be secured much more, without penalizing run-time performance.
- Locality of the decision making process. In a distributed architecture based on policy agents, all AC decisions are made locally and this allows achieving minimum performance penalty.
- Performance scalability. Since the authorization process is naturally distributed over the computing environments of application systems, authorization is an issue local for those environments. Thus a greater number of applications does not cause longer response latency experienced by each application client.

The main challenges facing the approaches are as follows: 1) automation of mapping a global policy into various representations specific to local AC mechanisms, where even for the same AC model there could be different implementations and configurations, 2) the consistency of the enforced global policy, as Hale et al. point out in [Hale 1999], and 3) the preservation of policy semantics when they are mapped into local mechanisms, similar to the problem of translating a program written in a high-level language into architecture-specific binary code.

As the author's experience of performing similar mappings with such commercial systems as Unicenter/TNG [CA 1999] from Computer Associates shows, the process of administering the mapping of the subject's global credential information into local creden-

tials could be so costly and resource-consuming that only very advanced IT departments could afford it. There might be no other way to solve the problem of managing AC in enterprise applications when they are already deployed.

Approaches based on policy agents also suffer from a number of inherent limitations. First, the granularity and expressiveness of AC policies in a policy domain can be only as good as those supported by its most coarse-grain and least expressive mechanism. Second, policy changes can be very slow. For example, on some operating and DBM systems activation of such changes requires re-initialization of system components or even the entire system, which makes policy changes an expensive and prone to temporary inconsistency and frequent downtime periods. This can easily make policies based on periodic authorizations [Bertino 1996a] unaffordable. Because of these challenges and limitations, we believe it is very difficult to support a positive answer to the question of whether this approach employed for new applications is best. Below we describe in details one of the approaches representing the direction of policy agents.

3.2.1.1 Security Policy Mediators from the University of Tulsa

Hale et al. propose in [Hale 1999] an approach for the coordination of security policies and subject credentials across heterogeneous information systems with the focus on loosely coupled federations, where no central authority for federation management is possible. Their approach is twofold. The first part is a ticket-based simple authorization model, to which a number of authorization models (owner-based DAC [NCSC 1987], lattice-based MAC [Bell 1975], RBAC [Sandhu 1996], TBAC [Thomas 1994]) are shown to be mapped. This enables the employment of a single language for authorization rules. The part of their

work we are interested in is an architecture for authorization process and the policy mediation, which enables the consistency of enforced authorization policies.

In the model, each enterprise manages its own policy mediators [Weiderhold 1992]. A security mediator is installed on each computer system that manages protected enterprise resources, as well as on the client systems from which subjects access those resources. Subjects hold a partially implicit and potentially heterogeneous collection of rights to various information resources to which they need to have access. When subject access resources across organizational boundaries, they are called “foreign subjects.” Security mediators determine access rights according to the global policy. In case of foreign subjects, the mediators translate subject credentials according to trans-organizational authorization policy. Each mediator installed on the server host contains the following: a model of the database containing resources, to whom the local system provides access; global security policy expressed in the language of simple authorization model, which Hale et al. developed as part of their work; and coordination policy for managing access by foreign subjects. Coordination policies can take different forms -- mapping foreign subjects to local subjects, assigning local proxies to act as trusted delegates of foreign subjects, requesting *vouchers* from trusted sources for foreign subjects, or mandating joint authorization with local subjects.

Client mediators bundle subject credentials with query fragments to distribute the query to remote systems. Mediators at the application systems, translate the incoming requests into local requests according to the database model of the local system resources, apply their coordination policy to the incoming requests based on the received credentials,

and then authorize the requests according to the global security policy. The authorization might be performed by the application AC mechanism.

The major advantage of this approach to the problem of controlling access to application resources is the support of multiple AC models where each system or enterprise can enforce a model most suitable for its own environment without requiring any changes to what already exists, while continuing the enforcement of the organization-wide authorization policy. Such a global policy is mapped into a concrete authorization model. This enables high adaptability to the changes in applications and computing environments, although it does not accommodate changes in policy types well.

Another significant advantage is that mediators can hide from the application the process of correlating foreign and local subject privilege attributes. This solves the problem of multiple inconsistent subject privilege attribute sets maintained independently in most commercial systems today, and the problem of accessing application systems across organizational boundaries. However, security technologies like SESAME already have addressed this problem by the means of single sign-on when a subject has one set of privilege attributes used for accessing multiple applications.

The approach of security policy mediators inherits all the disadvantages of policy agents direction. In addition, the use of mediators on the clients makes it less scaleable because the amount of change becomes not proportional to the number of application servers but to the number of clients, which is usually significantly larger.

3.2.2 Proxies and Interceptors

The approach of proxies and interceptors, or just “proxy approach” for short, is due to the obvious desire to add new functionality, AC in this case, on top of the old one. This way, existing applications can be enhanced with new features and behavior without changes to their internals. Most capable security services such as CORBA and DCOM follow the approach by using invocation interception in order to enforce various security policies.

The idea is based on either proxing an application interface or intercepting communications between interacting application systems by some other means. Access to an application is controlled externally to it because authorization decisions are made before a system gains control and/or after it dispatches an invocation to another system. In order to achieve it, invocations are intercepted either in the communication or middleware layers, as illustrated in Figure 3-9. Interception can also occur at the application layer, when a

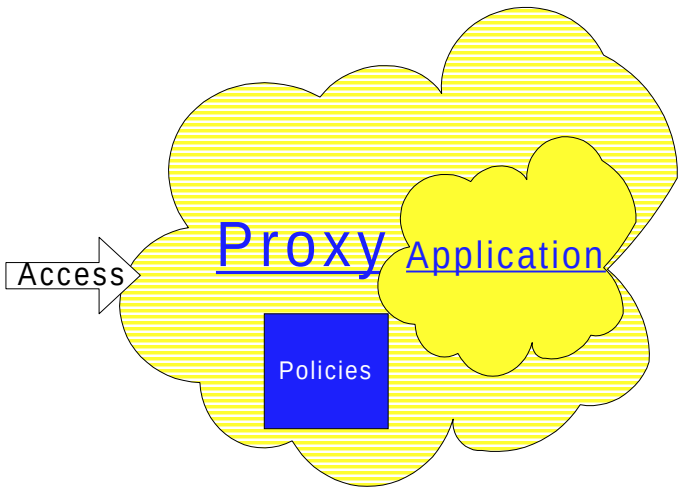


Figure 3-9. Proxies and Interceptors

system is “wrapped” into its interface surrogates, or additional code is inserted by a compiler or other similar tool.

The main advantage of the direction is that it requires hardly any changes to the application system since the reference monitor is implemented externally to it. Another advantage is the ability to make all the decisions locally because interceptors and proxies can be deployed even in the same process space. Also, if authorization decisions are made locally and use local data, the approach features inherent performance scalability.

There are a number of significant limitations however. First, AC granularity cannot be finer than the method level and its arguments (but only when the arguments can be interpreted outside of the method implementation). That is, no approach under this direction allows the control of access to the resource other than interface instances, methods defined on them, and arguments. Second, authorization decisions cannot be made just-in-time. They always have to be made either before or after an application system is in the possession of execution control. Third, because of the above reasons, variables, whose values become available at some point after the method is invoked but before the decision needs to be made, cannot be used in authorization rules.

The main disadvantage is that insuring the consistency of enforced policies as well as the coherency of data used for authorization decisions becomes a challenge,[1] since there are as many instances of access controls as application systems. The administration of proxy-based AC mechanisms will have significant overhead and a high human error rate unless thoroughly automated, which brings us back to the main objective of the policy agents.

1. Although it can be partly solved using the policy agents approach.

We describe and analyze several main approaches following this direction: views represented as objects and used for enforcing AC, role classes, SafeBots, AC in Legion systems and security meta objects.

3.2.2.1 Views as Objects

Hailpern and Ossher describe a model in which application objects can have multiple "views" [Hailpern 1990] whereas each view represents a certain set of methods invokeable by a specified collection of clients. They suggest the model of views for controlling access of clients to servers in object-oriented systems. Even though their approach was originally made in the context of local inter-object invocations, views can be used in distributed heterogeneous application systems.

One of the proposed methods for implementing views is to materialize views as objects (view objects) [Hailpern 1990]. Specifically, all method invocations are addressed to a particular view with the server, client, and method selector as arguments. The view object then checks if the client has access permissions for the given server and method. If the check succeeds, it invokes the server on behalf of the client using a primitive form of invocation not available to the client. View objects act as proxies for server objects, and perform and enforce authorization decisions. The approach is a common representative of the direction.

3.2.2.2 Role Classes

Similar to views as objects, Barkley suggests to use proxies ("role classes" according to his terminology) in order to implement (role-based) AC for application systems [Barkley 1995]. Having methods with the same signatures as the original classes, proxy objects mediate invocation requests. Barkley deviates from a simple proxy model by introducing

another layer of proxies -- now on the client side (“client proxies”). Client proxies are used in order to determine what AC proxy should be used and to direct the call to that proxy. It is assumed that client proxies can be automatically produced and linked to the client application, hence making it completely transparent to the client developers and users. If the environment used for client-server communications cannot determine the right proxy and direct the invocation to it, then some mechanism similar to client proxies must be implemented.

3.2.2.3 SafeBots

SafeBots [Filman 1996a, Filman 1996b] is a concept based on software, possibly mobile, security agents. According to its vision, software security controls are active agents that “wrap” insecure components, communicate with each other, and are smart enough to adapt their actions to the local and global interaction contexts. These agents monitor communications by wrapping an application’s components and can be programmed to perform authentication, AC, intrusion detection, or other security controls. They can be structured either as wrappers for application components, or as independent SafeBot agencies that support the coordination of SafeBot activities, and may confederate with each other for different purposes.

In order to use SafeBots as security wrappers, the authors make a number of assumptions: application systems have well-defined interfaces, can be sequestered (i.e. cannot be invoked directly without going through SafeBots), and can be substituted. They propose the automation of application system wrapping.

SafeBots approach attempts to marry mobile agent technology with security controls and brings advantages and disadvantages of the former to the latter. SafeBots enable the implementation of cost-effective, redundant, extendable security controls including AC. The authors themselves list several inherent limitations [Filman 1996b]: it is difficult to wrap application systems with complex or rich interfaces (e.g., applications with complex GUIs, shells or other programming scripts and environments); SafeBots complicate and increase enterprise security administration; subverting a SafeBot could become a way to attack systems; and inept security designers could design SafeBots that actually reduce overall system security; at a time of crisis, SafeBot activity could tie up a system when the resources are most needed. In addition to these limitations, change of policies requires re-deployment of SafeBots over all systems affected by the change, which is very expensive and lengthy in large enterprise settings.

3.2.2.4 Legion

The Legion system [Grimshaw 1998, Grimshaw 1997, Wulf 1996], developed at the University of Virginia, defines a software architecture designed to support the use of large collections of heterogeneous computing resources distributed across local- and wide-area networks as a single, seamless virtual machine. Legion's components include a run-time system, Legion-aware compilers that target this run-time system, and programming languages that provide application programmers with a high level abstraction of the system. The system itself creates, schedules, and utilizes distributed objects to execute the application programs.

Legion security architecture follows the overall design philosophy of the project -- “No single policy or static set of policies will satisfy every user, so users must be allowed to determine their own priorities and to implement their own solutions as much as possible” [Grimshaw 1998]. The architecture requires that every class defines a special member function `MayI`, which has default behavior granting access. Legion security automatically calls this function before any method invocation, and permits the invocation only if `MayI` grants access. The approach supports mandatory AC via implementation inheritance or delegation. In order to exercise MAC, an implementation of the `MayI` method must either delegate its behavior to or inherit it from an organization-wide implementation of `MayI`. This raises the question of performance scalability. DAC policies are supported via custom implementation of `MayI` for a class. Class implementers can resort to the default implementation of `MayI`, granting access unconditionally, or inherit implementation from a class they trust, or write a new one. The code for implementing the security policy is localized to the `MayI` method rather than distributed among the member methods. Method `IWantTo`, a counterpart of `MayI`, is invoked by Legion security mechanisms every time an object makes an invocation on other objects. This enables the enforcement of lattice-based mandatory security policies [Bell 1975], which control the flow of information to and from objects.

By enabling nominal initial overhead with simple AC policies, Legion security design features minimality principle. The use of class-specific `MayI` and `IWantTo` methods makes the implementation of arbitrary discretionary policies on class-by-class basis easy. However, the enforcement of mandatory policies or those discretionary policies that need to be consistent across several systems, seem to be difficult because security administrators

do not have control over what logic is implemented by application objects launched by users. Futhermore, the change of enterprise policies requires changes in implementations of `MayI` and `IWantTo` methods, which is not a realistic requirement for contemporary enterprises with large-scale deployments of service-based and object-based applications. This can be avoided by using only one instance of these methods across multiple applications but then the performance should be addressed. Also, implementation inheritance or delegation requires control over the implementation of an application system, which becomes less and less realistic with advances of COTS and component systems provided by various vendors.

3.2.2.5 Security Meta Objects

Security meta objects (SMOs) [Riechmann 1997, Riechmann 1998] is a paradigm proposed recently by Riechmann and Hauck from the university of Erlangen-Nurnberg, Germany. The area of its application is strictly object-based systems.

They propose to “attach” one or more special objects to an object reference. These special objects are invoked for each security-relevant operation on the object reference. The special objects are not visible to the application; that is, protected and unprotected object references look the same to it. Such special objects can be considered as meta objects [Maes 1987]. SMOs are attached on a per-reference basis. There may be many references to an object which are not protected or protected by a different SMO. If a client invokes a method via a protected reference, a special check method of the meta object is implicitly invoked. This method gains access to some meta information, such as name and parameters of the method to be invoked. The check method can decide whether it wants to grant access or

not. To grant access, it returns control to the run-time system, which continues with the method invocation. If access is to be denied, an exception is raised or the invocation is terminated with an error result. If several SMOs are attached to the same reference, then they are “asked” sequentially before access is granted. A single SMO can be used to protect multiple references. It is not possible to detach SMOs from a reference unless the SMO removes itself.

Meta objects can be used for enforcing arbitrary AC policies as well as for implicit and transitive AC of object references passed as a parameter or result. Another advantage of the approach is that it allows the development of SMOs totally independent of the objects they protect and vice versa.

The approach provides flexibility lacking from other paradigms in this direction, such as Legion (Section 3.2.2.4). In particular, it allows “attaching” multiple SMOs to the same object reference so that several policies or additional functionalities can be composed. This is very similar to CORBA request interceptors [OMG 1996a], which are invoked before an invocation is scheduled on an interface implementation by the ORB or before it is prepared to be sent to another CORBA object. However, SMOs are “attached” to an object reference. Whereas, CORBA interceptors are “attached” to an instance of an interface implementation identified by an object key in the scope of the object adapter. Thus, the same interceptors are used to control access to an object, as opposed to the SMO paradigm, where different SMOs can be attached to different object references that “point” to the same object.

In addition to the general limitations of proxies and interceptors direction discussed in Section 3.2.2, SMO paradigm has a number of its own drawbacks. First, in order for the

SMO approach to be realistic, support for meta-objects is needed in the desired middleware technology. SMO authors use MetaJava system [Kleinoder 1996] to prototype the concept. However, to the best of our knowledge contemporary industrial middleware systems such as CORBA, DCE, Java, and DCOM do not support attachment of (meta) objects to object references. Second, they assume that object references are safe, that is, the references are only generated and controlled by a trusted run-time system and cannot be tampered with. It is a very restricting assumption. Third, we cannot find a solution for the problem when the policies governing access to an object change, in such a way that new SMOs need to be added or old ones removed or replaced, after the object's reference has been released. This means that policies for an object cannot be changed after an object reference is released to the world. This renders SMO-based solutions unusable for the real-life computing enterprises, unless the limitation is somehow addressed.

3.2.3 Authorization Servers

The third direction in AC for distributed application systems is based on authorization services. Such a service is logically one per policy domain, even though its instances can be replicated in order to achieve desired level of availability, fault tolerance, performance and scalability. Authorization decisions are made by an instance of the service -- authorization server. An application system enforces decisions made by an authorization service without knowing how they have been made, as shown in Figure 3-10. Thus both the application and the authorization server are part of a reference monitor [Anderson 1972].

We consider a research project on generalized framework for AC (GFAC) [Abrams 1991, Abrams 1990a, Abrams 1989, Abrams 1990b] at the MITRE Corporation as a pre-

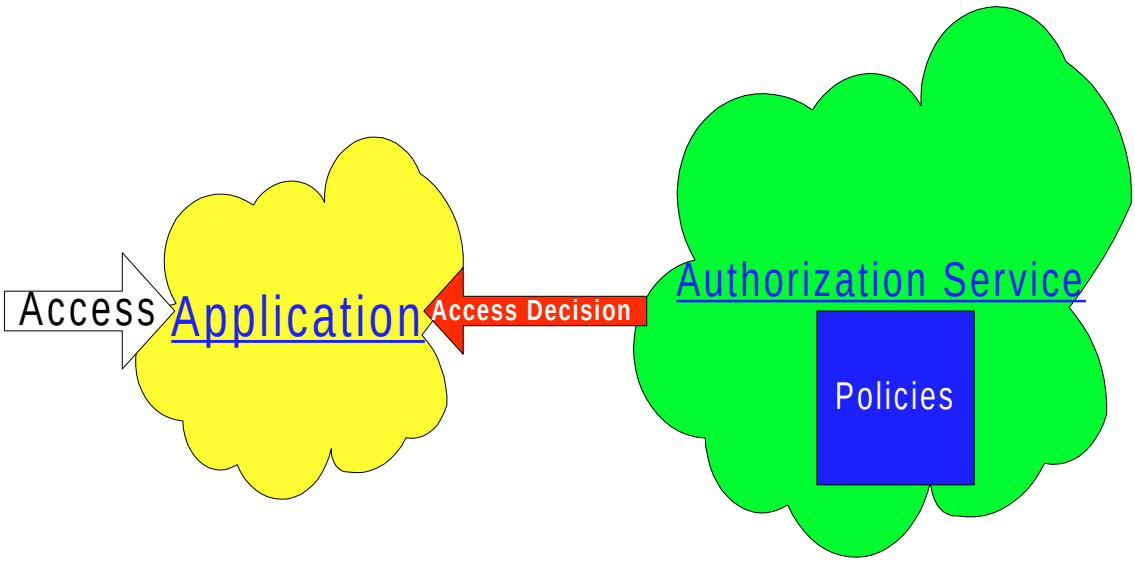


Figure 3-10. Authorization Servers

decessor of all other attempts to develop the concept of authorization service. The project endeavored to build a theoretical framework that explicitly recognizes the main information components for AC -- subject and object security-related attributes, access context, authorities, and rules, where they showed that "the rules for AC are an entity that is separate from, although necessarily related to, the model of the trusted computing base (TCB) interface" [LaPadula 1990]. Moreover, La Padula concludes that in a networking environment "one can conceive of an access control engine realized as a server, with access requests handled via a remote procedure call mechanism" [LaPadula 1990].

The main advantages of approaches based on the concept of an authorization server are:

- Logical centralization of AC rules, which gives inherent consistency and coherency of authorization policies enforced throughout a policy domain.

- Ease of policy change and update because authorization is made in a logically single place.
- Since authorization logic is centralized and decoupled from the application logic, it is possible to replace a policy with a new one of a different type without affecting application systems.
- Centralization of authorization rules naturally features single point of administration for all systems belonging to one AC policy domain, significantly lowering the cost of administration.
- Since an application system decides when to obtain an authorization decision from the server, it can do so right at the time when such a decision is needed.
- Authorization decisions on resources of any level of granularity can be obtained from the server because an application uses the server while it is processing a request. This lifts the limitation of the other approaches, namely Proxies and Interceptors, in which the granularity can be only as fine as a method on an interface instance.

For this approach to be feasible, several important issues must be addressed. First, it is much more challenging to design an implementation of such a server so that it does not become a bottleneck in terms of performance. Second, if the server fails, all application systems served by it will have to resort to a simplistic and very limiting policy such as “always deny” or “always grant,” which would render systems un-operational. Thus provision of high degree fault-tolerance needs to accompany such servers. We describe in detail the

reported work on authorization servers since it directly relates to the subject of this dissertation.

3.2.3.1 Authorization Server from HP

The design principles described by Varadharajan et al. in [Varadharajan 1998] were used in the development of Praesidium Authorization Server [HP 1996], which is a rule-based authorization facility for distributed application systems. The work, according to the paper, began in 1993, although the paper, apparently the first research reporting the work, appeared in 1998. This is one of the first works in the area of authorization service practical design and implementation for distributed systems that has been reported in the literature.

Varadharajan et al. outline several design principles for authorization in distributed systems that some architects of enterprise security systems might find useful, although these principles are not supported with any study on their validity. The authors propose to classify security information in a two-dimensional space: one dimension is the generality and the other is the dynamics of the information. Hence, they identify three groups of information: generic and static, specific and static, and specific and dynamic. Using this classification, they suggest to design distributed security infrastructure in such a way that the information is stored either in a central server and is "pushed" to the target by the client, or near or on the target and "pulled" at the time of the decision process. The design suggestions outline three main parts of an authorization infrastructure: 1) a domain-wide central authority storing and managing generic-static security information, 2) a domain-wide central authority dealing with specific-static information, and 3) a per target (or per a group of related targets) component dealing with specific-dynamic information.

Another contribution of the work is the location and the types of authorization checks. Although it is not a novel point of view, we did not encounter similar considerations in the literature. The paper suggests considering three points of authorization checks: a check if a subject should access an application at all (Level I), then a check on the type of the function to be performed (Level II), and a check from within the application program (Level III). We believe that the value of this classification is in the establishment of the common language and conceptual points for reasoning about authorization in distributed systems.

The authors distinguish between two stages in the functionality of their distributed authorization service: the administration phase and the “run-time” phase. They make such a delineation because they bring forward two arguments for maintaining distinct representations of authorization information in the service. The arguments are the existence of information captured during the administration phase that can be compiled before access decision time for the performance purposes, and the possibility to use different replication strategies for the administrative information versus the information needed for access evaluation decisions at the application servers. This premise is the driving force behind the design of the authorization server, which considers the system as two domains -- administration (the management of privileges and profiles granted to subjects) and run-time (furnishes authorizations to applications).

The run-time domain consists of an evaluation engine and the run-time database of precompiled rules. The authorization decisions are made in regards to the following:

Level I DCE IDL interface name (process names for GSS-API),

Level II Name of the procedure specified by the DCE IDL file (application-defined for GSS-API),

Level III authorization rule.

The service design outlines several elements and approaches that have been employed in other similar works including Adage (Section 3.2.3.3 on page 86) and the work described in this dissertation. The main elements are the encapsulation of authorization functionality into a service available in the distributed environment and the explicit division of the server into administrative and run-time domains.

The main drawback of the reported work is the lack of any study on the validation of the design principles and evaluation of the authorization service proposed in the paper. No research analyzing the suggested approach is in the paper or published separately.

3.2.3.2 Distributed Authorization Service from the University of Texas

Woo and Lam from the University of Texas at Austin researched the theory and practice of constructing a distributed authorization service [Woo 1993a, Woo 1993b, Woo 1993c, Woo 1993d, Woo 1998]. As as result, they designed such a service [Woo 1993c, Woo 1998], the key features of which are a language-based approach for specifying authorization rules and authenticated delegation. Their design is based on the prior work of Neuman [Neuman 1993], where he outlines an authorization protocol for distributed application systems.

They observe that most existing application systems perform their own authentication, authorization, accounting and auditing [Woo 1993c]. Even though authorization is often

perceived to be tightly coupled with an application and hence cannot be easily abstracted, Woo and Lam suggest that a better approach would be to factor these functions out and implement them separately as a set of core services. The set can in turn be used as a basis for building other generic services and application systems.

The main motivation of their research was to study two problems in the construction of an authorization service for distributed systems: (1) how to identify the commonalities in authorization requirements of application systems and to design an appropriate abstract representation to capture these commonalities, as well as (2) what secure protocols should be used for off-loading authorization from application systems to authorization servers and for interactions among various enterprise entities. Apparently they identify these problems by drawing an analogy with authentication services in distributed systems [Burrows 1990, Lampson 1991, Woo 1992], where common representation of user credentials and interaction protocols for secure exchange of authentication information are the distinguishing factors of various architectures [IETF 1993, Molva 1992, Neuman 1994b, OMG 1996b, OSF 1996, Schiller 1988, Tardo 1991]. However, it is not evident that those are really the main problems in constructing a distributed authorization service. An application system, for example, might just use RPC over secure (i.e. authenticity, confidentiality, and integrity protection) channel to obtain an authorization decision from an authorization server [Beznosov 1999b, Varadharajan 1998, Zurko 1998] thus avoiding the issue of the interaction protocols.

The authors claim the following advantages of a separate authorization service [Woo 1993c]: 1) savings in re-implementation effort for each application system, 2) application

systems are relieved of the authorization task, which can lead to higher throughput, 3) a specialized authorization service can afford the use of better methods in making AC decisions than would be justified for individual application systems, 4) an authorization service can be verified to be secure once and for all, reducing the complexity in verifying the security of an application system, 5) anonymity (if desired) can be achieved with the use of a trusted authorization service, 6) a uniform authorization service can contribute to the uniformity of accounting and auditing functions, hence facilitating the construction of distributed accounting and auditing services.

The architecture of the distributed authorization service, proposed by Woo and Lam, consists of five main entities:

1. **Service Locator**. It responds to a client's request with a list of application systems that implement the requested service, and possibly a list of authorization servers for the application systems. In its functions, such a locator is very similar to CORBA directory or trader services.

2. **Authentication server**. An authentication server authenticates users during their initial sign-on and supplies them with an initial set of credentials, as well as enables mutual authentication between clients and servers.

3. **Authorization server**. An authorization server performs authorization on behalf of an application system if the system elects to off-load its authorization to the server. To do so, an application needs to contract an authorization server for this

purpose using a contracting protocol. An authorization server provides clients with authorization certificates which are to be forwarded by clients to applications along with their requests.

4. **Group server**. A group server maintains and provides clients with group membership information in the form of (non)membership certificates to be forwarded to the authorization server together with the client's requests.

5. **System monitor**. By the means of several processes executing a distributed algorithm, a system monitor tracks the values of system predicates indicating overall system status. It is not clear why Woo and Lam included a system monitor, pertaining more to the network management then to security, in the architecture of an authorization service.

The entities are services that in concert provide the functionality required for an application system to delegate authorization and to monitor the systems. While being logically disjoint, all or some of them can be integrated into one server.

Woo and Lam designed a protocol enabling the interaction among the five entities. We are going to omit a detailed description of the interaction and the supporting protocol, which can be found in [Woo 1993c]. Here, we will point to the key features of the interaction required for the successful use of authorization in the model.

Application system *E* locates, possibly through a service locator, an authorization server *A*, and, after mutual authentication via authentication service, contracts it to authorize access to *E*. The contract is enacted using a contracting protocol at the end of which A

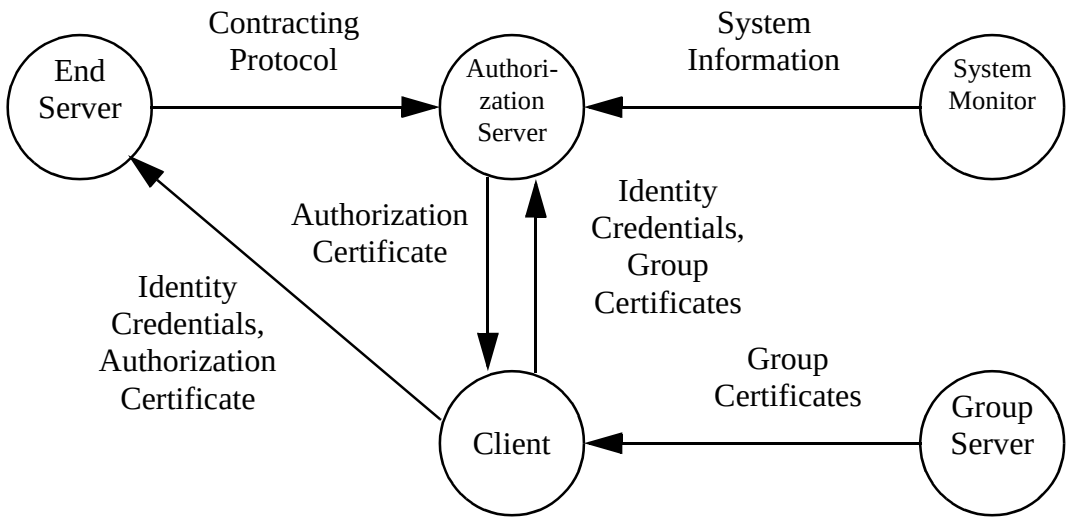


Figure 3-11. Authorization-related Interactions (from [Woo 1993c])

has an authorization specification, which is a description of authorization policies, expressed in generalized ACL (GACL), governing access to *E* services. Upon success in contracting, *E* notifies service locator that A is delegated to perform authorization for *E*. From now on, every client is responsible for obtaining a reference to *A* from a service locator, and acquiring an authorization certificate from *A*, before *E* will serve the client. A group service comes into play, when a client requests an authorization certificate from *A*, which might require the client to obtain one or more (non)membership certificates from a group server, before *A* can authorize the client.

The ideas used by Woo and Lam (we will refer to it as WL architecture/service) are very similar to the ones in Kerberos -- a client goes to a trusted third party (TGS in Kerberos, and authorization service in WL work) and gets a ticket (session key in the former and authorization certificate in the latter) in order to access a server. The work seems to be an effort to cure Kerberos and provide authorization service that would be in harmony with it. Kerberos lacks the management of subject privilege attributes and does not assume any

middleware infrastructure in place. This is why such different functionalities and corresponding services, as authentication, privilege attribute management (via group service), location discovery, authorization, and even system monitoring are mixed into WL architecture.

The concept of a group service deviates from the traditional model where subject privilege attributes (including group membership ones) are identified during authentication phase, and fixed during the session lifetime. The group service allows the use of authentication technology that is not capable of identifying all privilege attributes of the subject during authentication phase. This can potentially make the authorization process very inefficient, if the client has to interact additionally with the remote group server for each application request.

The granularity of authorization decisions in WL solution cannot be fine because prior to contacting an application, the client needs to know exactly for what authorization it should ask the authorization service. For example, in a health care organization that has a service allowing various queries of type “give me records of those patients that have attribute X,” the client would have to obtain authorizations for accessing records of all patients selected by the query, which is not possible to know before the query is performed. In some cases, a client knows at most the application and its function that it wants to invoke, and the invocation arguments, and not what resources have to be accessed in order to perform that function. This makes WL authorization service architecture useful only for such network services as Telnet [Postel 1983], FTP [Postel 1985], etc., where the only authori-

zation required is to open a session with the server. The rest is controlled via operating system AC mechanisms.

Another critique about WL approach is the contracting protocol between an application system and the authorization server. The authors assume that the application is the owner and the source of its authorization policies. However, in most mid to large size organizations, authorization policies are enterprise-specific, and not application-specific. Thus, the policies should not reside on the application. An application ideally should not be involved in policy management, administration, or distribution.

WL architecture cannot be used in those distributed computations which require an invocation chain with the delegation of client privileges to the intermediate services, because the client is involved in obtaining authorization for any invocations on its behalf. This limits the approach to those invocations where there are no delegated sequences of calls among remote application servers.

Overall, the WL approach to application authorization achieves its goals listed above. However, it has significant limitations that render the applicability to only simple distributed systems that have Kerberos as their primary security technology, and do not have requirements for fine-grain AC or invocation with delegation of client privileges.

3.2.3.3 Adage

Zurko et al. report on the design and their studies made on the Authorization toolkit for Distributed Applications and Groups (Adage) [Zurko 1998], which is mainly an authorization service for distributed computing environments. Adage architecture was based on the

following principles: user-centered design, policy neutrality, modularity and the use of RBAC foundation. The primary goal of their work was to prototype an authorization service for use with distributed applications whose emphasis was on the usability of its administrative interface and tools.

The Adage system consists of a policy definition client for administering policies and a policy decision server for furnishing authorization decisions. The client contains the GUI and Authorization Language (AL) interpreter and communicates with the Authorization Decision Server (ADS) through the administration API. The GUI and AL can be replaced with other clients. Applications wishing an authorization decision access the ADS through the authorization API. Administration and authorization APIs are defined using CORBA IDL and implemented via CORBA technology.

The ADS stores policy information supplied by the administrative clients in a database called the User Authorization Database (UAD). The ADS contains a translator for transforming the information in the UAD into a form more suitable for making fast authorization decisions. This database is called the Engine Authorization Database (EAD). The authorization engine is the other major piece of the ADS, which uses the EAD to find rules applicable to a given decision.

The main research objective of Adage project is to design an authorization service for distributed application systems that would enable the use of administrative and application interfaces constructed according to the principles of psychological acceptability and usability and to perform usability study of the system. Such a goal is orthogonal to the goal of this

work described in Chapter 2. Hence, we believe that studies reported in this dissertation are complemented by the results reported by the Adage project.

3.3 Chapter Summary

The idea of authorization decisions being separated from application logic is not new. An abstract model of a reference monitor [Anderson 1972] is a classical example of authorization decisions being made and enforced outside of applications. The industry achieved considerable results in regards to the control of access to operating system and middleware resources. Most operating systems implement authorization logic in the security part of their kernels. There are also special-purpose ad-on security software packages that furnish authorization decisions to operating systems [Benantar 1996, CA 1998a, CA 1998b, IBM 1976].

Middleware technologies provide several means to control the use of distributed services exposed via application interfaces. There are two groups of technologies used for securing distributed software systems. One group is the technologies that merely provide party authentication, communication protection, and access control independently of the underlying communication technology: Kerberos [IETF 1993, Neuman 1994a], SESAME [Kaijser 1998, Parker 1995] and GAA API [Ryutov 2000a]. This enables using and mixing any desired communication protocols and media, but developers are overburdened with significant efforts to integrate the security technology with the underlying communications.

Another group is middleware technologies, such as CORBA [OMG 1996b], DCE [Gittler 1995], Java [Lai 1999], and DCOM [Microsoft 1998], that provide the underlying com-

munication infrastructure along with the security subsystem, thus enjoying reasonable integration of both and much more seamless use of the former by developers. Moreover, some of them enable basic access control completely outside of an application system because access decision and enforcement occur before the remote call is dispatched to the application server.

The Java Authentication and Authorization Service (JAAS) is designed to provide a framework and a standard programming interface for authenticating users and for assigning privileges to users. Access control is enforced only on system resources, such as files, sockets, etc. but not on Java objects and other application resources. JAAS has very generic and extensible support for different privilege attributes which can be easily defined via new classes. The source code base, the identity of the code signer, and the value of the subject privilege attribute are passed to the authorization code via `Policy` class interface for authorization decisions. JAAS allows any granularity of authorization decisions, and it does not constrain implementers of authorization policies to any particular mechanism or to the information used for the decisions. It also enables seamless change of policies. However, the architecture does not address the consistency of authorization policies across multiple applications. Nor does it have any provisions for achieving performance and administration scalability.

In the DCE, application systems are expected to enforce and provide administrative access to authorization policies themselves. An application system can use DCE access control list (ACL) but it has to implement most of access control functionality, including ACL storage and manager, and its administration. DCE Security supplies an application

only with the caller's subject and group identities. Cross-application administration of authorization logic is not directly supported although administrative interface for doing the administration on per-application basis is defined, yet it is not a scalable solution.

The security model of DCOM resembles DCE security. As with DCE, ACLs are used to code authorization policies. The main advance of DCOM is the capability of enforcing policies outside of objects with the presence of process and host-specific policies in addition to the capability for an application to use DCOM Security API for its own AC. The authorization model is significantly hindered by the granularity of the so-called "component-specific" policy where there is no distinction among different objects and their methods in the same OS process. Component- and host-wide policies implicitly introduce the notion of access policy domains; still it is not clear if such domain partitioning is an administratively scalable and functionally successful solution. The administration has to be performed individually on each host or even for each process, which is better than in DCE but still limited. Although DCOM Security provides ways for application systems to exercise fine grain AC in an application-specific way, application-specific policies cannot be enforced and only security-related attributes of subjects and objects can serve as input for external AC.

SESAME is an architecture for security services which does not specify a communication layer. Thus it cannot control pre/post invocation events. This is why AC and other security functionality has to be specifically activated by an application. This prevents SESAME from providing AC externally to applications. Another drawback of SESAME authorization is the lack of support for applying one policy to several application systems

located on separate hosts. The unit of authorization check is an application system. All these, especially the granularity of AC, make SESAME less attractive then JAAS, DCE, DCOM or CORBA technologies for engineering access control to application resources. However, SESAME is neutral to the underlying communication protocols, and is known for its advanced model of privilege attributes management and propagation. This makes it indispensable for building heterogeneous, multi-technology and multi-organization distributed applications that require authorization based on privilege attributes, other than user identity, and the use of different communication technologies.

In CORBA Security, access control can be enforced completely outside of an application system. AC decisions are based on subject privilege attributes, required rights of the method, and the access control policies of the domains to which the object belongs. The AC model scales very well without losing fine granularity, for the decisions could be specific to each object, if the object is located in a separate domain, or a large group of objects could be associated with one policy domain. Unlike DCOM, CORBA objects residing on different computers can be associated with the same policy domains. Because CS defines advanced concepts of privilege attributes, it enables AC policies based on roles, groups, clearance, and any other security-related attributes of subjects. User grouping via privilege attributes, object grouping via policy domains, and method grouping via the concept of required rights enable high administration and performance scalability of AC mechanisms. If an application system is to enforce its own AC, it can do so with the help of CORBA Security API, which allows it to obtain subject security attributes, including privilege attributes. However, application-specific policies are difficult to enforce and the use of application-specific information in the CORBA AC is limited.

Generic Authorization and Access Control API (GAA API), published as an IETF Internet draft, defines a framework for application authorization. The API aims to address the lack of standard authorization interfaces for those applications which use the generic security service (GSS) API. This is why GAA API's authorization model specifically fits into the existing GSS API. If an application uses GSS API, which provides very generic low-level abstraction, and it requires the protection of fine grain resources or the enforcement of complex authorization policies, then GAA API defines interface with enough capabilities for most applications. The main advantage of the API over the other reviewed models is the support for the very flexible and powerful concept of additional conditions that can support application-specific policies. The drawbacks of the API are that it only defines the interface between an application and an authorization mechanism, and the model addresses neither administration scalability nor the consistency of authorization policies across multiple applications.

Ideally, all security functionality should be engineered outside of an application system, therefore making it so called "security unaware." However, this is difficult to achieve for the majority of application systems, where access control, and other security policies, are too complex, or require too fine control, to be supported by the general-purpose security technologies. This is why fine-grain control of distributed application resources is done traditionally in an ad-hoc manner [Wilson 1997], and there are no automated means to ensure enterprise-wide consistency of such controls.

The research community has being working towards systematic ways of controlling access to resources in distributed heterogeneous application systems. There are three main

research directions in addressing the problem. They are policy agents, interface proxies and interceptors, as well as enterprise-wide authorization services.

The direction of policy agents is motivated mainly by the goal of accommodating the existing body of products and technologies already deployed in organizations. The key property of the direction is centralized AC management via the translation of authorization rules into languages supported by local mechanisms, and the distribution of the rules across systems, which is achieved with the help of policy agents residing on computers hosting applications systems.

Approaches under this direction have a number of advantages: there is inherent fault tolerance; enterprise security is naturally compartmentalized without penalizing run-time performance; the architecture facilitates achieving nominal performance overhead; there is high degree of run-time autonomy -- a trait essential for achieving performance scalability and fault tolerance.

The main challenges facing the approaches are the consistency of enforced global policies and automation of mapping a global policy into various instances of AC mechanism languages and representations. The approaches also suffer from a number of inherent limitations. First, the granularity and expressiveness of AC policies in a policy domain can be only as good as the policies supported by the most coarse-grain and least expressive AC mechanism in that domain. Second, distribution of policy updates can be very slow, which would easily make policies based on periodic authorizations un-affordable. The direction of policy agents becomes irreplaceable, if other approaches, such as proxies and authorization services, fail in those circumstances when application systems are already deployed.

The question if it is the best way to address the problem of application-level AC for newly developed systems remains opened.

The approaches under another direction employ either interface proxies or interception of inter-application communications. Access to an application system is controlled externally. Authorization decisions are made before an application system gains control and/or after it dispatches an invocation to another system. In order to achieve it, invocations are intercepted either in the communication, middleware, or at the application layer.

The main advantages of the direction are that it does not require almost any changes to the application system, the reference monitor is implemented externally to it, and its size can be controlled by security developers. This makes the direction a good alternative to policy agents approach for controlling access to resources of already deployed applications. Moreover, if an existing application lacks any AC mechanism, proxies and interceptors become the only choice. Another advantage is the ability to make all the decisions locally to an application system, which facilitates performance scalability.

There are a number of significant limitations though. First, the granularity of AC cannot be finer than method and, when arguments can be interpreted outside of method implementation, its arguments. Second, the decisions always have to be made either before or after an application system is in possession of control. Third, variables, whose values become available at some point after the method is invoked but before a decision needs to be made, cannot be used in authorization decisions. Fourth, since there are as many instances of access controls as application systems, insuring the consistency of enforced

policies as well as the coherency of data used for authorization decisions becomes a challenge.

Another direction is based on authorization services. Decisions provided by an instance of the service, authorization server, are enforced by an application system. Both an application system and an authorization server constitute a reference monitor, which requires an application system to be trusted to enforce AC decisions.

The goal of authorization services is to factor common AC decision functions out of application systems and implement them separately as an infrastructure service. The main advantages of the direction are inherent consistency and the coherence of authorization policies; the ease of policy changes and updates because authorization is made in a logically single place; the ability to change policies and their policy types without affecting application systems; the relatively low cost of access control administration; the ability to obtain authorization decisions just when they are needed; and potentially any level of granularity of protected resources.

However, in order to construct a successful architecture for a distributed authorization service, one must address several key problems. They are performance, fault tolerance, scalability, security of communicating authorization information, the guarantee of authorization decisions being enforced, and the common representation of information used for making the decisions.

We expect that successful architectural solutions most probably will employ a combination of proxies, interceptors, policy agents, and authorization services because solutions

from all three groups complement each other. For systems with existing AC mechanisms tightly integrated into applications, *policy agents* is the only choice. In those existing systems where AC mechanisms are missing, weak, or have too coarse granularity, *interceptors* and *proxies,* combined with the ideas from *policy agents* and *authorization services* could cure the problem. New applications with requirements for fine-grain access control, complex or very dynamic authorization policies or to be deployed in organizations of different types (e.g. military, government, finance, health care, telecommunications) and sizes, will be best constructed with the use of the *authorization server* approach.

4 Supporting RBAC Using CORBA Security

We surveyed the AC mechanisms of the existing middleware technologies in the previous chapter and showed that they are inadequate for solving the problem of controlling access to application resources completely. However, some of the mechanisms, such as in CORBA and DCOM, allow the enforcement of authorization policies outside of applications. In addition, they are very well integrated with the corresponding services. These two factors make the use of middleware AC mechanisms, when they are sufficient, more favorable than application-level control. The latter is used when the mechanisms are functionally inadequate. Before a system architect opts to employ application-level AC, it is important to take maximum advantage of middleware AC. This is why the study of middleware AC capabilities is crucial for engineering the protection of application resources.

In this chapter, we make two contributions. First, we show the capabilities of the CORBA AC mechanism by providing a detailed and illustrative description. More importantly, we propose a CORBA protection system configuration which formally defines the state of the system. Using the definition, we specify an algorithm for making authorization decisions in CORBA. In addition to the precise explanation of the CS AC semantics, the algorithm fills in the gap in the specification [OMG 1996b], which uses only English prose to explain how AC decisions are performed. Second, we show how role-based access control (RBAC) models could be supported using the CORBA Security service. Using the

defined configuration of the CORBA protection system, we provide definitions of $RBAC_0$ and $RBAC_1$ models in the language of CORBA Security. Furthermore, we describe what is required from an implementation of the CORBA Security service in order to support $RBAC_0$-$RBAC_3$ models. Our approach allows an implementation compliant with CS specification to support $RBAC_0$. Additional functionality, which is beyond the scope of CS specification, should be implemented in order to support $RBAC_1$ and/or $RBAC_2$. This work advances the understanding of CORBA AC mechanisms' capabilities, which is vital to the use of middleware in protecting application resources. The content of this chapter is based on the materials from [Beznosov 1999a].

4.1 Overview of RBAC and Motivations

RBAC [Sandhu 1996] is a family of reference models in which permissions are associated with roles and users are assigned to appropriate roles. A role can represent competency, authority, responsibility or specific duty assignments. Some variations of RBAC include the capability to establish relations between roles, between permissions and roles, and between users and roles. There are four established RBAC reference models: unrelated roles ($RBAC_0$), role-hierarchies ($RBAC_1$), user and role assignment constraints ($RBAC_2$), and both hierarchies and constraints ($RBAC_3$). RBAC supports three security principles: least privilege, separation of duties and data abstraction.

A major purpose of RBAC is to facilitate access control administration and review. RBAC is a promising approach to address the needs of the commercial enterprises better than lattice-based mandatory access control (MAC) [Bell 1975] and owner-based discretionary access control (DAC) [Lampson 1971]. Recent series of papers describe ways to

model or implement RBAC using the technologies employed by the commercial users: Oracle [Notargiacomo 1995], NetWare [Epstein 1995], Java [Giuri 1998], DG/UX [Meyers 1997], object-oriented systems [Barkley 1995], object-oriented databases [Wong 1997], MS Windows NT [Barkley 1998], and enterprise security management systems [Awischus 1997]. Evidence of RBAC recognition in the US government is the fact that the proposed rules on security from the Department of Health and Human Services [DHHS 1998] include RBAC as one of the required choices for access control.

At the same time, the commercial market is experiencing the spread of systems based on CORBA technology. Due to its general nature, CORBA Security (CS) is not tailored to any particular access control model. Instead, it defines a general mechanism which is supposed to be adequate for the majority of cases and could be configured to support various access control models. For example, it was shown how to implement lattice-based MAC using the CORBA authorization model [Karjoth 1998]. In the next few years we expect to witness significant financial investments in the enterprise-wide deployment of CS in commercial and government organizations, including those who will construct their security policies utilizing RBAC concepts. It is important to foresee if CS will fully support RBAC models. However, we are not aware of any work in the research community that has explored the potential of CS for the support of RBAC reference models.

4.2 CORBA Access Control Mechanisms

First, we give a detailed, though informal, description of the CORBA AC mechanism. Then we formulate a CORBA protection state configuration and define the authorization

algorithm. We will use the language of the configuration later in the chapter to discuss the support of RBAC by CORBA Security.

4.2.1 Informal Description

We introduced the main concepts of CORBA Security in Chapter 3. Before we go into detailed discussion of CS AC mechanisms, let us briefly review CS. In short, all object invocations are mediated by the appropriate CS functions for the enforcement of various security policies. The functions are tightly integrated with the ORB because all messages between CORBA objects and clients are passed through the ORB.

CS authentication architecture is very much similar to the one of SESAME. A user uses a *user sponsor* to authenticate to the CS environment. A user sponsor is a logical part of client application. It authenticates on behalf of a user with and obtains authenticated credentials from an instance of interface `SecurityLevel2::PrincipalAuthenticator`, as shown in Figure 4-1. Instances of user sponsor implement user interface

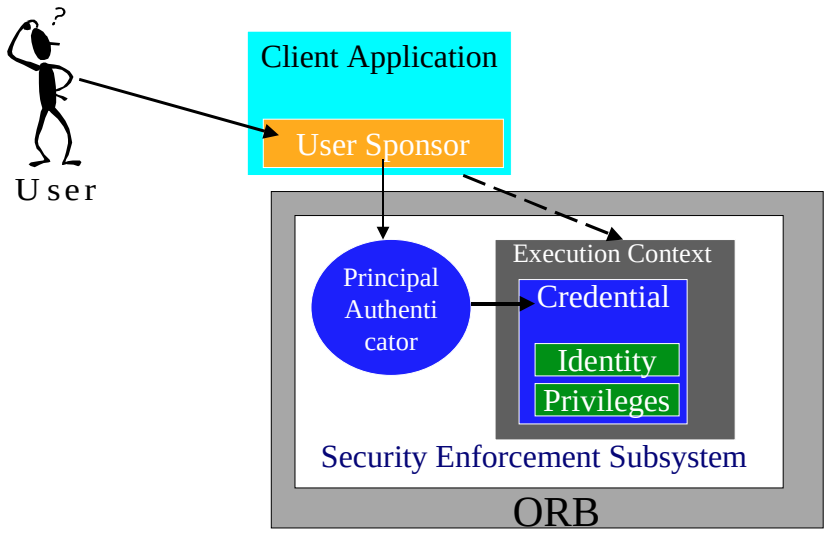


Figure 4-1. Execution Context Creation

specific to the authentication method supported by the concrete implementation of CS. For example, for password-based authentication, it prompts the user for user name and password. For authentication based on smart-cards, it interacts with a smart-card reader and (probably) prompts the user to insert the card in the reader. CS standard does not mandate any particular authentication method. What it does specify is the interface of `PrincipalAuthenticator`. An instance of `PrincipalAuthenticator` conducts the actual authentication and creates `Credentials` object for a new subject. Based on the authentication data it received from a user sponsor and on the underlying security technology (Kerberos, SESAME, or any other capable technology) as well as on any rules it adheres to, `PrincipalAuthenticator` instantiates `Credentials` with various information. The information in `Credentials` constitute the identity of the new subject, which initiates requests on CORBA objects on behalf of the user. Authenticated security attributes are part of the information stored in the `Credentials` object.

Access control and other protection in CS is policy-based. There are several types of policies. One of them is AC policy. Any policy is associated with a domain, which is called *policy domain* in CS terminology. A policy domain is an abstraction that allows security administrators to group objects in groups and assign policies to the groups. Objects that have common security requirements are grouped in the same security policy domains. Domains allow the application of AC policies to security-unaware objects without requiring changes to their implementations or interfaces. Figure 4-2 illustrates the concepts of domains and policies. It shows that a policy domain is associated with a policy. And objects (small circles) are grouped in the domain. They are governed by the policy. Policies of more than one type could be associated with the same policy domain and each object can

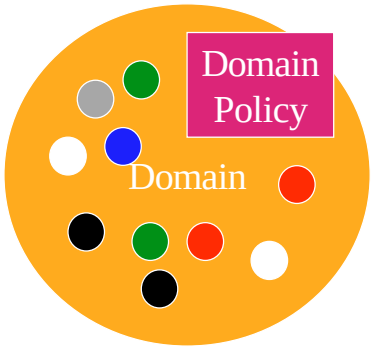


Figure 4-2. Domains and Policies in CORBA Security

belong to more than one policy domain. Domains could be organized in federations, hierarchies or be completely unrelated.

The policy enforcement code uses three sources of information: the policy of the domain(s) to which the target belongs, the information from the client's credentials, and the message itself which specifies target object and the name of the method to be invoked. In the remainder of this section, we discuss in detail the AC mechanisms available in CS.

For illustrating our discussion, we will use Figure 4-3. The concept of a user is absent from the CS AC model. Instead a *principal* represents the user completely. The term *principal* in the CS model is equivalent to *subject* in traditional AC terminology. We will use these two terms interchangeably in this discussion. The notion of a *session* is indistinguishable from the notion of a principal. Thus multiple principals can act on behalf of a single user. They all potentially have different sets of credentials and therefore exist in CS as completely independent entities. Among other data, principal credentials contain security attributes. Hereafter, we understand attribute to mean "security attribute." From the CS AC model point of view, a principal is nothing but an unordered collection of authenticated attributes. An attribute is a four-tuple $a = \{t, a, v, ds\}$ with certain type t, defining authority

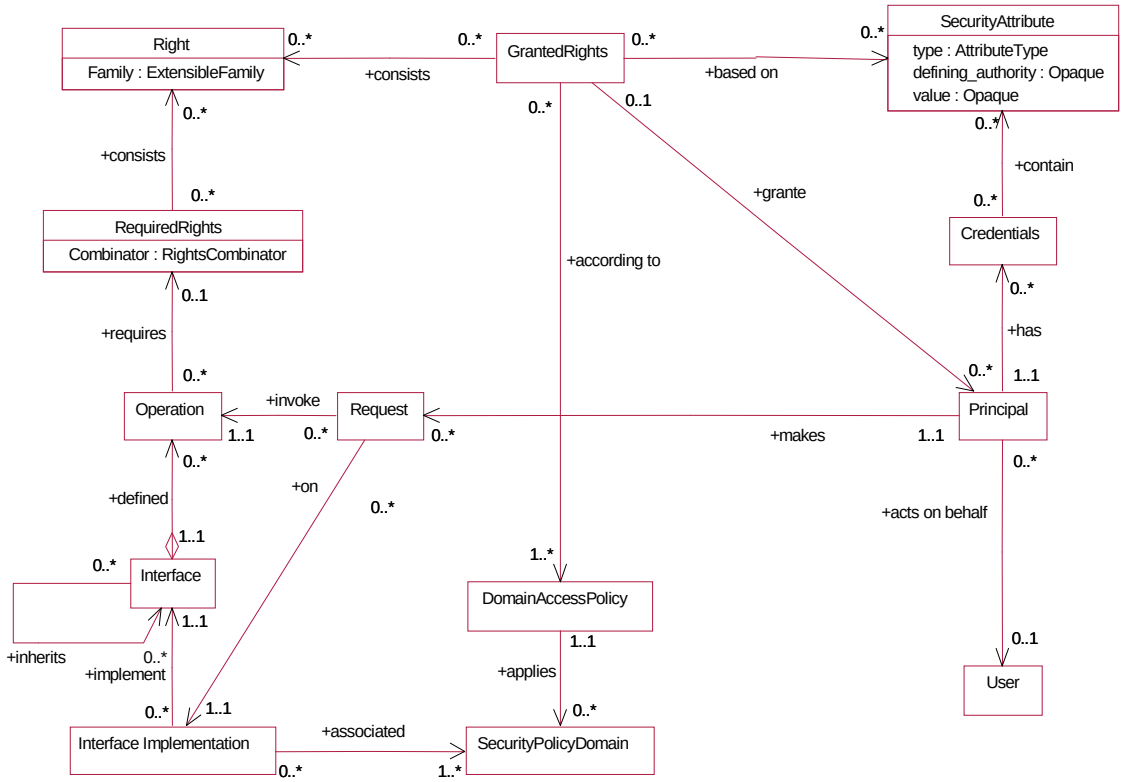


Figure 4-3. Relationships Among the Key Elements of CORBA AC Mechanisms

a, value *v*, and delegation state *ds*. Where $ds \in DS = \{i, d\}$ and state *i* indicates attribute possessed by the immediate invoker, and *d* -- by the intermediate one. Attribute types are partitioned into two families: privilege attributes and identity attributes. The family of privilege attributes enumerates attribute types that identify principal privileges. These types include access identifier, primary and secondary groups the principal is a member of, clearance, capabilities, etc. Identity attributes, if present, provide additional information about the principal. Examples of their types are audit id, accounting id, and non-repudiation id, reflecting the fact that a principal might have various identities used for different purposes. Principal credentials may contain zero or more attributes of the same type. An example of security attributes assigned to authenticated principals is provided in Table 4-1. One of the

standard CORBA attribute types is role. Due to the extensibility of the schema for defining security attributes, an implementation of CS can support attribute types that are not defined by the CORBA Security standard. Although the normative part of CS does not mandate the

Principal	Attributes
p_1	a_1
p_2	a_2, a_6
p_3	a_2, a_3
p_4	a_4, a_5

Table 4-1. Security Attributes Possessed by Authenticated Principals

way the attributes are managed, assignment of such attributes to users is meant to be performed by user administrators.

All a principal does in the CORBA computational model is invoke operations on corresponding objects. In order to make a request one needs to know two things: object reference, which uniquely identifies an object, and operation name. CORBA interfaces can inherit from other interfaces via interface inheritance. An operation name is unique for an interface. Thus, any operation is uniquely identified by its name and by the name of the interface in which it is defined. Here, we use notation $i_k m_n$, to refer to n-th operation on k-th interface.

There is a global set of required rights for each operation. This set, together with a combinator (*all* or *any* rights), defines what rights a principal has to have in order to invoke the operation. Table 4-2 provides an example of required rights for operations on three interfaces i_1, i_2, and i_3. It is assumed that required rights are defined and their semantics are precisely documented by application developers who know the best semantics of each operation. Depending on the access policy (*DomainAccessPolicy*) enforced in a particular

Operations	Required Rights	Combinator	Meaning
i_1m_1	r_1	all	Only a principal who is granted right r_1 can invoke the operation.
i_1m_2	r_1, r_2	any	Any principal who is granted either r_1 or r_2 right can invoke the operation.
i_2m_1	r_2, r_3	all	Only a principal who is granted both r_2 and r_3 rights can invoke the operation.
i_2m_2	r_2, r_3, r_4	all	Only a principal who is granted all r_2, r_3, r_4 rights can invoke the operation.
i_3m_1	r_1, r_2, r_3, r_4	any	Any principal who is granted either of r_1, r_2, r_3, r_4 rights can invoke the operation.

Table 4-2. Required Rights Matrix

AC policy domain, a principal is granted different rights (*GrantedRights*) according to what privilege attributes it has. Each *DomainAccessPolicy* (DAP) object defines what rights are granted for each security attribute. An example of a mapping between principal privilege attributes and granted rights is provided in Table 4-3. Security administrators are responsi-

Attributes	Granted Rights	
	Domain	
	d_1	**d_2**
a_1	r_1	r_2
a_2	-	r_1
a_3	r_2, r_3	-
a_4	r_3	r_1, r_4
a_5	r_1, r_2, r_3	r_2, r_3, r_4
a_6	r_6	r_1

Table 4-3. Granted Rights Per Attribute

ble for defining what rights are granted to what security attributes in what delegation state on domain per domain basis. Whenever a principal attempts an operation invocation, principal's effective rights are computed via operation `AccessPolicy::get_effective_rights()`. CS specification purposefully does not define how the operation combines rights granted through different privilege attribute entries shown in Table 4-3. The specifiers let CS implementers define the operation's internal behavior ([OMG 1996b, p. 122]). A simplest implementation of

`get_effective_rights()` could be such that the set of rights granted to a principal is a union of rights granted to every security attribute the principal has. For our example, we will assume exactly this implementation of the operation. If we use our example of security attributes assigned to principals p_1, p_2, p_3, and p_4 (Table 4-1), and the examples of required (Table 4-2) and granted (Table 4-3) rights, then Table 4-4 shows what rights the

Principal	**Granted Rights**	
	Domains	
	d_1	**d_2**
p_1	r_1	r_2
p_2	r_6	r_1
p_3	r_2, r_3	r_1
p_4	r_1, r_2, r_3	r_1, r_2, r_3, r_4

Table 4-4. Granted Rights Per Principal

principals are granted in each domain. Therefore, the principals can invoke operations as shown in Table 4-5. Note that because principal p_2 is granted only right r_6 in domain d_1, it is not permitted to invoke any operation because right r_6 is not sufficient for invoking any operation according to the Required Rights Matrix (Table 4-2).

Principal	**Permitted Operations**	
	Domains	
	d_1	**d_2**
p_1	i_1m_1, i_1m_2, i_3m_1	i_1m_2, i_3m_1
p_2	-	i_1m_1, i_1m_2, i_3m_1
p_3	i_1m_2, i_3m_1	i_1m_1, i_1m_2, i_3m_1
p_4	i_1m_1, i_1m_2, i_3m_1, i_1m_2, i_2m_1	i_1m_1, i_1m_2, i_3m_1, i_1m_2, i_2m_1, i_2m_2

Table 4-5. Operations Permitted to Principals

4.2.2 CORBA Protection State Configuration

Having informally discussed the CS AC model, we define the protection state configuration of a CORBA system in Defintion 4-1. An implementation of security service compliant with CS is supposed to yield the same access control decision as the one described

Definition 4-1. CORBA System Protection State Configuration

A configuration of a CORBA system protection state is the thirteen-tuple (*A, IM, O, R, D, C, RRM, DS, IDM, GRM,* effective rights, combine, interface operation) interpreted as follows:

- *A* is the set of privilege attributes.
- *IM* is the set of operations uniquely identified by interfaces that they are defined on.
- *O* is a set of distinguishable interface instances.
- *R* is the set of rights.
- *D* is the set of access policy domains.
- $C = \{all, any\}$ is a set of rights combinators.
- *RRM* is the required rights matrix, with a row for every interface operation from IM and two columns. For the first column (Required Rights), we have $[\text{IM, Rights}] \subseteq \text{R}$. For the second column (Combinator), we have $[\text{IM, Combinator}] \in \text{C}$.
- $DS = \{i, d\}$ is a set of delegation states.
- *IDM* is the matrix of domain membership for interface instances with a row for every domain from D and a column for every interface instance from O. We denote the contents of (*D, O*) cell of *IDM* by [*D,O*]. We have $[\text{D, O}] \subseteq \{\text{T, F}\}$,[a] $[\text{d, o}] \equiv T \Rightarrow \text{o} \in \text{d}$.
- *GRM* is the granted rights matrix, with a row for every attribute from A and a column for every access policy domain from D. We denote the contents of the policy domain from D. We denote the contents of the $[A, D] \subseteq R$.
- *effective_rights:* $D \times 2^A \rightarrow 2^R$, a function mapping a set $a_1, a_2, ... a_l$ of privilege attributes (where $\forall i, s.t. 1 < i < l,\ a_i \in A$) in a domain $d_j \in D$ to a set of rights $r_1, r_2, ... r_p$ (where $\forall i, s.t. 1 < i < p,\ r_i \in R$) that are in effect for the given set of attributes.
- *combine:* $(D \rightarrow 2^R) \rightarrow 2^R$ a function mapping sets of rights returned from *effective_rights* for every domain in *D* the interface instance is a member of, to a set of effective rights.
- *interface_operation*: $M \times O \rightarrow IM$ a function mapping an operation name *m* and an interface instance $o \in O$ into an interface operation uniquely identified on the interface, which *o* implements.

a. *T* stands for true and *F* stands for false.

by Algorithm 4-1. Function *effective_rights* looks up *GRM* to obtain granted rights for each attribute in all domains to which object *o* belongs. It combines those rights according to its implementation and returns effective rights for each domain. Results returned from effec-

Algorithm 4-1. Authorization Decision in CORBA

Decide authorization for principal p = {a_1,a_2,...a_n} accessing operation with name *m* on interface instance o where $a_1, ..., a_n \in A$, *m* is a string that names an operation, and $o \in O$.

Require: *interface_operation(m, o)* $\in$ *IM*

1: *DER* $\leftarrow \varnothing$ {Empty an array of rights}
2: **for all** d s.t. IDM[d,o] == T **do**
3: *DER[d]* $\leftarrow$ *effective_rights(d, p)*
4: **end for**
5: *ER[d]* $\leftarrow combine(DER)$ {Combine effective rights into one set}
6: *i* $\leftarrow$ *interface_operation(m, o)*
7: **if** *RRM*[*i, Combinator*] == *any* **then**
{Any right is required}
8: **for all** r in *RRM*[*i, Rights*] **do**
9: if $r \in ER$ **then**
10: **return** T
11: **end if**
12: **end for**
13: **return** F
14: **else**
{All Rights are required}
15: **for all** r in *RRM*[*i, Rights*] **do**
16: **if** $r \notin ER$ **then**
17: **return** F
18: **end if**
19: **end for**
20: **return** T
21: **end if**

tive rights serve as input parameters for the function *combine*. The latter combines them according to its implementation. Rights returned by *combine* are checked against *RRM*. If the match succeeds, then access is granted. Otherwise, access is denied.

Table 4-5 shows what operations can be invoked by the principals from our example. For each domain, an access matrix from [Lampson 1971], such as in Table 4-6, could be constructed.

Principal	**Objects**		
	***i*₁**	***i*₂**	***i*₃**
p_1	i_1m_2	-	i_3m_1
p_2	i_1m_1, i_1m_2	-	i_3m_1
p_3	i_1m_1, i_1m_2	-	i_3m_1
p_4	i_1m_1, i_1m_2	i_2m_1, i_2m_2	i_3m_1

Table 4-6. Operations Permitted to Principals

Three general observations are worth noting for an access matrix constructed for any CS system. First, subjects cannot be objects, i.e. the CORBA access control does not have the concept of operations on principals. It only has the concept of operations on interfaces, which are objects according to the terminology of the access matrix [Lampson 1971]. Second, since $i_k m_p \equiv i_l m_q \Leftrightarrow k \equiv l \wedge p \equiv q$ (i.e. just $p \equiv q$ is not enough for $i_k m_p \equiv i_l m_q$), as in Table 4-6, the semantics of operations in a general case might be different. Thus, for each subject *s* and object *o*, the content of cell [*s*,*o*] is specific to the object, i.e. no operations permitted on one object could be permitted on another object because operations are semantically different for every interface unless interfaces are related via inheritance. Third, all implementations of the same interface in a given access policy domain are represented by the same object in the access matrix; therefore, implementations of the same

interface are indistinguishable from the access control point of view. This is one of the reasons policy domains are important in the CORBA access control model.

4.3 Support of RBAC by the CORBA

4.3.1 Access Control Model

Among the four RBAC reference models defined by Sandhu et al. [Sandhu 1996], $RBAC_0$ is the base model. It requires only that a system has notions of users, roles, permissions and sessions. There are no constraints on the assignment of permissions to roles and users to roles. $RBAC_1$ has hierarchies of roles in addition to everything $RBAC_0$ has. $RBAC_2$ has constraints on the assignment of users to roles and permissions to roles in addition to everything $RBAC_0$ has. $RBAC_3$ combines $RBAC_1$ and $RBAC_2$. In this section, we define $RBAC_0$ and $RBAC_1$ using the language of Defintion 4-1 for CORBA protection state configuration. This will help us show the correctness of our approach to configuring a CORBA system for supporting various RBAC models. But first we introduce the original RBAC definitions.

4.3.2 Original Definitions of RBAC models

According to the RBAC model, each session is a mapping of one user to possibly many roles. When a user establishes a session, he or she activates a subset of roles assigned to the user by the user administrator(s). The permissions available to the user are the union of permissions from all roles activated in that session. RBAC treats permissions as uninterpreted symbols because their semantics is implementation and system dependent.

Definition 4-2. $RBAC_0$

The $RBAC_0$ model has the following components:

- *U, R, P,* and *S* (users, roles, permissions and sessions respectively),
- $PA \subseteq P \times R$, a many-to-many permission to role assignment relation,
- $UA \subseteq U \times R$, a many-to-many user to role assignment relation,
- $user: S \rightarrow U$, a function mapping each session si to the single user $user(s_i)$ (constant for the session's lifetime), and
- $roles : S \rightarrow 2^R$ a function mapping each session si to a set of $roles(s_i) \subseteq \{r \mid (user(s_i),r) \in UA\}$ (which can change with time) and session s_i has the permissions $\bigcup_{r \in roles(s_i)} \{p \mid ((p,r) \in PA)\}$

Definition 4-3. $RBAC_1$

The $RBAC_1$ model has the following components:

- *U, R, P, S, PA, UA,* and user are unchanged from $RBAC_0$,
- $RH \subseteq R \times R$ is a partial order on R called the role hierarchy or role dominance relation, also written as $\geq$, and
- $roles : S \rightarrow 2^R$ is modified from $RBAC_0$ to require $roles(s_i) \subseteq \{r \mid (\exists r' \geq r)[(users(s_i),r') \in UA]\}$ and session s_i has granted rights $\bigcup_{a \in roles(p_i)} \{r \mid (\exists a'' \leq a)[(r,a'') \in PA]\}$ (which can change with time) and session s_i has the permissions $\bigcup_{r \in roles(s_i)} \{p \mid (\exists r'' \leq r)[(p,r'') \in PA]\}$

We reproduce definitions of $RBAC_0$ (Defintion 4-2) and $RBAC_1$ (Defintion 4-3) models from [Sandhu 1996] to help the reader in understanding the rest of the chapter.

4.3.3 $RBAC_0$: Base Model

For the base model $RBAC_0$, the four sets of identities are represented in CS as follows:[1] users in RBAC map to users in CS; roles are represented by set *A* of privilege attributes of type *role*; permissions are equivalent to the set of rights *R* in CS; sessions are equivalent to principals which are nothing but sets of security attributes, from the CS AC point of view. $RBAC_0$ in the language of CS is formally defined in Defintion 4-4.

Definition 4-4. $RBAC_0$ in the Language of CORBA Security

- U, A, R, P (users, attributes of type role, rights, and principals, respectively)
- $PA \subseteq R \times A$ a many-to-many assignment of granted rights to security attributes of type *role* relation.
- $UA \subseteq U \times A$ a many-to-many user to security attributes of type role assignment relation.
- *user:* $P \rightarrow U$, a function mapping each principal p_i to the single user *user(p_i)*, constant for the principal lifetime, and
- *roles:* $P \rightarrow 2^A$ a function mapping each principal pi to a set of privilege attributes of type role *roles(p_i)* $roles(p_i) \subseteq \{a \mid ((user(p_i),a) \in A)\}$ and principal p_i has the granted rights $\bigcup_{a \in roles(p_i)} \{r \mid ((r,a) \in PA)\}$.

1. We do not mention CS AC domains because, as it will be shown in the example below, RBAC models can be supported in CORBA using a single domain.

It is easy to see that the definition describes a system compliant with the $RBAC_0$ definition provided in [Sandhu 1996]. Given the definition, we will show how a CORBA protection system specified by a configuration language from Defintion 4-1 could be used to implement a security system compliant to this definition of $RBAC_0$. *PA* relation is specified by the granted rights matrix *GRM*. *UA* relation is managed by user administrators in CS that define what values of attributes of type role are assigned to users. However such management functionality is beyond the scope of CS specification, which means that functionality defined by *UA* relation is implementation-specific. An implementation of `PrincipalAuthenticator`[1] initializes new principal credentials with security attributes according to *UA*. An example is provided in Table 4-1, where attributes a_1 through a_6 have the type role. The value of the principal privilege attribute of the type *AccessId* is equivalent to the return value from the function *user*. An implementation of `PrincipalAuthenticator` should initialize principal credentials according to the function *roles*. Since a user in $RBAC_0$ can activate any subset of roles to which the user is assigned, implementation of *UA* ensures implementation of $RBAC_0$. Thus, we have shown that all relations, functions and sets specified in Defintion 4-4 can be directly supported by CS-compliant implementations. In order for a CS implementation to support $RBAC_0$ it should:

1. comply with CS standard, and

2. provide a means to administrate user-to-role assignment relation *UA*, and

1. As it was described in Section 3.1.5, a `PrincipalAuthenticator` conducts the actual authentication and creates Credentials object for a new principal.

3. provide a means for users to select through *user sponsor* a set of roles with which they would like to activate the new principal, and

4. implement `PrincipalAuthenticator` which creates principal credentials containing privilege attributes of type *role* according to relation *UA*, and

5. implement `PrincipalAuthenticator` which creates principal credentials containing one and only one privilege attribute of type *AccessId*.

A straightforward implementation of $RBAC_0$ in CS would be the one that uses privilege attributes of only type *role* for constructing granted rights tables, such as Table 4-3.

4.3.4 RBAC1: Role Hierarchies

$RBAC_1$ is $RBAC_0$ with role hierarchies. $RBAC_1$ in the language of CS is formally defined in Defintion 4-5.

Definition 4-5. $RBAC_1$ in the Language of CORBA Protection System

- *U, A, R, P, PA, U A* and user are unchanged from $RBAC_0$.
- $RH \subseteq A \times A$ is a partial order on *R* called the role hierarchy, written as $\geq$. It is the same as in [Sandhu 1996].
- $roles : P \rightarrow 2^A$ is modified from $RBAC_0$ to require $roles(p_i) \subseteq \{a \mid (\exists a' \geq a)[(users(p_i),a') \in UA]\}$ and principal p_i has granted rights $\bigcup_{a \in roles(p_i)} \{r \mid (\exists a'' \leq a)[(r,a'') \in PA]\}$.

Function *roles* is to be implemented and enforced by a `PrincipalAuthenticator` (Figure 4-1). A user provides to a *user sponsor* a set of roles with which they want the

principal to be activated. The `PrincipalAuthenticator`, during the authentication with the *user sponsor*, creates new credentials of the principal. The credentials have roles, requested by user, provided that they satisfy the definition of function *roles* for $RBAC_1$.

A valid implementation of $RBAC_1$ could be one that allows a user to specify any role junior to those of which the user is a member. In this case, an implementation of `PrincipalAuthenticator` activates all roles which are junior to the specified role.

In order for a CS implementation to support $RBAC_1$ it should:

1. implement $RBAC_0$, and
2. provide a means to administrate the role hierarchy relation *RH*, and
3. implement `PrincipalAuthenticator` which creates principal credentials containing privilege attributes of the type *role* according to relations *UA* and *RH*, as well as function *roles*.

4.3.5 $RBAC_2$: Constraints

Constraints in RBAC are predicates that apply to *UA* and *PA* relations, as well as to functions *user* and *roles* [Sandhu 1996]. Constraints on *UA* relation are to be enforced by an implementation of user administrator tools. Constraints on functions *user* and *roles* are the responsibility of `PrincipalAuthenticator` implementation. Constraints on *PA* relation are to be enforced by an implementation of security administrator tools.

In order for a CS implementation to support $RBAC_2$ it should:

1. implement $RBAC_0$, and

2. implement the support of constraints on *UA* relation by user administrator tools, and

3. implement `PrincipalAuthenticator` with the support of constraints on functions *user* and *roles*, and

4. enable enforcement of constraints on *PA* relation by security administrator tools.

4.3.6 $RBAC_3$: $RBAC_1$ + $RBAC_2$

$RBAC_3$ is a combination of $RBAC_1$ and $RBAC_2$ along with possibly additional constrains on the role hierarchy. It can be implemented in CS as well. Obviously, in order for a CS implementation to support $RBAC_3$ it should:

1. implement $RBAC_1$, and

2. implement $RBAC_2$, and

3. implement possible additional constrains on the role hierarchy.

The requirements for the support of $RBAC_1$ and $RBAC_2$ by CORBA Security service implementation have already been discussed. The implementation of additional static constrains on the $RBAC_1$ role hierarchy is to be done by user administrator tools. For the support of dynamic constraints, additional functionality in the implementation of `PrincipalAuthenticator` is required, in addition to the administrator tools.

4.4 Examples

To illustrate the points made in the previous chapter, we describe a protection state (defined by Defintion 4-4) of a CORBA system that implements an example role hierarchy. We show how a CORBA-based distributed system could be configured to support $RBAC_1$

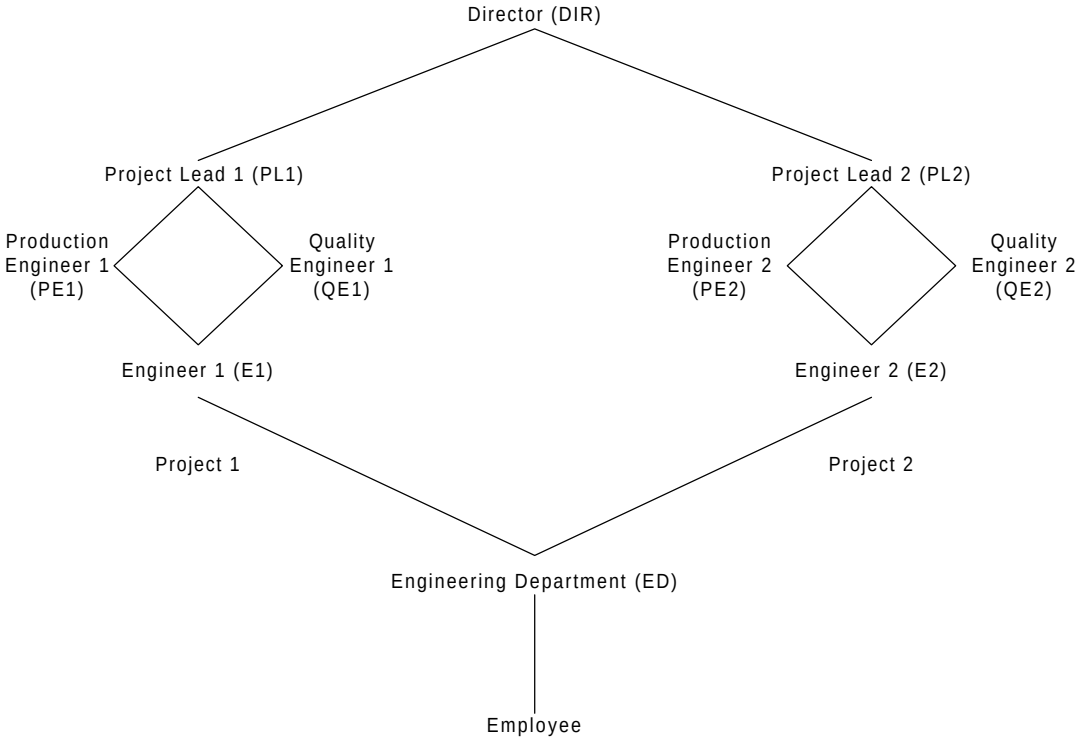


Figure 4-4. An Example Role Hierarchy (from [Sandhu 1998b])

with an example hierarchy from [Sandhu 1998b] shown on Figure 4-4 and to protect access to the implementations of CORBA interfaces shown in Figures 4-5 and 4-6. In RBAC role hierarchies, the convention is to depict junior roles (with less permissions) at the bottom, and senior roles (with permissions inherited from the junior ones in addition to the new per-

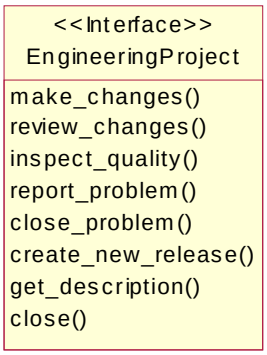


Figure 4-5. EngineeringProject Interface

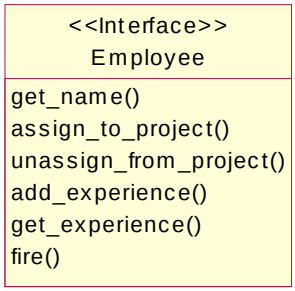


Figure 4-6. Employee Interface

missions) at the top. The following access control policies describe what actions are allowed. All other actions are denied.

Authorization Policies

1. Anyone can look up an employee's name and experience.
2. Everyone in the engineering department can get a description of and report problems regarding any project.
3. Engineers, assigned to projects, can make changes and review changes related to their projects.
4. Quality engineers can inspect the quality of projects they are assigned to.
5. Production engineers can create new releases.
6. Project leaders can close problems.
7. The director can manage employees (assign/un-assign them to/from projects, add new records to their experience, and fire) and close engineering projects.

We define that function *effective_rights* returns a union of granted rights per attribute, and *combine* returns a union of rights granted in each domain.

The intent of CORBA access policy domains is somewhat confusing. To help in understanding it, we provide two solutions for enforcing these policies. The first uses a single access policy domain. The second uses multiple domains.

4.4.1 Single Access Policy Domain Solution

In order to implement the role hierarchy in CS without using access policy domains, we introduce two new interfaces `EngineeringProject1` and `EngineeringProject2`, as shown in Figure 4-7. The following system protection

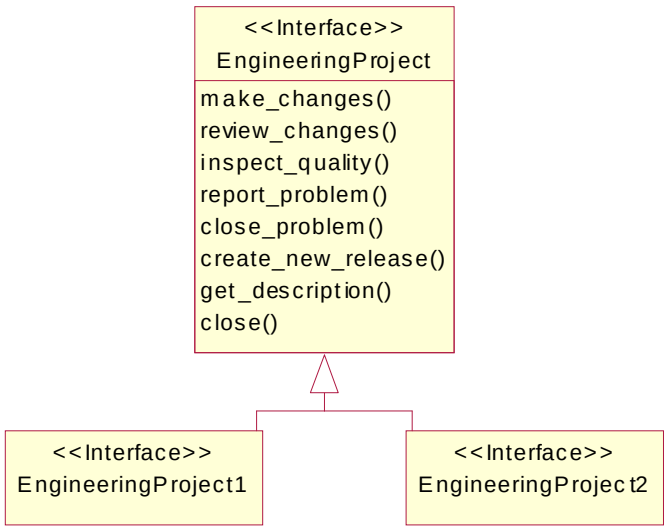


Figure 4-7. `EngineeringProject` Interface Hierarchy

state configuration could be used:

- A = {e, ed, e1, e2, pe1,pe2, qe1, qe2, pl1, pl2, dir}. All these attributes have type *role*.

- IM = {Employee::get_name, Employee::assign_to_project, Employee::unassign_from_project, Employee::add_experience, Employee::get_experience, Employee::fire, EngineeringProject1::inspect_quality, EngineeringProject1::make_changes, EngineeringProject1::report_problem, EngineeringProject1::review_changes, EngineeringProject1::close, EngineeringProject1::close_problem, EngineeringProject1::create_new_release, EngineeringProject1::get_description, EngineeringProject2::inspect_quality, EngineeringProject2::make_changes, EngineeringProject2::report_problem, EngineeringProject2::review_changes, EngineeringProject2::close, EngineeringProject2::close_problem, EngineeringProject2::create_new_release, EngineeringProject2::get_description}.

  We do not use any implementations of interface `EngineeringProject`. Only derived interfaces are used.

- *O* = {e, ed, e1, e2, pe1,pe2, qe1, qe2, pl1, pl2, dir, prj1, prj2}. prj1 is an instance of `EngineeringProject1`, and prj2 is an instance of `EngineeringProject2`. All other elements of *O* are instances of interface `Employee`.

- *R* = {gn, atp, ufp, ae, ge, f, mc1, rc1, iq1, rp1, cp1, cnr1, gd1, c1, mc2, rc2, iq2, rp2, cp2, cnr2, gd2, c2}[1]

- *D* = {d_1}

- *C* = {all} - we use only one combinator.

1. We used first letters of each operation to create a corresponding right.

- *RRM* is shown in Table 4-7. We omitted column with rights combinators because required rights for all operations have the same combinator - "all."[1]
- *DS* = {i, d}
- In the *IDM*, all interface instances are the members of the only access policy domain.
- *GRM* is shown in Table 4-8.

Operations	**Rights**
Employee::get_name	gn
Employee::assign_to_project	atp
Employee::unassign_from_project	ufp
Employee::add_experience	ae
Employee::get_experience	ge
Employee::fire	f
EngineeringProject1::get_description	gd1
EngineeringProject1::inspect_quality	iq1
EngineeringProject1::make_changes	mc1
EngineeringProject1::review_changes	rc1
EngineeringProject1::report_problem	rp1
EngineeringProject1::close_problem	cp1
EngineeringProject1::create_new_release	cnr1
EngineeringProject1::close	c1
EngineeringProject2::get_description	gd2
EngineeringProject2::inspect_quality	iq2
EngineeringProject2::make_changes	mc2
EngineeringProject2::review_changes	rc2
EngineeringProject2::report_problem	rp2
EngineeringProject2::close_problem	cp2
EngineeringProject2::create_new_release	cnr2
EngineeringProject2::close	c2

Table 4-7. Required Rights Matrix for Single Domain Solution

1. We could have used "any" as well. When an operation's required rights set consists of only one right, the effect of either combinator is the same.

Privilege Attribute	Granted Rights
e	gn, ge
ed	gd1, gd2, rp1, rp2
e1	mc1, rc1
pe1	cnr1
qe1	iq1
pl1	cp1
e2	mc2, rc2
pe2	cnr1
qe2	iq1
pl2	cp1
dir	atp, ufp, ae, f, c1, c2

Table 4-8. Granted Rights Matrix for Single Domain Solution

- $effective\_rights(d_j, a_1, a_2, \ldots, a_l) \subseteq \bigcup_{a_i,\ 1 \le i \le l} \{r \mid r \in GRM[a_i, d_j]\}$ -- union of granted rights per attribute.

- $combine(r_{1,d_1}, r_{2,d_1}, \ldots, r_{l,d_1}, \ldots, r_{1,d_p}, r_{2,d_p}, \ldots, r_{m,d_p}) \subseteq$

  $\bigcup_{in\ each\ domai\ d} \left\{ r \mid r \in \left\{ r_{1,d_1}, \ldots, r_{m,d_p} \right\} \right\}$ -- union of rights granted in each domain.

The CORBA protection system configuration described above allows enforcement of the sample policies listed on page 118. For example, a lead of project 1 with role pl1 activated is able to invoke operations *get_name* and *get_experience* on all implementations of interface `Employee` as well as all but *close* operations on all implementations of interface `EngineeringProject1`.

From observing the configuration of the CORBA protection system in this solution, significant administrative overhead could be noticed. The overhead is due to the gratuitous use of a separate interface (`EngineeringProject(1,2)`) per project . This is because we purposefully limited our solution to a single access policy domain. It is shown below

how the unnecessary redundancy of the protection system configuration data is eliminated by using multiple access policy domains and a hierarchy of such domains.

4.4.2 Multi-domain Solution

Once we have an access policy domain per project, we can go back to using one `EngineeringProject` interface for all projects. We also take advantage of the CS capability to compose domains in various hierarchies. We choose a limited and easy to understand tree-like hierarchy shown in Figure 4-8. The following configuration of a system protection state could be used:

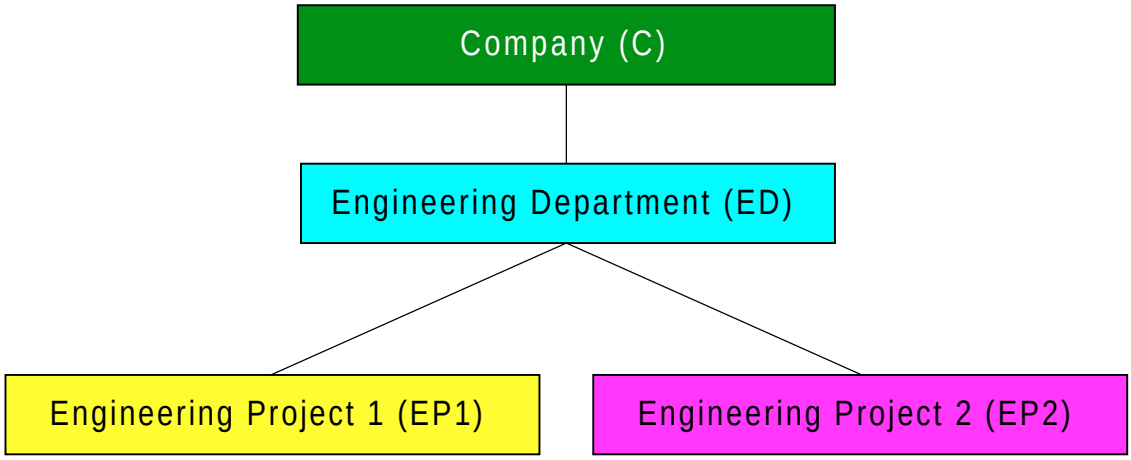


Figure 4-8. Domain Hierarchy for Multi-domain Solution

- *A*, *O*, *C*, *DS*, *effective_rights*, and *combine* are the same as in the single domain solution.
- *IM* = {Employee::get_name, Employee::assign_to_project, Employee::unassign_from_project, Employee::add_experience, Employee::get_experience, Employee::fire, EngineeringProject::inspect_quality, EngineeringProject::make_changes, EngineeringProject::report_problem,

EngineeringProject::review_changes, EngineeringProject::close, EngineeringProject::close_problem, EngineeringProject::create_new_release, EngineeringProject::get_description}.

- *R* = {gn, atp, ufp, ae, ge, f, mc, rc, iq, rp, cp, cnr, gd, c}.
- *D* = {C, ED, EP1, EP2}
- *RRM* is shown in Table 4-9. It is the same as in Table 4-7 except one interface `Engineering-Project` is used instead of two identical interfaces with different names.

Operations	**Rights**
Employee::get_name	gn
Employee::assign_to_project	atp
Employee::unassign_from_project	ufp
Employee::add_experience	ae
Employee::get_experience	ge
Employee::fire	f
EngineeringProject::get_description	gd
EngineeringProject::inspect_quality	iq
EngineeringProject::make_changes	mc
EngineeringProject::review_changes	rc
EngineeringProject::report_problem	rp
EngineeringProject::close_problem	cp
EngineeringProject::create_new_release	cnr
EngineeringProject::close	c

Table 4-9. Required Rights Matrix for Multi-domain Solution

- *IDM* is shown in Table 4-10. As illustrated in Figure 4-9, if an object belongs to a child domain, according to the domain hierarchy shown in Figure 4-8, then it is also a member of all the parental domains.
- *GRM* is shown in Table 4-11.

Interface Instance	Domains			
	C	ED	EP1	EP2
e	X			
ed	X	X		
e1	X	X	X	
pe1	X	X	X	
qe1	X	X	X	
pl1	X	X	X	
e2	X	X		X
pe2	X	X		X
qe2	X	X		X
pl2	X	X		X
dir	X			
prj1	X	X	X	
prj2	X	X		X

Table 4-10. Interface Instance Domain Membership Matrix (IDM) for Multi-domain Solution

The CORBA protection system configuration described above allows enforcement of the same policies as the configuration in the solution for a single domain. This time, there is no need either in having separate `EngineeringProject(1,2)` interfaces per project or in having redundant rights. In addition, *RRM* and *GRM* are more comprehensible.

Due to the hierarchy structure of the access policy domains, the described system can also support more flexible policies. For example, the *GRM* in Table 4-11, in addition to the sample policies already described on page 118, supports a policy which allows project leaders to add experience (right ae) to the records of the employees working under supervision of the leaders. In order to enable it, whenever an employee is assigned to a project (we assume each employee works on one project at a time) an interface implementation representing the employee is moved to access policy domain of the corresponding project. Also,

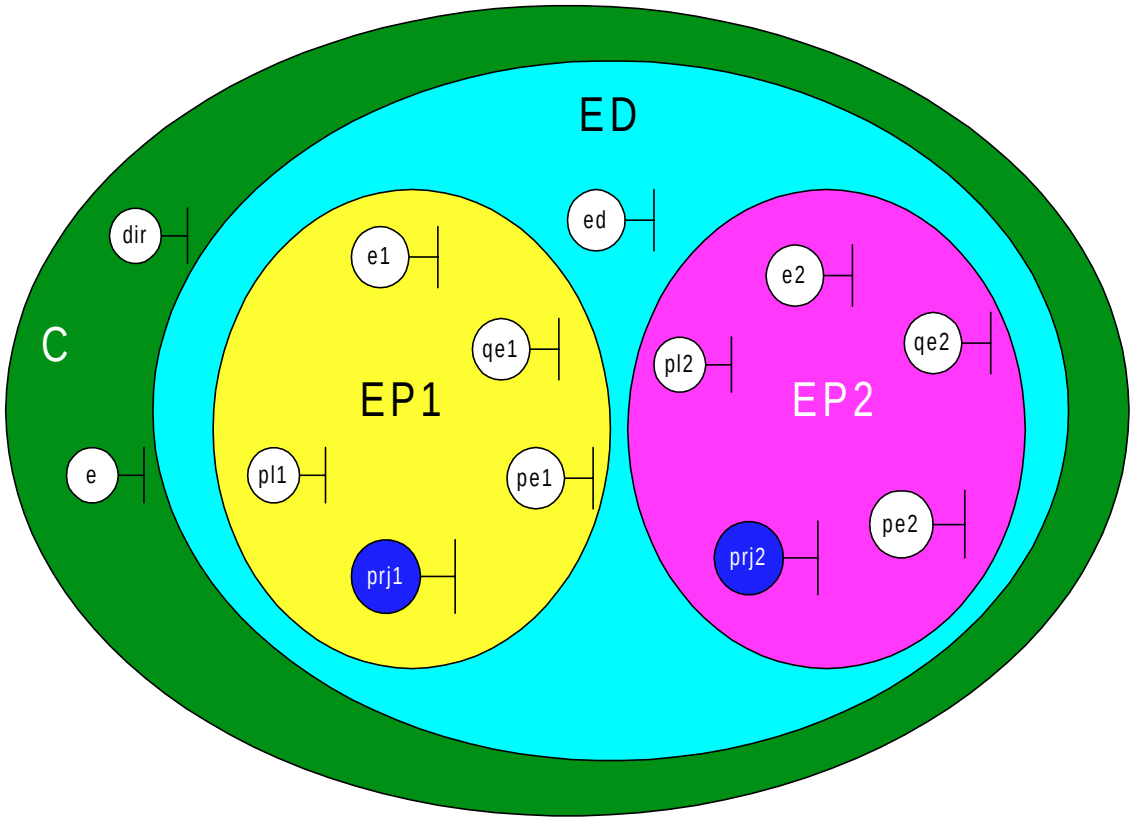


Figure 4-9. Interface Instance Domain Membership

Privilege Attribute	Granted Rights			
	Domains			
	C	ED	EP1	EP2
e	gn	ge	-	-
ed	-	gd, rp	-	-
e1	-	-	mc, rc	-
pe1	-	-	cnr	-
qe1	-	-	iq	-
pl1	-	-	cp, ae	-
e2	-	-	-	mc, rc
pe2	-	-	-	cnr
qe2	-	-	-	iq
pl2	-	-	-	cp, ae
dir	atp, ufp, ae, f, c	-	-	-

Table 4-11. Granted Rights Matrix for Multi-domain Solution

the *GRM* enforces finer grain policy which allows only colleagues from the same department to look up employee experience (right ge) .

4.5 Conclusions

The understanding of middleware AC mechanisms is critical for protecting resources of enterprise applications. In this chapter we not only described in details AC mechanism of one of the most capable middleware security technologies -- CORBA Security -- but also defined a configuration of the CORBA protection system. Using the configuration definition, we suggested an algorithm which formally specifies the semantics of authorization decisions in CS.

We defined $RBAC_0$ and $RBAC_1$ models in the language of CS and described how $RBAC_0$-$RBAC_3$ could be implemented using CS. We discussed what functionality needs to be implemented, besides compliance with CS standard, in order to support RBAC. We illustrated the discussion with a single access policy domain and multi-domain examples of the CS protection system configuration, which supports a sample role hierarchy and access policies.

Implementations compliant with the CS specification can support $RBAC_0$-$RBAC_3$. However, additional functionality not specified by CS is required. Implementations of `PrincipalAuthenticator` interface and *user sponsor* need to support roles and their hierarchies ($RBAC_1$). To support constraints ($RBAC_2$), a `PrincipalAuthenticator` has to enforce them. Tools to administer user-to-role and role-to-rights relations are also required.

This chapter develops a framework for implementing as well as for assessing implementations of RBAC models using CS. It provides directions for CS developers to realize RBAC in their systems and gives criteria to users for selecting such implementations that support models from the $RBAC_0$-$RBAC_3$ family. This work advances the understanding of the CORBA AC mechanism's capabilities and by this maximizes its utility which is vital to the use of middleware in protecting application resources.

Although RBAC is shown to supersede major AC models, its capabilities are limited and there could be authorization policies that would be challenging to model with it. Also, the granularity of the CORBA AC mechanism is still limited to the level of interface operation. This is why we believe that the use of RBAC and CORBA does not address the needs of all application domains. The rest of this dissertation discusses the second part of our approach, which addresses those cases when the RBAC model and CORBA mechanism are inadequate.

5 Resource Access Decision Service

In the previous chapters we stated the problem of controlling access to the resources of enterprise distributed applications and reviewed available technologies along with related work. For those application domains where authorization policies can be supported by RBAC and the granularity of the CORBA AC mechanism is sufficient, the framework for implementing RBAC models using CORBA Security developed in last chapter could be an adequate solution. But what to do with the applications whose AC needs cannot completely be addressed by either the RBAC model or CORBA Security? In this chapter, we introduce an approach which meets the requirements of other applications -- an architecture for resource access decision (RAD) service. Furthermore, we demonstrate its utility on examples with complex access control policies. Some sections of the chapter are based on the material from [Beznosov 1999b].

RAD defines a conceptual architecture that encapsulates authorization logic in an authorization service which is external to the application and is also independent of the specific security models and policies. Such an architecture not only significantly simplifies both application and security system development but also allows organizations to uniformly manage and enforce their security policies.

The RAD approach addresses most other issues important for protecting application resources in enterprise distributed applications. It is possible to use as many types of subject

security attributes for authorization decisions as the underlying authentication technology provides. The service architecture allows the use of information obtained from work-flow systems and other sources, thus supporting policies specific to the application domain. It also enables the use of application-specific information in AC decisions. Due to the encapsulation of authorization logic into a separate service, which can be implemented as a network server, consistency of AC policies enforcement across applications can be easily achieved. In addition, the architecture supports the multi-policy authorization model, and it enables security administrators and application developers to maintain a clear separation of responsibilities. To achieve these benefits, our design requires application-level enforcement of authorization decisions and assumes agreement on the semantics of resource names between the application developer and the owner.

RAD architecture is mostly independent of the underlying security technology, although the current design takes advantage of the CORBA-compliant security infrastructure and compliments it with the capability of more sophisticated authorization. Note that it is by no means a replacement or substitution of standard CORBA Security service [OMG 1996b]. Still, the RAD approach can be applied to most distributed computing environments.

Moreover, we show that the decoupling of authorization logic from application can be done without complicated interactions between an application and the authorization service and without significant communication overhead. Factors specific to the application domain can be supported by authorization systems using the traditional access matrix [Lampson 1971] as an underlying implementation.

5.1 RAD Architecture

The main objective of RAD is to decouple application-level authorization logic from application logic. As discussed above, the finest granularity level of AC provided by the main middleware technologies is at the level of operations on middleware objects. The authorization service is to make decisions for access to those information and computational resources that are not first class objects or their operations. Thus, the service complements middleware AC mechanisms. It relies on and uses the middleware security environment for secure authenticated communications (i.e. message authenticity, confidentiality and integrity protection) between the service and the applications as well as among the service components. It also assumes that the underlying security provides a means for an application to obtain security attributes of the accessing subject. As we showed in Chapter 3, these assumptions are valid for most middleware security technologies.

5.1.1 Interface Between Application Systems and RAD Service

The RAD approach is a representative example of authorization services direction described in Section 3.2.3. Like most of these services, RAD provides authorization decisions to an application system (AS). Authorization logic is encapsulated into RAD service external to the application, which is traditionally part of an application program. Since the service can be logically centralized, the approach allows applications to enforce AC according to the same enterprise-wide set of authorization policies thus naturally enabling policy consistency. In our approach, the authorization decision is obtained after the method on the object is invoked. Hence, an application can exercise access control of any granularity level by associating a resource name with protected elements of any size and semantics.

The flow of interactions between application client, application system and an instance of authorization service is depicted in Figure 5-1. The sequence of the interaction is as follows:

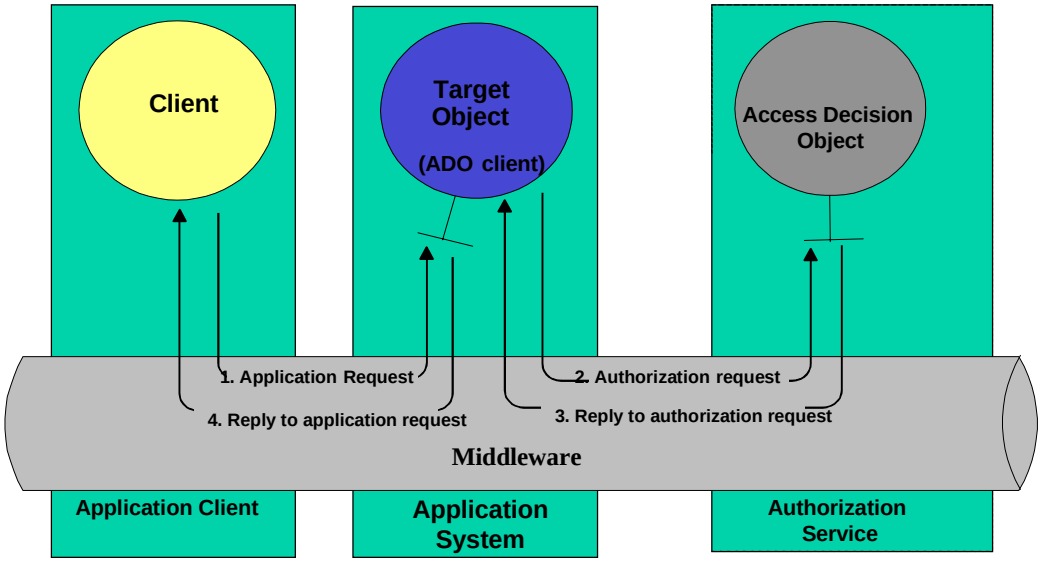


Figure 5-1. Interactions among Client, Application System, and RAD Service

1. A client of the application system invokes an operation on the application.

2. While processing the invocation, the application requires an authorization decision from the authorization service.

3. The service makes a decision, which is returned to the application.

4. The application enforces the decision. If access was granted by the authorization service, the application returns the expected results of the invocation. Otherwise, it either returns partial results or raises an exception.

Simple interfaces between the application and the authorization service are used. An application developer only needs to program a single invocation on the authorization ser-

vice in order to obtain a decision. Each authorization request consists of client subject security attributes, the name of the resource to be accessed, and the name of the operation to be performed on the resource. The security attributes of the invoking subject are supposed to be obtained by the application from the middleware security infrastructure. The application is expected to compute the resource and operation names as part of its application logic. For each authorization request, it receives back a binary (yes/no) decision. An application obtains an authorization decision only from one instance of RAD. It is the contract between the application and its enterprise environment to request an authorization decision and to enforce it.

A nominal amount of data is passed between the application and the authorization service in order to make authorization decisions. When making an authorization request, an application passes the following three parameters: a sequence of name-value pairs representing a name of the resource to be accessed; name of the access operation (e.g. “create,” “read,” “write,” “use,” “delete”); and authenticated security attributes of the subject on behalf of which the client is requesting access to the named resource.

Security attributes here are regular attributes of the current user session. Of the the parameters passed by the client, the first two (resource name and access type) are most worthy of discussion. We introduce an abstraction called “protected resource name” or just “resource name,” used to abstract application-dependent semantics of entities, the access to which is controlled by the application. A resource name can be associated with any valuable asset of the application owner, the access to which is controlled according to the owner's interests. For example, electronic patient medical and billing records in a hospital

are usually its valuable assets. The hospital administration is interested in controlling access to the records due to various legal, financial and other reasons. Therefore, the hospital administration considers such records as protected resources. Moreover, different information in those records counts as different resources, examples of which can be records from different visits or episodes for one patient. At the same time, a resource name can be associated with less tangible assets, such as computer system resources, including CPU time, file descriptors, sockets, etc. The RAD service does not attempt to interpret the semantics of the resource name. We will show in the discussion of the RAD design that it uses the resource name only to obtain additional security attributes and to look up a set of policies governing access to the resource associated by an application system with the resource name.

Access operation abstracts the semantics of access to resource(s) associated with resource name. An application may manipulate patient records on behalf of different care givers, or may provide different hierarchies of menus to different lab technicians. In either case, it is up to the application system developers and the enterprise security administrators to agree on the semantics of the operation name used for each access. RAD does not interpret the semantics of access operation as shown in the description of the RAD design.

A system can communicate application-specific information to RAD service by encoding it in resource and/or operation names. For example, withdrawal of $500 from a bank account can be represented as an operation with the name “withdrawal:$500,” and the resource name carrying the account number. Simple and yet very generic data structures for

operation (arbitrary string) and resource name (a list of string name-value pairs) have good expressive capabilities for this task.

Before an application requests a RAD server for an authorization decision, it is supposed to identify what the resource name and the access operation name are associated with servicing the client request. There is no particular algorithm defined for performing such an association because for every application, or at least for every application domain, the method of associating protected entities with abstract resource names can be different.

Our approach is very similar to most solutions based on authorization services in the way the client, AS and the RAD server interact, but it is different in the internal composition of its elements.

5.1.2 Logical Composition of RAD

RAD architecture aims to enable implementation of its components by various vendors due to the diversity in the requirements to AC policies, performance, scalability and other system properties from different government and commercial markets. Components of the following types comprise a RAD service (Figure 5-2): The `AccessDecisionObject` (ADO) serves as the interface to RAD clients and coordinates the interactions between the RAD components. Zero or more `PolicyEvaluators` (PEs) perform evaluation decisions based on the AC policies governing the access to protected resources. The `DecisionCombinator` (DC) combines the results of the evaluations made by potentially multiple PEs into a final authorization decision by applying certain combination policy. The `PolicyEvaluatorLocator` (PEL), for a given access request to a protected resource, keeps track of and provides references to a DC and potentially several PEs, which

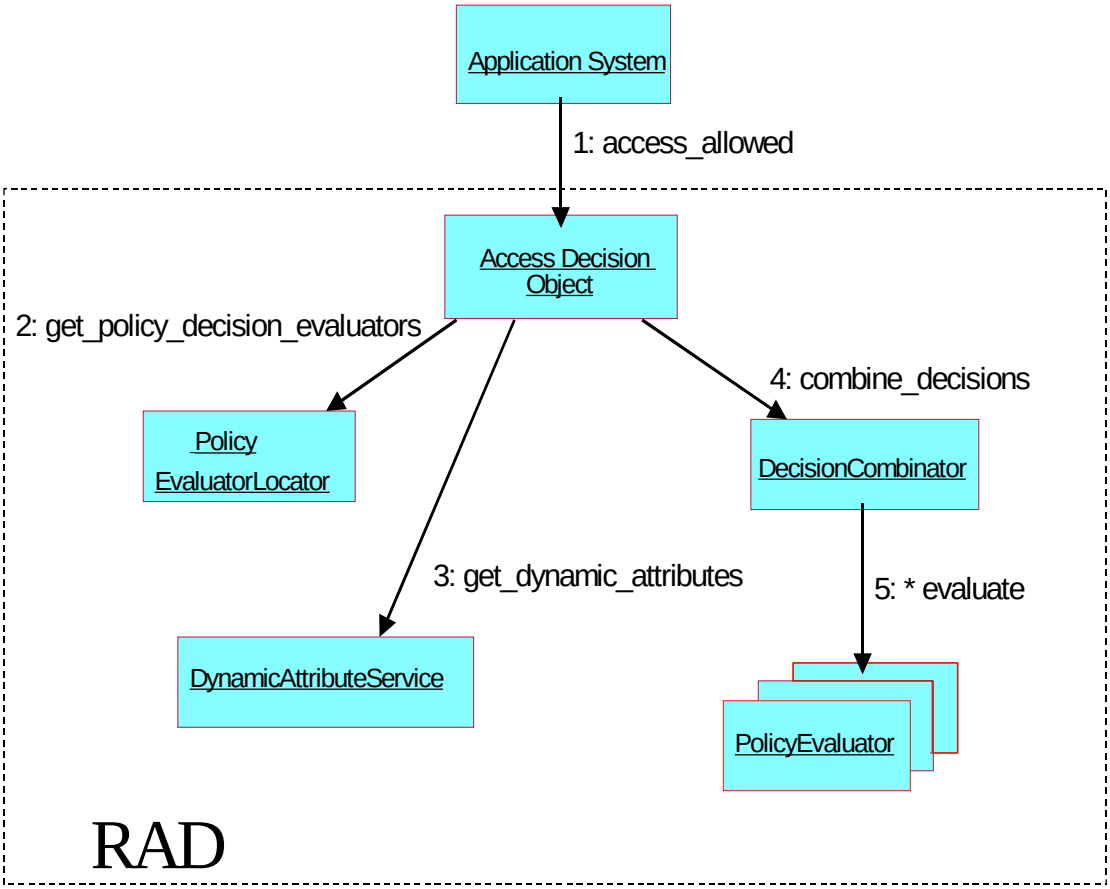


Figure 5-2. Interactions among RAD Components

are collectively responsible for making the authorization decision. The `DynamicAttributeService` (DAS) collects and provides dynamic attributes about the client in the context of the intended access operation and resource name.

The components are only logically disjointed while in practice they can be co-located in the same process or host. This feature is provided to further the support for dynamic composition and re-configuration, as well as for high availability and fault tolerance of the services based on RAD architecture.

Figure 5-2 shows interactions among components of authorization service. They are the following:

1. The authorization service receives a request via the ADO interface.

2. The ADO obtains object references to those PEs and DC which are associated with the resource name in question.

3. The ADO obtains dynamic attributes of the subject (client) in the context of the resource name and the intended access operation to be performed.

4. The ADO delegates an instance of DC for polling the PEs (selected in Step 1) and combining multiple results of evaluations made by PEs into a final decision. This is because there can be several PEs responsible for making authorization decision.

5. The DC obtains decisions from PEs and combines them according to the combination policy. The decision is forwarded to the ADO, which in turn returns the decision to the application.

To clarify the work of RAD components, we provide a short example of processing an authorization request in Figure 5-3. It shows the sequence of invocations among RAD com-

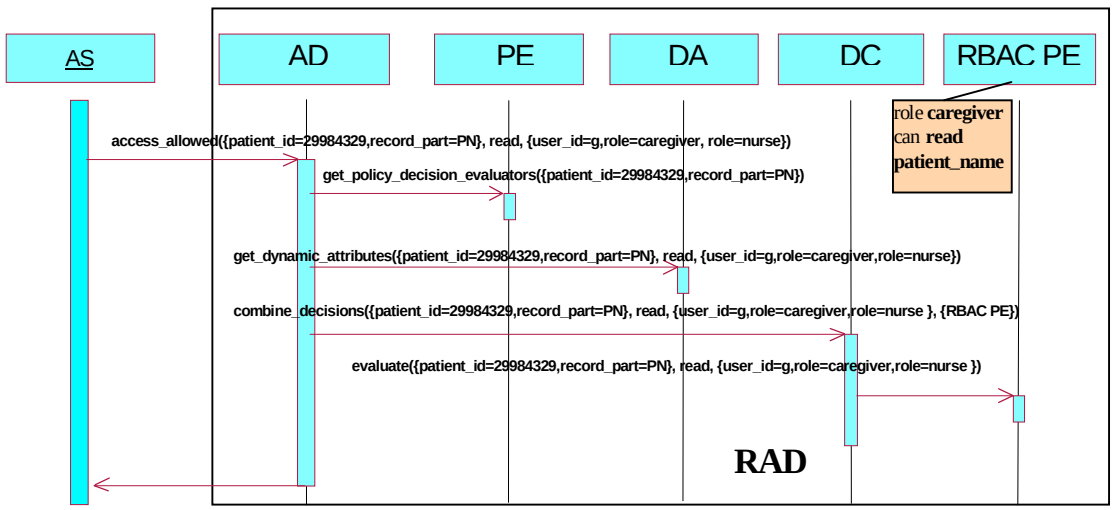


Figure 5-3. Interaction Diagram for Hypothetical Case

ponents in a hypothetical case. For the sake of illustration, let us assume that there is an authorization policy containing a statement that a user can read the patient's name (PN) if the user is performing role *caregiver*. In this example, AS is requesting authorization to perform access operation *read* on resource *patient name*. The resource is part of the medical record on the patient with ID *29984329*. Access is to be performed for a user with user_id *d*, who activated role *nurse* which is senior to *caregiver*. The ADO obtains a list of references to PEs and DC, which should be used for making an authorization decision on a resource with name {patient_id=*29984329*, record_part=*PN*}. The PEL returns a reference to the DC and a reference to one PE – *RBAC PE*. The DAS does not change the list of security attributes, which specifies that the user ID is *d* and the roles the user activated are *caregiver* and *nurse*. *RBAC PE* implements authorization based on roles. According to the authorization rules, users acting as *caregiver* have access to the names of all patients. Thus the PE returns "yes" and the DC returns the same answer to the ADO, which authorizes the AS to access the name of the patient with ID *29984329* on behalf of user *d*.

RAD architecture is such that all its components could be replaced dynamically by different implementations as long as they comply with the interface specifications. This enables the support for insertion and deletion of applications, changes in policies and the computing environment. For instance, if application insertion introduces new resources to be protected, a new PE (or even a set of PEs) can be dynamically added and PEL is reconfigured to use them. We will illustrate the support for changes in authorization policies in Section 5.2, where we discuss sample authorization policies and show how RAD can support them and their changes.

Unlike most authorization services [Simon 1997, Varadharajan 1998, Woo 1993c] RAD architecture does not restrict its implementations in the type of supported authorization policies. This is why the scope of authorization policy representation is beyond the scope of RAD architecture, as shown in Figure 5-4. Each PE can be administered using a

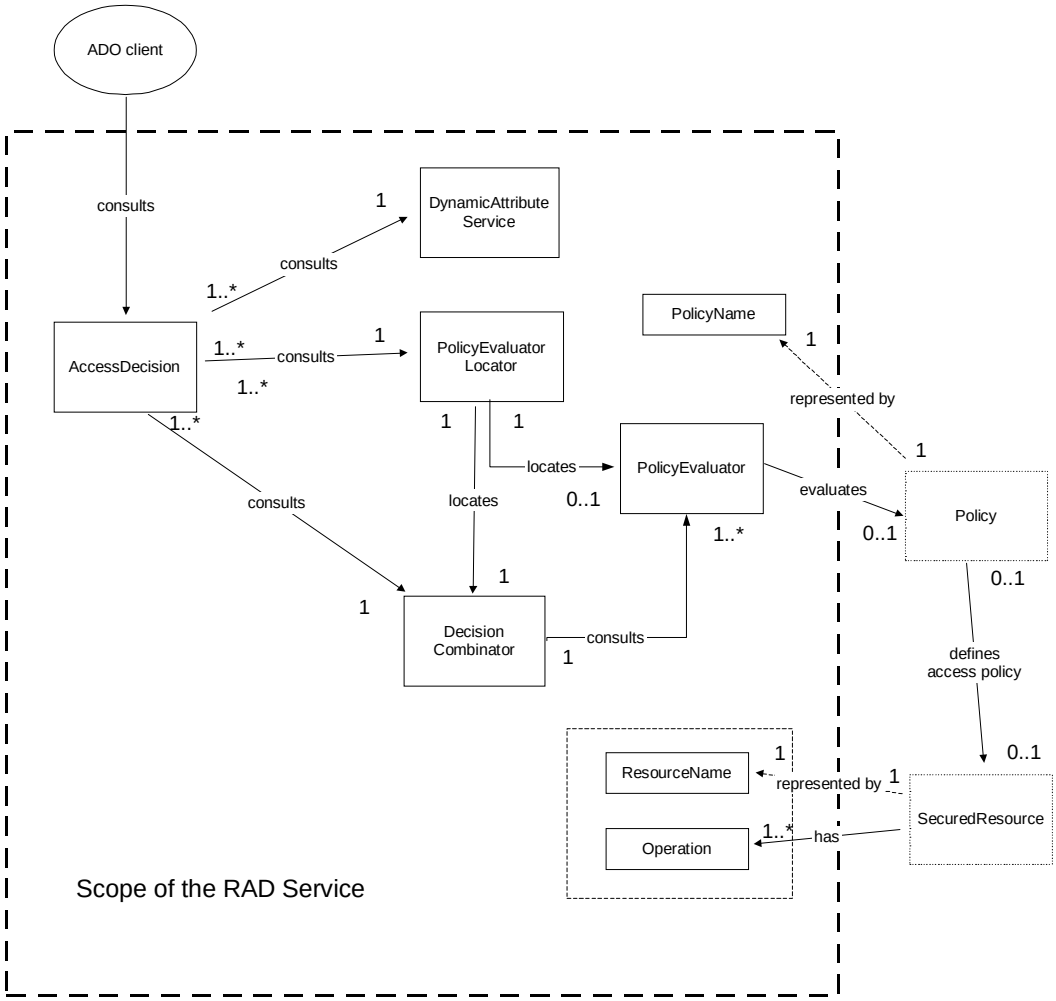


Figure 5-4. Main Run-time Elements and Their Appurtenance to the Architecture Scope (from [OMG 1999c])

different interface and AC rules written in a different language. Such a design enables the

use of the existing policy engines, which were not originally developed to be PEs (e.g. RACF [Benantar 1996]), and the support for future ones.

One authorization engine (supporting a particular policy) per request is used in Argos [Jonscher 1995] to evaluate requested access. The introduction of multiple evaluators and a combinator in RAD provides ways for more than one policy (even of different types) to govern authorization decisions for the same request. This is similar to [Bertino 1996b], where Bertino et al. define an explicit authorization model with conflict resolution and overriding rules. In RAD architecture, such rules are implemented by a particular DC.

One of RAD's distinguishing architectural elements is the use of DAS. It enables the support of policies based on the factors whose value can change from request to request or is determined by the state of organizational work-flow. These factors are furnished by DAS in the form of dynamic attributes, syntactically equivalent to subject security attributes. ADO obtains them from DAS before it passes the request to the corresponding DC and PEs. Dynamic attributes are attributes whose value can be determined only at the time when a request for an authorization decision takes place. Thus they are specific to the request in question. Examples of such attributes are relationships between physicians and patients in a hospital [Barkley 1999]. The introduction of DAS in RAD architecture increases the variety of information available for making authorization decisions, and enables the use of the traditional access matrix [Lampson 1971] to support complex and dynamic AC policies. We will illustrate the benefit of DAS in Section 5.4.

All RAD components, in addition to the run-time interfaces described above, have interfaces to administer them. Those interfaces constitute the RAD administrative model, the scope and main elements of which are shown in Figure 5-5.

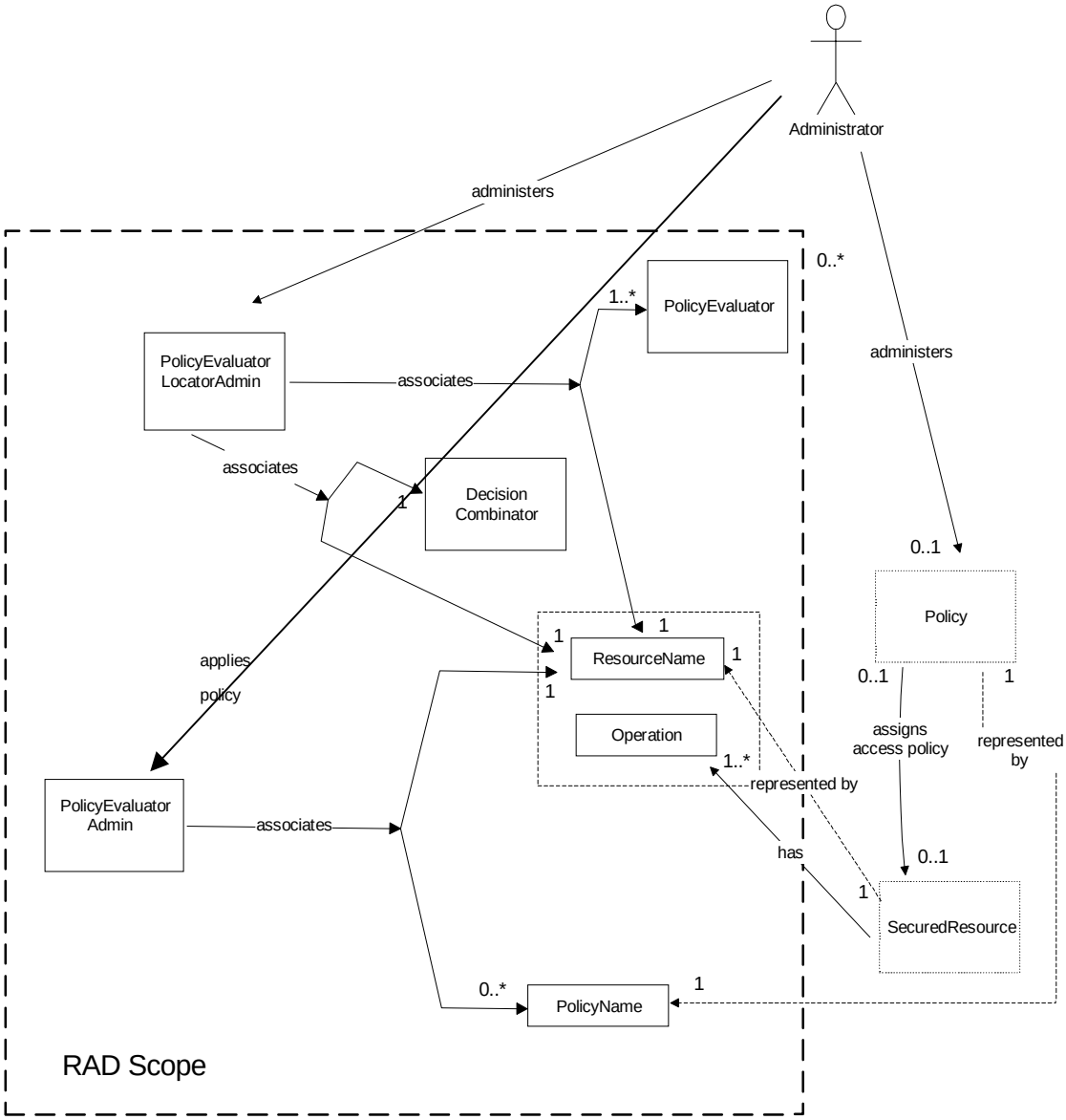


Figure 5-5. Administrative Elements and Their Appurtenance to the Architecture Scope (from [OMG 1999c])

Even though RAD architecture purposefully does not provide a means of specifying authorization policies and their representation, it allows RAD administrators to apply policies defined via implementation-specific PE interfaces to protected resources. This is carried through with the notion of policy name and with administrative interfaces for PE and PEL. A policy name is employed to associate the policy with a resource name for those PEs that can evaluate more than one policy. By naming a policy and avoiding a definition of policy representation, we keep RAD architecture open to the multitude of existing and future authorization languages.

Run-time and administrative interfaces and the supporting data structures, all defined in OMG IDL, along with prose description of their semantics, constitute RAD architecture. Its computational view is showed in Figure 5-6. The administrative part of RAD architec-

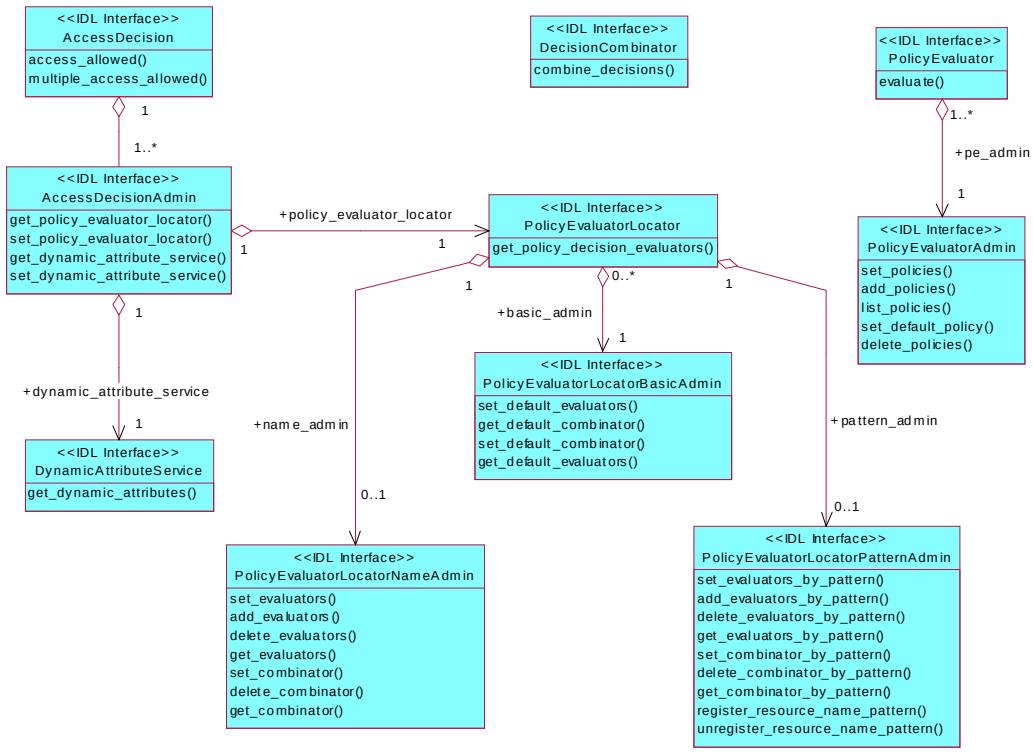


Figure 5-6. Computational Part of RAD Architecture

ture is designed to allow replaceable RAD objects within an implementation. For instance, `AccessDecisionAdmin` interface contains operations for inspecting and specifying the reference to PEL. Operation `set_policy_evaluator_locator()` allows a RAD administrator to "point" the ADO to a different instance of PEL. After the change, the ADO will use the new PEL. This is an example of how we address the goal of supporting changes in policies and the computing environment.

5.2 Example

RAD conceptual architecture is very generic, and the role of RAD components as well as the interactions among them could be hard to understand. This section provides a detailed example for illustrating RAD architecture and its capabilities. The example further clarifies RAD concepts. It also shows how policies based on roles and relationships can be supported by a RAD service.

We consider a set of simplified but typical access control policies in the health care domain which has arguably one of the most complex AC requirements. Consider a hospital computing enterprise consisting of many distributed systems, which are used for registration and billing, collecting results of laboratory tests and transcribed X-ray images, as well as for storing all other clinical information about patients including records of their visits to the hospital (for out-patients) and their stay over night, when they have complicated cases (for in-patients).

Hospital employees involved in the care process are called *caregivers* for short. A caregiver accesses many of those clinical, laboratory, transcription and financial systems

either directly with specialized client software or via general-purpose application programs. Such programs interact with several application servers in order to provide caregivers with information needed for patient diagnosis and treatment. Access to patient information (patient records) is controlled by AC mechanisms employed by the computing enterprise.

5.2.1 Initial Policies

Let us assume that the hospital adopts the policy listed in Table 5-1 to control employee access to the patients' medical records. Let us also assume that all patient records consist of the parts shown in Table 5-2.

Rule No.	Rule Definition
P1.1	Any **caregiver** can read patient's name.
P1.2	**Registration clerk** can modify patient name and demographic information.
P1.3	**Nurse** can read patient's name and demographic information, modify current episode demographic information, read current episode regular records and test results.
P1.4	**Technician** can modify current episode regular and sensitive test results.
P1.5	**Physician Assistant**, in addition to what a nurse can do, can also read all regular records of patients.
P1.6	**Physician**, in addition to what a physician assistant can do, can also modify current episode regular and sensitive records, as well as read regular, sensitive records and test results from previous episodes.
P1.7	**Psychiatrist**, in addition to what a physician can do, can also modify mental information.

Table 5-1. Access Control Policy (Policy 1)

This policy is coarse-grain in regards to the classes of users. The policy allows any nurse to read regular records of any patient in the hospital; technicians have full access to test results of all patients in the hospital; physicians have full access, except mental information, of the patients who have ever received care at the hospital. In addition, the policy does not reflect the fact that patients have relatives, guardians and other representatives,

Part name	Abbreviation
Patient name	PN
Demographic data	DD
Current episode demographic data	CDD
Current episode regular records	CRR
Current episode sensitive records	CSR
Current episode regular test results	CRT
Current episode sensitive test results	CST
Regular records from previous episodes	PRR
Sensitive records from previous episodes	PSR
Regular test results from previous episodes	PRT
Sensitive test results from previous episodes	PST
Mental information from all episodes	AMD

Table 5-2. Parts of Patient Medical Records

that are eligible to know some information about the status of their patient. Nonetheless, let us assume that the healthcare organization in our example has such privacy requirements that Policies 1 suffices. We will consider a new policy to deal with more complex AC decisions later in Section 5.4.

5.3 Modeling Policies

Policy 1 can be implemented using the RBAC model with role hierarchy -- $RBAC_1$ [Sandhu 1996]. In order to define the configuration of an $RBAC_1$ system, one needs to specify role hierarchy, user-to-role and permission-to-role relations, as well as functions *user* and *roles*. We define the role hierarchy (RH) in Figure 5-7. According to this hierarchy, for example, role *physician assistant* has as many permissions as role *nurse* plus its own permissions, because *physician assistant* is senior to *nurse*. User-to-role assignment relation (UA) is shown in Table 5-3, where we can see that user *g* is assigned to role *caregiver*, and user *d* is assigned to roles *nurse* and *technician*. This means that, when user *d* logs into the system, the user can activate either role *caregiver*, *nurse* or *technician*, whereas user *g* can only activate role *caregiver*. This because, according to the role hierar-

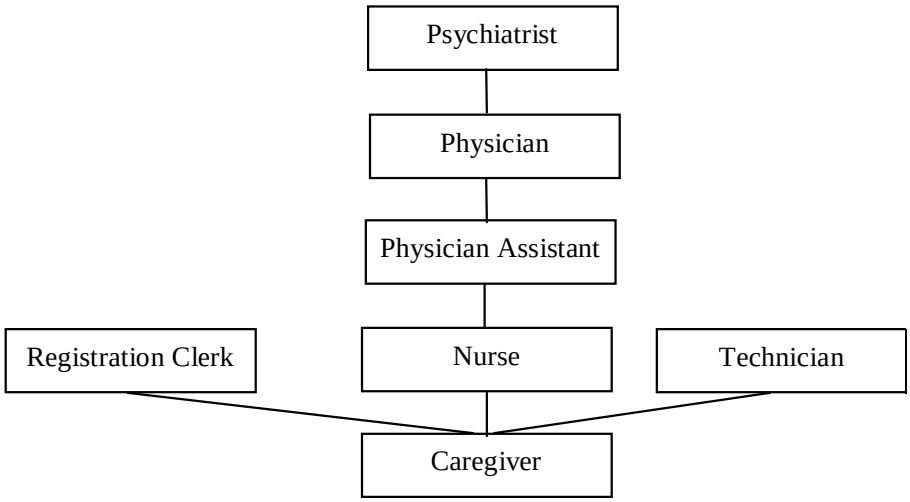


Figure 5-7. Role Hierarchy (RH relation)

Roles	Users						
	a	b	c	d	e	f	g
Psychiatrist	+						
Physician		+					
Physician Assistant			+				
Nurse				+			
Registration Clerk					+		
Technician				+		+	
Caregiver							+

Table 5-3. User to Role Assignment Relation (UA)

chy, a user can act in any role junior to the one he or she is assigned. If user *d* activates role *nurse*, then the subject will be granted all permissions assigned to roles *caregiver* and *nurse*. The permission-to-role assignment relation (PA) is presented in Table 5-4, according to which a *nurse* is assigned permissions to read demographic data (DD), current episode regular records (CRR), and current episode regular test results (CRT), as well as read and write current episode demographic data (CDD).

The configuration of a RAD server that performs authorization according to the $RBAC_1$ system defined by the above PA, UA and RH relations, is depicted in Figure 5-8.

Role	Resource											
	PN	DD	CDD	CRR	CSR	CRT	CST	PRR	PSR	PRT	PST	AMD
Psychiatrist												RW
Physician				W	RW		R		R		R	
Physician Assistant								R		R		
Nurse		R	RW	R		R						
Registration Clerk	W	RW										
Technician						RW	RW					
Caregiver	R											

Table 5-4. Permission-to-role Assignment Relation (PA)

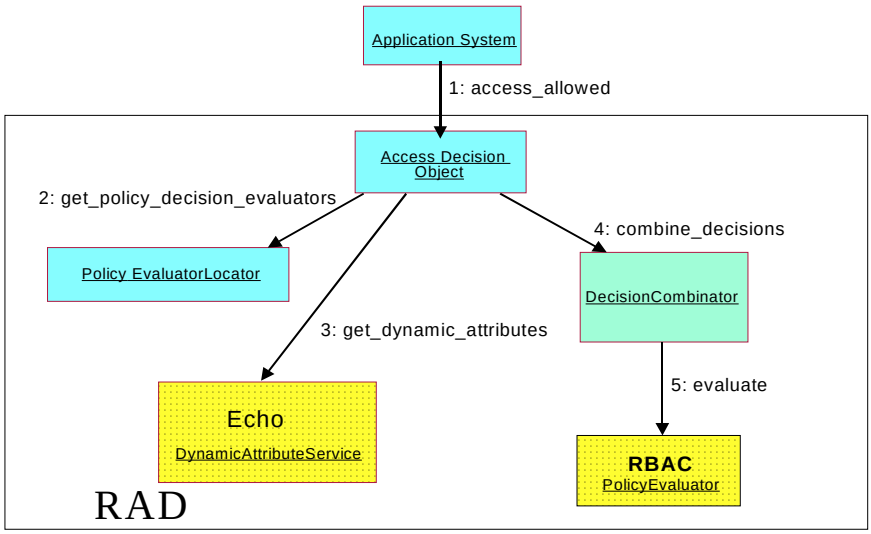


Figure 5-8. RAD Configuration for Role-based Policies

ADO obtains a reference for the DC and the only PE (RBAC PE) from the PEL, which always returns the same references. The DAS returns the same list of security attributes that it received from the ADO. RBAC PE evaluates authorization requests using PA relation. The DC denies access if the PE returns “unknown” as the result of evaluation (for example if the resource name is not found in the PA table), otherwise it returns whatever the PE returns.

We assume the availability of a distributed security environment which supports activation of roles by users during the authentication process with enforcement of UA and RH relations, and implementation of functions *user* and *roles*. This means that roles are implemented by the underlying security environment. There are security technologies capable of fulfilling this assumption. For example, we showed in Chapter 4 that CORBA security service [OMG 1996b] can support $RBAC_{0\text{-}3}$ models. This is why in our example, an application making authorization request to a RAD service supplies a list of principal security attributes which contains all roles activated by the user. The list, as described earlier, is obtained by the AS from the distributed security environment.

Another way of modeling the policy with RAD using $RBAC_1$ would be to assign the task of determining the user roles to the DAS or to RBAC PE itself. We preferred the first choice to the latter two because activated roles are security attributes managed by user administrators. They persist throughout the user session and should be activated during the authentication phase when the user logs into the system. This choice also supports dynamic separation of duties,[1] a commonly required RBAC feature.

One of our claims is that RAD architecture supports policy changes in a scalable way. Let us inspect how policy changes affect a RAD service. Changes to Policy 1 can either result in the replacement of the authorization model supporting the policy, or in changes to the system configuration which is defined via UA, RH, PA relations and functions *user* and *roles*. We will discuss in Section 5.4 how a RAD service can be reconfigured when an

1. Separation of duties is achieved by ensuring that mutually exclusive roles must be invoked to complete a sensitive task [Sandhu 1996]. Dynamic separation of duties is enforced in RBAC via constraints on role activation so that a user will not be able to activate mutually exclusive roles simultaneously even though each of them can be activated by the user.

authorization model is replaced. Now, we show what has to be done when only the system configuration is to be updated.

Only minimal alterations are required to accommodate RBAC re-configuration. Changes in UA, RH, *user* and *roles* do not affect RAD components because in our configuration they are entirely implemented by the underlying middleware security service. Changes to PA will result in different evaluations made by RBAC PE. For example, if rule P.1.5 in Table 5-1 was modified to allow physician assistants to read current episode sensitive records (CSR) of patients, then PA would be modified to have PA[*Physiscian Assistant, CSR*] = {R}. This would result in RBAC PE granting access for reading CSR to anyone whose list of activated roles contains "*Physician Assistant.*"

5.4 Advanced Policies

RAD architecture provides good support for changes not only in the policy content but also in its type. In this section, we show how a RAD service can be re-configured to support a more complex policy.

The policy listed in Table 5-1 (from now on called *role-based policy*) allow an employee to access records of all patients, regardless of whether the employee is involved in the provision of care to the patient or not. The principle of least privilege[1] is not fully supported. Let us assume that a new legislation requires the hospital to ensure that patient records are accessed not only according to the employee functions but also depending on whether the employee is actually involved in the patient care process. For example, only

1. The principle of least privilege requires that users should only be granted privilege for some activity if they have a justifiable need for its associated authorizations [Amoroso 1994].

the attending physician is now allowed to modify current episode records of the patient. Also, let us assume that now the patient's relatives, guardians and designated representatives have the right to limited access of the patient's records. To become compliant with the new regulations, the hospital replaces the old policy with the new one listed in Table 5-5.

Rule No.	Rule Definition
P2.1	Any **caregiver** can read patient's name.
P2.2	**Registration Clerk** can modify patient name and demographic information.
P2.3	**Nurse** can read patient's name and demographic information.
P2.4	**Attending Nurse**, in addition to the rights of any other nurse, can modify current episode demographic information, can read current episode regular records and test results.
P2.5	**Technician** can read patient's name and modify current episode regular test results.
P2.6	**Related Technician**, in addition to the rights of any other technician, can modify current episode sensitive test results.
P2.7	**Attending Physician Assistant**, in addition to what an attending nurse can do, can also read all (i.e. from the current and previous episodes) regular records and all regular test results, as well as modify current episode regular records.
P2.8	**Attending Physician**, in addition to the rights of an attending physician assistant, can modify current episode sensitive regular records and can read all regular and sensitive records from previous episodes.
P2.9	**Attending Psychiatrist**, in addition to what an attending physician can do, can also modify mental information.
P2.10	**Patient Relative** can read patient's current episode demographic and patient's name.
P2.11	**Patient Guardian** can read previous episode regular data.
P2.12	**Patient Spouse** can read previous episode sensitive data.
P2.13	**Patient Representative** can read previous episode regular data provided that patient gives a consent.

Table 5-5. New Policy (Policy 2)

The policy requires that only caregivers who are related to the treatment process for a given patient can have access to the corresponding parts of the patient record according to their job description. The new policy follows the least privileged security principle more closely then the old one. However, authorization decisions for such a policy can be made

only if the relationship between the patient and the user is taken into account. It is very challenging to make authorization decisions if only the RBAC model is employed. This means that, without more expressive authorization mechanisms, additional control must be exercised via manual procedures in the medical records department, which would severely inhibit the automation of the hospital health care process. To avoid this situation, the relationship between the user and the patient should be computed each time an authorization decision is to be made.

When AC logic is tightly coupled with application logic, the main challenge is to modify authorization logic in all clinical applications of the hospital so that they reflect the changes in hospital policy. This is a tremendously difficult, time consuming, expensive and error-prone process! For example, in order to accommodate the new policy, our hypothetical hospital would have to make changes in all its application systems that access patient records. With RAD, however, such changes can be made by dynamically reconfiguring the authorization service without any changes to the applications.

In order to enforce the new policy, we configure RAD service with new DAS and DC, as well as two different PEs. One PE is RBAC PE (the same as before). The other PE uses relationships instead of roles while making authorization decisions. For the sake of brevity, we employ name *RelBAC* to signify the use of relationships in authorization decisions. Therefore, the other is RelBAC PE. The new configuration is shown in Figure 5-9. The state of the authorization system for the new policy is described by 1) a role hierarchy, which is the same as the one shown in Figure 5-7, 2) a new PA relation (Table 5-6), 3) a

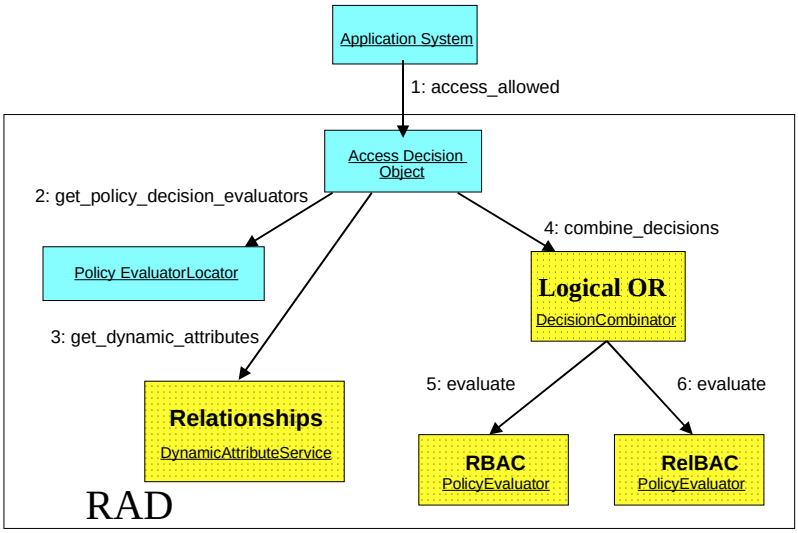


Figure 5-9. RAD Configuration for Relationship-based Policies

Roles	**Resources**											
	PN	**DD**	**CDD**	**CRR**	**CSR**	**CRT**	**CST**	**PRR**	**PSR**	**PRT**	**PST**	**AMD**
Psychiatrist												
Physician												
Physician Assistant												
Nurse		R										
Registration Clerk	W	RW										
Technician						RW						
Caregiver	R											

Table 5-6. Permission Assignment (PA) Relation for Role Hierarchy (New Policies)

relationship hierarchy (RSH) (Figure 5-10), and 4) a relationship to permission assignment (RSPA) (Table 5-7).

We outlined the support for such dynamic factors as relationships with RBAC mechanisms in [Barkley 1999]. Here we give a more concrete example of how the support of relationships can be implemented using the RBAC model and RAD service. Putting it simply, RelBAC is the same as $RBAC_1$ except that in RelBAC, role hierarchies should be viewed

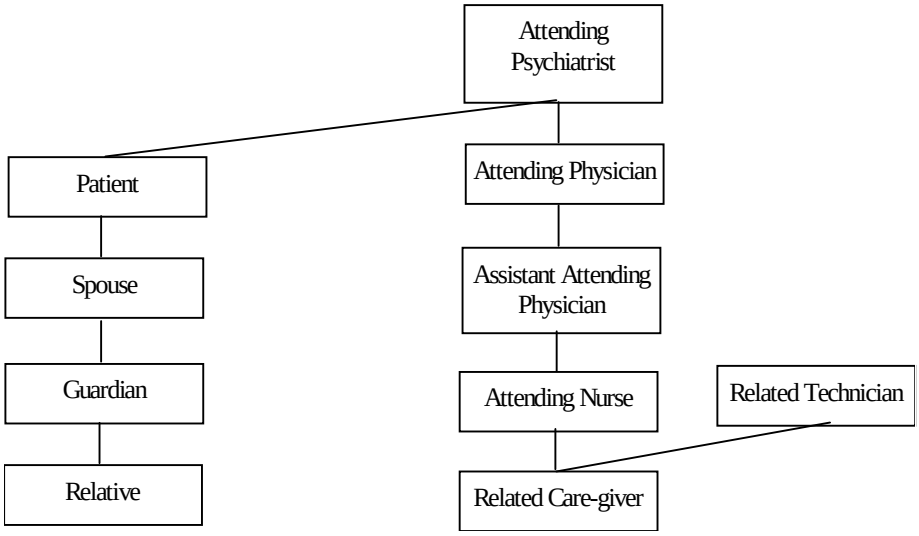


Figure 5-10. Relationship Hierarchy Relation (RSH)

Relationship	Resource											
	PN	**DD**	**CDD**	**CRR**	**CSR**	**CRT**	**CST**	**PRR**	**PSR**	**PRT**	**PST**	**AMD**
Attending Psychiatrist												RW
Attending Physician					RW		R		R		R	
Attending Physician Assistant				W		R		R		R		
Attending Nurse			RW	R		R						
Related Technician							RW					
Related Caregiver	R											
Patient												
Spouse									R		R	
Guardian								R		R		
Relative	R	R										

Table 5-7. Relationship to Permission Assignment Relation (RSPA)

in the context of a particular resource owner. In our example, relationship hierarchies are patient-centric, and they represent "roles" towards the patient. For instance, *attending physician* is a relationship that could be between a hospital physician and a patient. Role *phy-*

sician is an attribute of a user session, which persists through all actions undertaken during the session, whereas the value of the relationship between the user and the patient is always determined when a request to access patient data is authorized. A distinguishing feature of supporting RelBAC by RAD is that every time a request is to be authorized, the dynamic attribute service determines the relationship between the user and the patient whose records the user requested to access. The relationship information is added to the list of security attributes as new attributes of type *relationship* and values listing all the relationships junior to the one in question.

The two PEs work in concert coordinated by the DC. RBAC PE grants access only to those users who perform roles authorized to access patient data according to the PA relation showed in Table 5-6. For example, a user acting in role *physician assistant* is granted access to read demographic data (DD) for all patients in the hospital. On the other hand a user acting in role *physician* is denied access for reading patient current episode sensitive records (CSR) unless the user is determined to have *attending physician* relationship with the patient whose CSR records are to be accessed. The RelBAC PE grants such an access by basing its evaluation decision on RSPA (Table 5-7) and the value of *relationship* attributes inserted by DAS. The DC invokes RBAC and RelBAC PEs, and grants access if any of the two do so. Otherwise, it denies access, i.e. DC implements logical OR.

Let us walk through with a sample authorization request for the new policy. For illustration purposes, assume that a nurse with user_id *d* attends a patient with patient_id *29984329*. Consider an authorization request for operation *read* on current episode regular

records (CRR) associated with patient_id *29984329* on behalf a user with user_id *d*, who activated role *nurse*. The event sequence, illustrated in Figure 5-9, is the following:

1. The ADO receives the authorization request from the application.

2. The ADO obtains a list of references to PEs and DC, which should be used for making authorization decisions on resource with name {patient_id=*29984329*, record_part=*CRR*}. The PEL returns a reference to the DC and two PEs – *RBAC PE* and *RelBAC PE*.

3. The DAS adds two new attributes of type *relationship* with values *attending nurse* and *related caregiver* to the list of existing attributes which already has user id *d* and roles *caregiver* and *nurse.*

4. The ADO delegates the DC to make the decision.

5. *RBAC PE* denies access because, according to its PA relation (Table 5-6), neither role *nurse* nor *caregiver* has permission to read CRR data. The decision reflects the new authorization rules (P2.1 and P2.3 in Table 5-5) that do not allow reading CRR by any-one unless that person acts as physician assistant and attends the patient.

6. The DC requests *RelBAC PE* to evaluate the request. The PE uses its RSPA relation (Table 5-7) to determine that the access should be granted because RSPA[*attending nurse, CRR*] contains permission *R*. Thus the PE grants access.

Finally, the DC (implementing logical union) returns to ADO the same answer, and the ADO authorizes the application to access current episode regular records of patient with ID *29984329* on behalf of user *d.*

Now the RBAC and the RelBAC PEs work together to enforce the new authorization policy. However, it is possible to assign each rule from the policy to a specific PE based on its distinguishing function. By checking the Policy 2 (Table 5-5), we can find that rules P2.1, P2.2, P2.3 and P2.5 are suitable to be evaluated by the RBAC PE, while the RelBAC PE evaluates all other rules.

5.5 Discussion and Conclusions

In this chapter we presented an approach to separating authorization and application logic for those distributed applications which resort to application-level access control. The decoupling is a means to achieve the established earlier objectives of controlling access to the resources of enterprise distributed applications.

Our approach is formulated as an authorization service architecture -- RAD. The architecture is simple, generic and yet capable of supporting authorization decisions for wide variety of application domains. The main property, separation of authorization and application logic, is maintained when RAD approach is used because application delegates authorization decisions RAD-based authorization service. The architecture can support any level of protected resource granularity because of the generic data structure representing a resource name, which is used by applications for referring to the resources in question. The architecture is policy-neutral as opposed to other authorization service architectures [Varadharajan 1998, Zurko 1998], which allows implementation of various types of policies. For example, we demonstrated how role-based policies can be supported by RAD.

The architecture is also neutral to the nature of information used for making authorization decisions, as long as the information can be syntactically represented in the form of principal security attributes. This feature allows RAD-based services to support wide variety of authorization information. Moreover, the introduction of dynamic attribute service (DAS) defines a standard way to utilize request-specific information. We showed how RBAC policy engine can be combined with DAS that supplies user-patient relationships, in order to support policies based on caregiver-patient relationships in health care organizations. Because authorization requests to RAD-based services are invoked from within applications, the applications can provide the service with information available only while the application processes the client request, which is not supported, for example, by [Woo 1998]. Because the architecture enables encapsulation of authorization logic into a server, which can serve more than one application, the consistency of policies enforced across multiple applications is inherently supported.

New applications can be added and removed from the enterprise computing environment without affecting such a server. Changes to authorization policies, as we showed, cause re-configuration of RAD components or their composition and possibly replacement of some of them, which theoretically can be done dynamically without shutting down the server. The architecture enables administration scalability because changes to authorization policies can be done in one location. We will show in Chapter 5 using a CORBA-based prototype that RAD architecture enables component replacement with minimum affect on the work of the server. The above substantiates our earlier claim that RAD approach is adaptable to frequent changes in policies, applications, computing environment, and users.

The design of Adage [Zurko 1998] follows a pattern similar to that of our work. Their Authorization Decision Server (ADS) is encapsulated into a separate entity in the distributed environment with administrative and authorization interfaces. They are exposed to the management clients and the application servers via CORBA interfaces. In each authorization request to ADS, an application specifies the name of the accessing subject, the name of the resource (target in Adage terminology), and the action to be performed on the resource.

There are also many differences in the design. The foremost difference is in the partitioning of the authorization service into internal components. In Adage, an RBAC authorization engine, two rule databases and a translator are predefined and built into the ADS. Also Adage's authorization language syntax and semantics are fixed and predefined in the language interpreter. RAD architecture, on the other hand, allows different evaluation engines with their own rule languages and administrative interfaces to co-exist as long as few simple obligations for integrating those engines are fulfilled. This is achieved by defining not only interfaces for RAD clients and administrators but also interfaces for policy evaluators, decision combinators and other RAD internal components. The definition of RAD internal interfaces allows dynamic installation of third party RAD-compliant components in a RAD server. Furthermore, Adage authorization server can be used as one of RAD policy evaluators.

RAD re-uses CORBA Security service infrastructure. It relies on the service to provide all other security functionalities such as user security administration (group membership, role assignment, etc.), authentication, communication integrity and confidentiality, audit

and non-repudiation. The authorization engine and ADS administrative tools in Adage, however, are meant to be tightly integrated with user administration and authentication parts of the security infrastructure in order to evaluate activation rules used when a user is entering or leaving a role. This is needed to maintain static and current cardinalities of each role and the current labels of each subject if the enforced policies require static and dynamic separation of duties [Gligor 1986]. Moreover, the engine is designed to perform partly the user administration work (to enforce static separation of duty) and authentication work (dynamic separation of duty). Another difference is the existence of two logically distinct databases in Adage ADS. One is used to store Adage policy objects defined through the AL interpreter. Another stores a compiled form of the AL definitions that is optimized for evaluation by the authorization engine.

The body of work described in this chapter has been served as a foundation for Resource Access Decision Facility specification [OMG 1999c] from the Object Management Group which shows its practical usability. However, no matter how promising this approach is, it is important to establish its functional and performance feasibility. This is why we have developed a prototypical authorization service according to RAD architecture. We describe the service and the results of our studies in the next chapters.

6 CAAS -- Prototypical Implementation of RAD

In the previous chapter, we proposed a solution to the problem of controlling access to the resources of distributed enterprise applications -- an architecture for an application authorization service, RAD. We also showed that the architecture features key benefits: it enables the separation of application and authorization logic; it supports AC on fine-grain resources; it can be configured to implement different AC models, particularly RBAC; it supports the use of factors specific to the application domain or to the organizational workflow, such as relationships between the user and the resource owner; it enables the use of authorization engines created by different developers and administered by disparate authorities; and its distributed nature enables the consistency of authorization decisions across enterprise applications.

However, it is an open issue as to how one can design and implement a flexible (i.e. responsive to the changes in policies and conditions), extendable (i.e. capable of accommodating new functionality), and portable authorization server based on the conceptual architecture of RAD and what performance implications arise from employing such an approach. Answering these questions is critical in order to understand the validity of our and any other approach in this problem area. To the best of our knowledge, no research on authorization mechanisms for application systems reported in the literature, which we sur-

veyed in Chapter 3, examined the aspects of designing, constructing, and addressing performance in such mechanisms.

In order to study these issues, we designed and implemented an experimental test-bed -- CORBA-based Application Authorization Service (CAAS). It adheres to RAD architecture, and serves as a framework for our research on the RAD approach. Besides developing CAAS to serve as a test-bed, we also wanted to gain an understanding of the principles for constructing application authorization services.

This chapter is devoted to the design and implementation of CAAS. The main design requirements were flexibility, extensibility, portability and configurability. We actively utilized design patterns which provided us with simple and elegant solutions to general problems of constructing object-oriented component-based distributed security services. The service is based on CORBA and Java technologies, and utilizes CORBA Naming service.

We showed by the means of implementation that RAD architecture is feasible and its computational model, defined in IDL, is correct. Besides the feasibility proof, we gained more understanding of the design and implementation of an authorization service for distributed applications.

The chapter is organized as follows. The next section gives an extensive overview of CAAS design and explains the main elements of its components. We illustrate the points of the section by describing in detail designs of DC and PE in Sections 6.2 and 6.3. We discuss the results of designing and developing CAAS and conclude the chapter in Section 6.4.

6.1 Overview of CAAS Design

As mentioned earlier, the main goals for CAAS construction were proving RAD approach feasibility, and developing an experimental framework for further research on the support of application-specific fine-grain, complex and dynamic access control policies, while providing a necessary degree of usability, fault tolerance, scalability and availability. This is why, besides making CAAS design confirm to RAD architecture, we strived to achieve its configurability, implementation affordability, portability, as well as flexibility and extensibility sufficient for the current and future research. In this section we give an overview of CAAS main design elements that allowed us to achieve the objectives.

6.1.1 Middleware Technology

To make CAAS implementation portable and extendable, we used standard technologies as much as possible. CORBA became the middleware technology of choice. Its security service provided the functionality necessary to model different authorization policies. CORBA Naming service allowed CAAS distributed components to discover each other in a platform-independent way. We were free to choose any implementation language for each CAAS component. The choice of CORBA influenced the overall composition of the service's main elements, shown in Figure 6-1. All of them interact via Interoperable Inter-

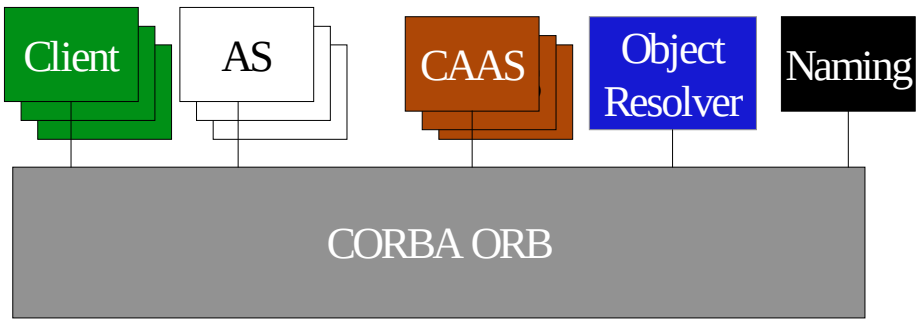


Figure 6-1. CAAS Main Elements

RAD Component	IDL Interface Defined by RAD Architecture	Extended IDL Interface Defined by CAAS Design
ADO	AccessDecision	AccessDecisionExt
	AccessDecisionAdmin	AccessDecisionAdminExt
PEL	PolicyEvaluatorLocator	
	PolicyEvaluatorLocatorAdmin	
	PolicyEvaluatorLocatorBasicAdmin	PolicyEvaluatorLocatorAdminExt
	PolicyEvaluatorLocatorNameAdmin[a]	
	PolicyEvaluatorLocatorPatternAdmin[a]	
DAS	DynamicAttributeService	DynamicAttributeServiceExt
		DynamicAttributeServiceAdminExt
DC	DecisionCombinator	
PE	PolicyEvaluator	PolicyEvaluatorExt
	PolicyEvaluatorAdmin	PolicyEvaluatorAdminExt

Table 6-1. Correspondence Between IDL Interfaces Extended by CAAS Design and RAD

a. Not implemented in the current version of CAAS

ORB Protocol (IIOP) [OMG 1999a], which is a standard communication protocol for CORBA-based systems communicating over TCP/IP. The next major design decision was about the interfaces CAAS components should provide.

6.1.2 Component Interfaces

IDL interfaces defined in RAD architecture expose functionality common to all services based on the architecture. CAAS design is required to provide additional functionality exposed via interfaces. The functionality should allow run-time interfaces to obtain references to administrative interfaces and enable graceful shutdown of the components. Therefore, we introduced extensions to RAD run-time and administrative interfaces listed in Table 6-1. These extensions allow the implementation of additional functions without altering the RAD interfaces. Due to CORBA IDL interface inheritance capability, newly defined interfaces were seen by CAAS clients as base RAD interfaces unless additionally defined operations and attributes were used.

6.1.3 Implementation Language

The next design decision was about the implementation language. It was influenced by two requirements -- the implementation portability and the ease of programming for graduate students, mostly unprofessional developers. To address them, we used Java as the implementation language. Implementations of Java Virtual Machine (JVM) are available for most operating systems, and the language provides several advantages for rapid development such as object-orientation, thread and garbage collection support. Java also provides dynamic loading of classes, and this allows great flexibility in configuring and changing CAAS behavior at boot- and run-time, and loading of Java classes compatible with underlying ORB middleware.[1]

However, Java imposed several constraints. Most CAAS components provide multiple IDL interfaces -- run-time and administrative. Run-time interfaces are used during the computation of authorization decisions. Administrative interfaces define operations through which the behavior of CAAS components can be configured. Given that, we decided for each CAAS component to implement both types of IDL interfaces using a single Java class, as shown in CAAS architecture in Figure 6-2. For example, Java class `DynamicAttributeService` implements both IDL interfaces `DynamicAttributeServiceExt` and `DynamicAttributeServiceAdminExt`. In Java, an IDL interface is implemented using a class which defines public methods corresponding to the operations and attributes of the IDL interface [OMG 1999b]. However, we could not use inheritance

1. For the time of developing CAAS only few ORB vendors had Portable Object Adapter (POA), which precluded us from using POA in order to achieve complete code portability on the server side.

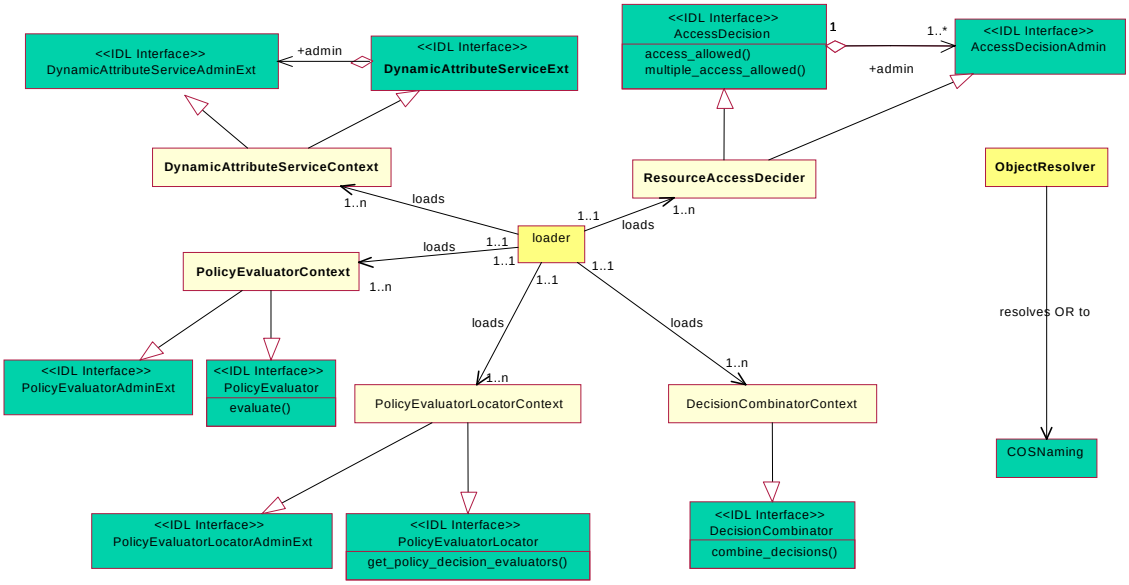


Figure 6-2. CAAS Architecture

for implementing run-time and administrative IDL interfaces because Java does not support multiple class inheritance.

To work around the single-inheritance restriction of Java, we implemented components using a delegation mechanism known as the *Tie* approach [Pedrick 1998]. In this approach, a single *tie* class implements a number of CORBA interfaces. However, the *tie* only implements the minimum mechanisms needed to interact with the ORB environment. The actual implementation of the component's operations is done in a *delegate* class implementing the `ComponentOperation` interface, as shown in Figure 6-3. With this approach, we obtained greater flexibility in composing objects since the delegate class is not restricted to inherit from any particular class. The only requirement is that the delegate class implements the `ComponentOperation` interface.[1]

1. One drawback of delegation is that systems relying on object composition may be more difficult to comprehend [Gamma 1995].

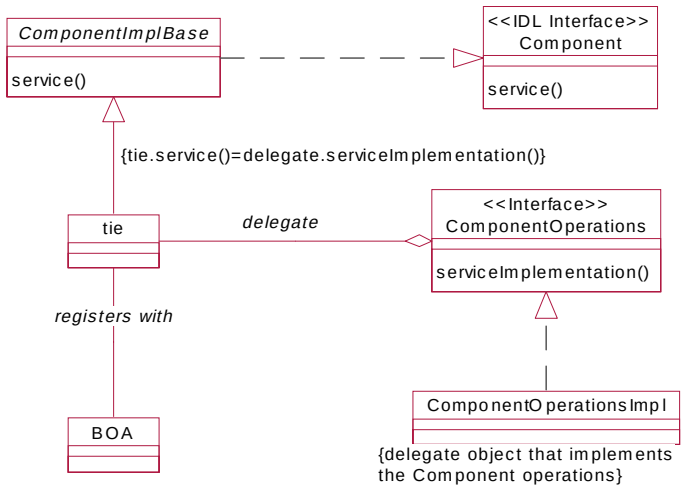


Figure 6-3. Implementing a CORBA Object Using the Tie Approach

Current versions of Java ORBs support concurrent invocations by executing the instances of a CORBA object in more than one thread. Although a performance benefit, this feature requires carefulness in changing an object state. To address this issue, we decided in the current version to use fully synchronized methods for the implementation of CAAS. Although this property does not guarantee that the system will be free of liveliness failures such as deadlocks and resource starvation, it does guarantee consistency of values at the object level. This design solution allows synchronized method implementations to be used in concurrent settings [Lea 1996]. However, this introduced unnecessary synchronization which can affect overall run time performance because calls to synchronized methods are more expensive, than to un-synchronized ones. Also, synchronized operations on CAAS components are of a coarse granularity which can cause threads to block and unblock unnecessarily.

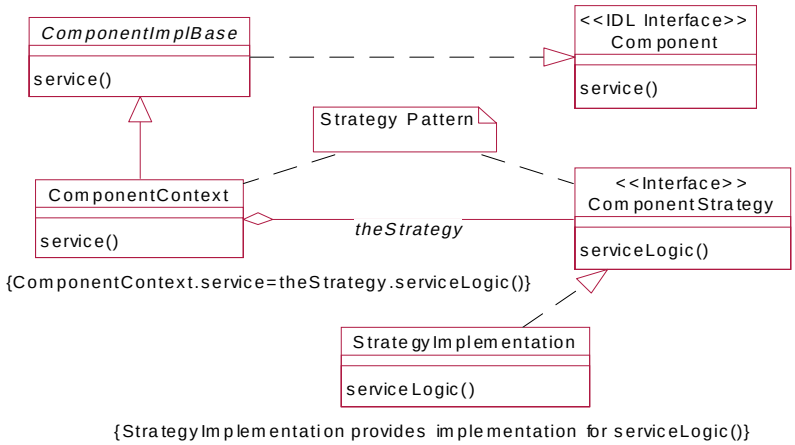


Figure 6-4. Implementing a server using Strategy pattern

6.1.4 Design Extensibility

During the design process, it became evident that different instances of the same CAAS component, such as DC and PE, must implement different logic. For instance, a DC can combine results from multiple PEs in more than one way. One solution would be to implement one class per component behavior. However, this would create many related classes that differ only slightly in their functionality. The solution we chose was based on the design pattern *Strategy* [Gamma 1995].

In *Strategy* pattern, a *Context* class implements the logic common to all other implementations, and a *Strategy* class provides behavior specific to the concrete implementation, as illustrated in Figure 6-4. The pattern allowed us to implement families of algorithms related to each CAAS component (strategy classes) and common functionality (context classes).

Since Java was our implementation language, we defined strategies as Java interfaces. In this case, component contexts are Java classes implementing the services published by

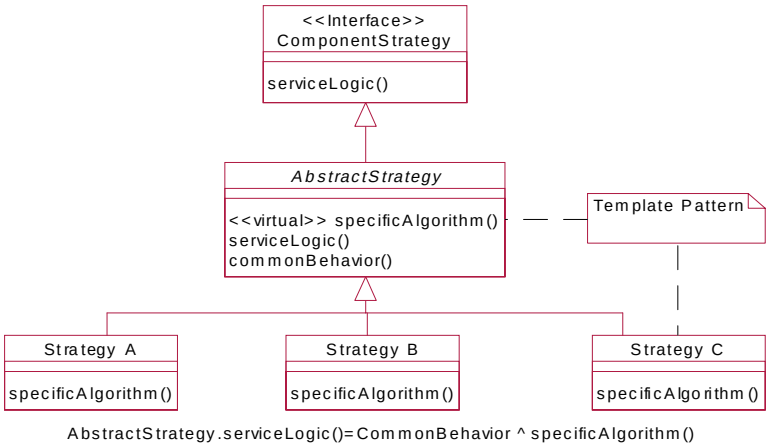


Figure 6-5. Applying Template Method Pattern

the strategy interfaces. With the implementation of the strategies for the DC and PE components, we took a step further: their implementation is based on a design pattern known as the *Template Method* [Gamma 1995]. The idea (illustrated in Figure 6-5) behind the pattern is to define an outline or skeleton of an algorithm in a base class while leaving some steps to be defined in subclasses.

Template Method pattern was used in the design of DC and PE because implementations of each of these components tend to share a common functionality. For example, implementations of DC need to resolve references to PE objects received from the ADO regardless of the decision combination policy being implemented. Similarly, PE implementations need to maintain associations of policies to resource names independently of how the policies are stored and evaluated. Such a common functionality can be implemented in an abstract strategy class (Figure 6-5). This class is later refined to obtain specific implementations (strategies A, B, C in the example in Figure 6-5).

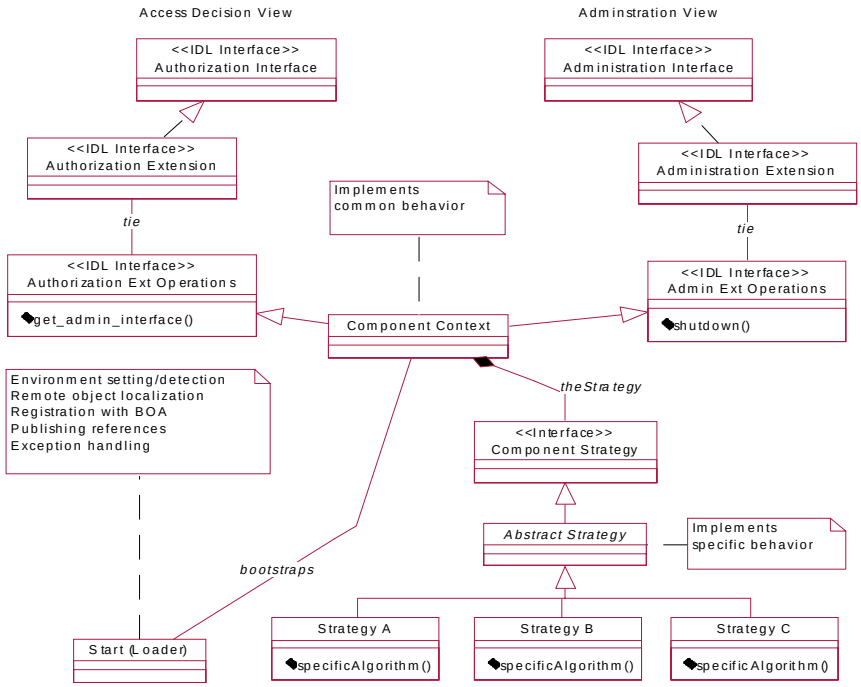


Figure 6-6. Structure Common to Most CAAS Components

6.1.5 General Component Structure

We structured all components in the same fashion, as shown in Figure 6-6. This made the design and coding faster, because the former could be re-used and the developers had to learn only one structure in order to understand the principles of work for each component. It was also easier to see the differences. For example, PEL does not have an extension to its administrative interface, whereas DC lacks an administrative interface due to its simplicity.

6.1.6 Component Initialization and Discovery

We wanted to study CAAS performance under different configurations and loads. Do so would require CAAS to provide a number of capabilities: to use different policy evaluators and/or decision combinators; to allow the deployment of different components in dif-

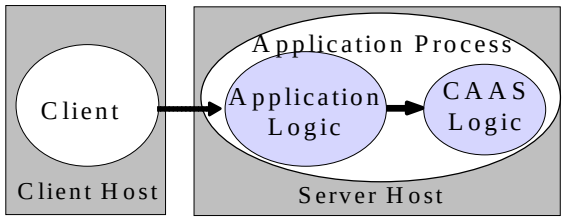


Figure 6-7. Reference Configuration

ferent locations on the network and in the system by co-locating components in the same process or host; and to change its configuration with relative ease and repeat the same experiments over time. For example, it should be possible to combine application and authorization logic in one process, as shown in Figure 6-7, or to load each CAAS component in a separate process (Figure 6-8). Moreover, we wanted to have the capability of loading the service in different configurations without recompiling the source code.

In order to ease the process of booting CAAS components in different configurations, we introduced two techniques. First is the use of a component loader, shown in Figure 6-2, which enables any number of instances of the same component to be loaded in one process. All the information needed by the loader was provided via either configuration file or the command line parameters. However, once the components are loaded, it is necessary for them to discover each other, i.e. obtain corresponding object references, no matter if they are located in one process, on one machine or on different network nodes. It was also desired to avoid the use of the middleware when process co-located components communicate with each other, in order to avoid unnecessary overhead. This is why the second technique -- uniform URL-like representation of component references -- was introduced. The technique allowed us to choose the ways of posting and obtaining object references of the components by simply changing the content of the symbolic readable reference represen-

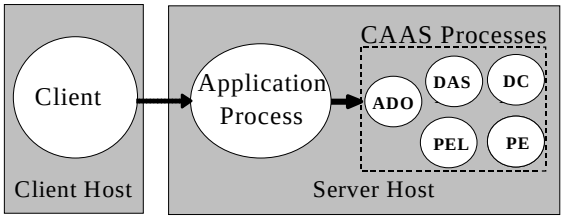


Figure 6-8. CAAS Configuration with Each Component in a Separate Process

tations, specified in the configuration file or command line, and to avoid superfluous use of middleware when both the client and the target are located in the same process.

For discovering components located in different processes, either interoperable object reference (IOR) stored in the stringified form in a text file, or CORBA Naming service can be used. The latter is most convenient when the components are located on different machines. Since CORBA Interoperable Naming Service [OMG 1998a] implementations were not available at the time of the development, we utilized object locator approach (shown in Figures 6-1 and 6-2) similar to the one in TAO [Schmidt 1998] and discussed in [Schmidt 1999]. The main benefit of the locator is the complete portability in locating naming service. It is done by sending a UDP broadcast to a predefined port. If the locator instance is available on the network, it will respond with a stringified IOR for the naming service root context, which is sufficient for finding a component IOR by its name in the naming hierarchy. The design is sufficiently generic to discover IORs of other CORBA objects.

To illustrate CAAS design elements discussed above, we discuss DC and PE in the next two sections. Although other components are equally important, their design is similar to DC and PE. A more detailed description of ADO, PEL and DAS can be found in [Espinal 2000].

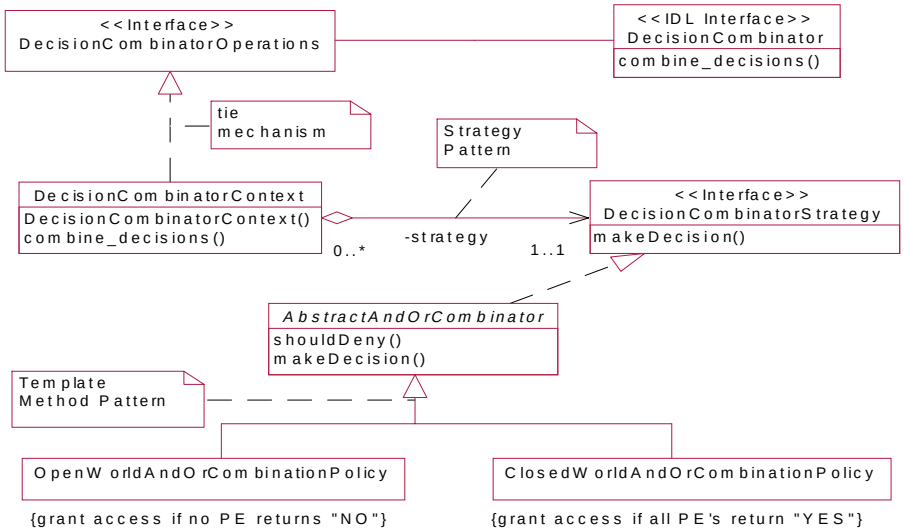


Figure 6-9. DecisionCombinator Design

6.2 Decision Combinator

DC encapsulates the “decision combination” logic which is delegated to an object implementing `DecisionCombinatorStrategy` interface (Figure 6-9). DC only has a run-time interface in the current version of CAAS, the `DecisionCombinator` with `DecisionCombinatorContext` as the class implementing the IDL interface. Nonetheless, the design of `DecisionCombinatorContext` uses the Tie approach to accommodate the introduction of future administrative interfaces.

DC features the simplest design of all CAAS components. However, DC objects can exhibit different behavior. For instance, a DC can combine results from multiple PEs in more than one way, e.g. one type of DC can combine multiple results using a logical AND combination policy, whereas another type can combine multiple results using a majority vote policy. These two forms of policy, however, do not necessarily change the way a DC consults the PEs; that is, in both cases a DC may not need to consult all of them. Taking

these issues into account, we designed DC using the Strategy pattern. With this pattern, a `DecisionCombinatorContext` class implements functionality required to consult PEs handed by ADO. Different decision combination policies are then delegated to an object implementing the `DecisionCombinatorStrategy` Java interface. One of the interface implementations is class `AbstractAndOrCombinator`. The class is further refined (using the Template pattern) into two classes, `OpenWorldAndOrCombinationPolicy` and `CloseWorldAndOrCombinationPolicy`. With the former policy, a DC grants access if no PE object denies access, and with the latter it implements a stricter combination policy -- it grants access only if all PE objects do so.

Having described the simplest component -- DC -- we will discuss the design of PE, which is the most complex.

6.3 Policy Evaluator

The function of a PE is to evaluate one or more of the authorization policies in regards to a resource given a list of principal security attributes, the resource and operation names. The PE has run-time and administrative IDL interfaces -- `PolicyEvaluator` and `PolicyEvaluatorAdmin`. The two are extended with `PolicyEvaluatorExt` and `PolicyEvaluatorAdminExt` IDL interfaces (see Figure 6-10). For their implementation, we use a single Java class, `PolicyEvaluatorContext`.

As mentioned in Section 6.1, an IDL interface is implemented in Java with an *implementation* class [OMG 1999b], and thus inheritance cannot be used for implementing multiple IDL interfaces. Because of this constraint, we used the Tie approach for implementing

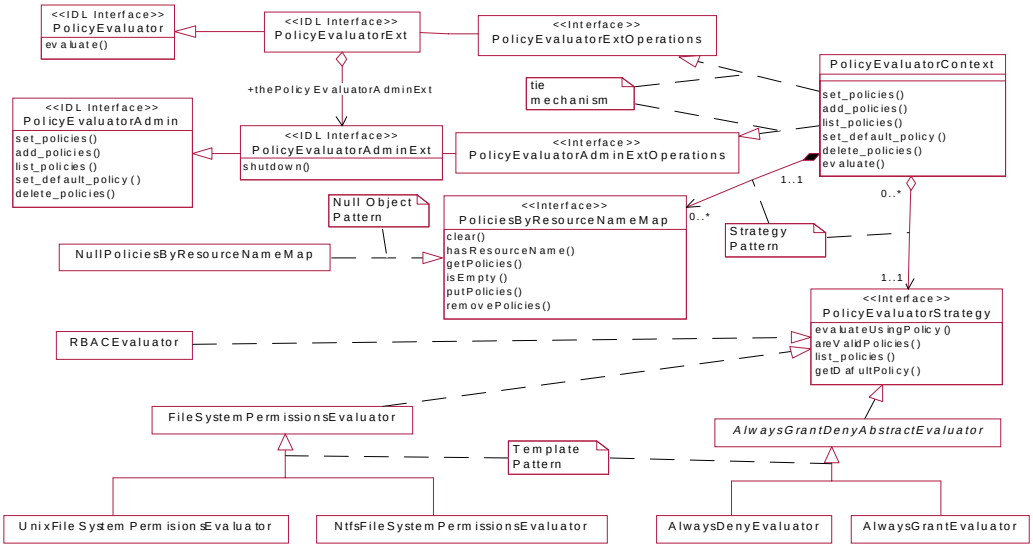


Figure 6-10. PolicyEvaluator Design

`PolicyEvaluatorExt` and `PolicyEvaluatorAdminExt` IDL interfaces. In the case of PE, `PolicyEvaluatorContext` delegates the functionality of its operations to objects that implement the `PolicyEvaluatorExtOperations` and `PolicyEvaluatorAdminExt` Java interfaces (see Figure 6-10).

Different instances of PE can exhibit different behavior. For instance, a CAAS service may utilize PE components implementing policy evaluation mechanisms based on filesystem permissions, RBAC, or even default evaluation policies which always grant or deny access. However, most of these instances of PE may use the same mechanisms to associate resource names to access control policies.

To avoid the introduction of many related PE classes that differ only in their evaluation policy, we use a solution based on the *Strategy* pattern. With this pattern, `PolicyEvaluatorContext` implements functionality common to most other implementations of PE. For example, addition and removal of authorization policies is not likely to change between

PE instances. Different evaluation policies are then delegated to an object implementing the `PolicyEvaluatorStrategy` Java interface (see Figure 6-10). Similarly, the management of resource name associations for accessing policies may vary between PE instances. Consequently, `PolicyEvaluatorContext` delegates the implementation of such functionality to objects implementing the `PoliciesByResourceNameMap`. By using this interface, the association can be implemented by employing any form of storage suitable to the current needs independently of `PolicyEvaluatorStrategy` implementation.

Implementations of `PolicyEvaluatorStrategy` interface are further refined using the Template Method pattern, as shown in Figure 6-10, which allows extensions and modifications to policy evaluation mechanisms with relative ease as the needs for different evaluation logic change during the system life cycle.

Another pattern we used in the design of CAAS components is the *Null Object* pattern [Grand 1998]. With this pattern, developers can provide “do-nothing” versions of classes for which no particular implementations exist during execution. In the case of PE design, it was used to define the `NullPoliciesByResourceNameMap` class as the default implementation of `PoliciesByResourceNameMap` interface (see Figure 6-10). The class relieves `PolicyEvaluatorContext` from testing for null values before accessing the interface methods.

6.4 Discussion and Conclusions

We claimed in the previous chapter that RAD architecture has the following properties: simplicity, flexibility and generality. Simplicity is achieved by using simple interfaces, by requiring AS to make simple operation invocations on RAD service, and by using simple structures for exchanging information between applications and a RAD service. Simplicity is also achieved by using encapsulation principles in RAD architecture and CAAS design. The programming complexity of making authorization decisions for an individual policy is encapsulated in PEL, DAS, and PE objects. While constructing CAAS, we found that DC greatly contributes to the simplification of CAAS design. This is because DC encapsulates decision combination policies which can completely change the overall authorization logic of CAAS. Total complexity increases only when complex access policies are added to CAAS, yet such complexity is still contained within the appropriate components. Increased complexity within PE implementation does not increase the complexity encapsulated by DAS or PEL and vice versa. However, it might be possible that in some cases, the introduction of more complex policy evaluators could increase the complexity of decision combinators.

Flexibility is another property present in CAAS. Changes in CAAS would manifest as changes in access control policies, policy evaluations and dynamic attributes; new access control polices, for example, can be implemented by changing or replacing existing PE and DC objects, as we showed it in the example of the previous chapter. In order to demonstrate the extensibility and flexibility of RAD, we designed CAAS to support run-time shut down, re-initialization, or replacement of its components. For example, we implemented different versions of DC, and showed that those versions can be replaced “on the fly.”

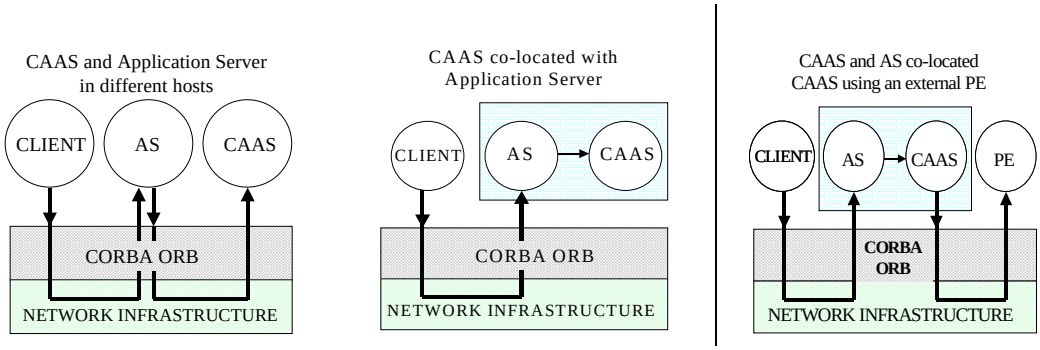


Figure 6-11. CAAS under different configurations

More evidence of RAD architecture flexibility is the support for different configurations of CAAS components. For conducting performance experiments using CAAS different configurations, we designed it to support the deployment of the components, without changing source code, co-located in a process, computer or distributed over a network (see Figure 6-11 for examples). This configurability allows such CAAS deployments that maximum performance (by avoiding ORB middleware and network overhead), availability, or flexibility (by having any component in any system in the network) are achieved.

After designing and implementing CAAS, we find RAD architecture sufficiently general in the sense that it can be implemented for different environments, with different requirement and design priorities. A straight-forward implementation, intended for environments with tolerant requirements, could be done with few lines of code without usage of design patterns. On the other hand, a RAD service can be implemented using a complex design to achieve fault-tolerance, high-performance and scalability. Our current implementation of CAAS tries to obtain a balance with a simple design which allows it to be flexible, extendable and configurable.

By utilizing standard technologies, namely CORBA and Java, we have developed a concrete implementation of RAD architecture -- CAAS. The implementation is flexible, configurable, extendable and portable. The design and implementation of components available in CAAS is covered with more detail in our technical report [Espinal 2000].

The main contribution of the work presented in this chapter is a concrete design of a prototype (CAAS) of RAD architecture. The design is sufficiently flexible to deploy CAAS under different configurations, and to experiment with different authorization policies of different granularity and complexity. We showed that RAD architecture is feasible and its computational model, defined in IDL, is correct. Besides the feasibility proof, we gained important insights into the design and implementation of an authorization service for distributed applications.

During our work on CAAS we actively utilized design patterns, which provided us with simple and elegant solutions to general problems of constructing object-oriented component-based distributed information systems. CAAS design and implementation is a required step towards a comprehensive study on support of application-specific fine-grain, complex and dynamic access control policies in heterogeneous distributed enterprise applications that are to constitute current and future information enterprises. The initial goal of using CAAS was to study the implications of RAD architecture on the system end-to-end performance. We report on the study and its results in the next chapter.

7 CAAS Performance Measurements

One of the main concerns about RAD-based authorization services is the overall system performance. Regardless of how attractive the approach is, if the resulting implementation impedes the application capability to comply with its performance constraints, the approach would not be of much help to the developers. In this chapter we report on our studies about CAAS performance.

The main question with the performance of authorization services based on RAD architecture is not whether a performance fee has to be paid but how much it is. One would expect middleware and communication overhead to affect the application response time the most. However, we need to qualify and quantify the overhead. Because RAD architecture defines multiple components that can be located in the same process, in the same host or in different hosts in a network environment their different compositions will affect overall run-time performance to various extent.

Another question is what features of RAD architecture or the design based on it inherently affect the performance of application systems. The third, equally important, question is what application domains can absorb the performance penalty, since not all the applications have the same strict constraints on their response time or the time is determined by other factors more then by the authorization delay. Knowing the performance penalty, can we identify the application groups where such a penalty is acceptable? So far, we have not

seen any reports in the literature about studies on either the performance trade-offs for authorization services or other questions stated above. In this chapter, we discuss how we addressed these questions.

We used CAAS as a test-bed. The focus of the experiments was the run-time performance of application systems that obtain authorization decisions from CAAS. We measured the performance under various configurations, loads and server-side application logic delays using a simple performance model.

The main contributions of the work are our performance measurements and the conclusions we have drawn from them. We identified factors affecting run-time performance of systems using CAAS and possible solutions for improving the performance of authorization services based on RAD approach. Moreover, we believe the performance results can be used to measure and reason about the performance of authorization servers in general. We also gained the understanding of how the amount of time spent on executing application logic affects the performance penalty experienced by an application. This helped us to qualify the applicability of CAAS and similar implementations to the different application domains.

The organization of this chapter is as follows. The next section discusses the performance model. We describe CAAS configurations used for the experiments in Section 7.2. The test environment and the experimental procedures are explained in sections 7.3 and 7.4 respectively. We report on the data and interpret it in Section 7.5. Based on the experimental data interpretation, we suggest the ways for achieving adequate performance for RAD-based services in Section 7.6. Conclusions are drawn Section 7.7.

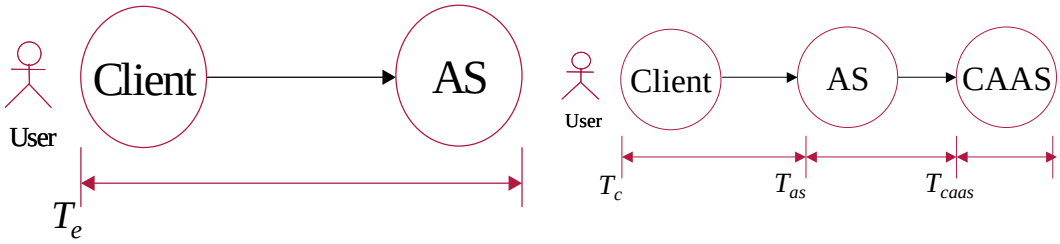
7.1 Measurement Model

It was crucial to define the model for our experiments, which would enable answers to the stated questions. A measurement model determines the scope of an experiment, what results could be obtained, and how they should be interpreted. It also determines the complexity and affordability of experiments. Since performance studies are not the central contribution of our research, we decided to follow a minimalistic approach, i.e. to use such a model that would allow us to obtain required performance measurements with the simplest and most affordable experimental framework.

While defining the framework, the first question for us to answer was if we should use absolute or relative performance measurements. Absolute measurements could be interpreted correctly only in the context of a standard benchmark with strictly defined implementation platform, language, middleware technology, and many other factors. Since we were not aware of any standard benchmark that would fit our goals, we saw little value in reporting absolute times to anybody who uses different implementation languages, ORBs, etc. or even their versions. Therefore, we decided to collect measurements relative to a reference model implemented with exactly the same programming and communication technologies as well as execution platforms.

The reference model we choose was an application system (AS) that has coupled application and authorization functionalities in one process with the former having exactly the same computational complexity as in the experimental configurations. Thus, by comparing performance of this and an experimental system, where authorization mechanism is encapsulated in CAAS, we could measure the difference in their performance.

The second question was about the meaning of performance in the context of this study. System performance has many meanings and multiple aspects. The way system performance is measured depends on how it is defined. If we defined the performance as a number of authorization requests served per a unit of time, or the latency time for each request, then we could have used time T_{caas} (Figure 7-1-b), when CAAS completes the pro-



(a) Reference Model **(b) Response Times for Configurations with CAAS**

Figure 7-1. Times for Measuring Performance

cessing of an authorization request, as the measure of CAAS performance. However, it, besides other reasons, would not allow us to have a reference model because there would be nothing to refer to. Nor did we decide to use time T_{as}, when an AS finishes processing an application request, which in turn contains time T_{caas}. Instead, we chose to measure response time T_c perceived by clients since it included response times at the other two points, and it was the main concern from the performance point of view, when authorization decisions were computed by CAAS. This is why our performance metric for CAAS is end-to-end response time that a client observes while interacting with an AS.

The definition of performance and the time representing it determined the reference model and the reference time shown in Figure 7-1-a. Using measured times T_c, and T_e, we calculated the percentage of response time increase I in the case of external authorization

for each configuration of CAAS with respect to embedded access control using the following formula:

$$I = \left(\frac{T_c}{T_e} - 1\right) \times 100 \quad (1)$$

Since CAAS first design was not optimized for concurrent access, we decided to leave performance scalability experiments outside of these experiments' scope. This is why for this study we measured run-time performance of CAAS in the presence of only one client, which sent requests to a single application system in a sequential manner as shown in Figure 7-1-b. That is, the client waited until it received the reply from its previous request before it made a new one.

We expected that, given the same complexity of authorization logic, the number of remote invocations made per each authorization request would effect the overall system performance the most. In its turn, the number depended on the composition of CAAS components and their location relatively to each other. For that reason we used different CAAS configurations to see how the composition of CAAS components affected the response time observed by the client.

7.2 CAAS Configurations

Given the multitude of different configurations that can be composed out of CAAS components, we needed to determine which of them should be used in the experiments. CAAS configurations determine the boundaries crossed by the messages sent during the computation of an authorization request. There are three types of these boundaries: object, process, and host. Note that whenever a message crosses process boundaries, it inevitably

goes through the ORB layer too. Thus ORB and process boundaries are considered as a one atomic layer. Whenever host boundaries are crossed, the messages travel over network as well. Thus by “crossing host boundaries” we imply traveling over network.

Another general observation important for understanding our choice of CAAS configurations is illustrated in Figure 7-2. Messages between CAAS components can travel in

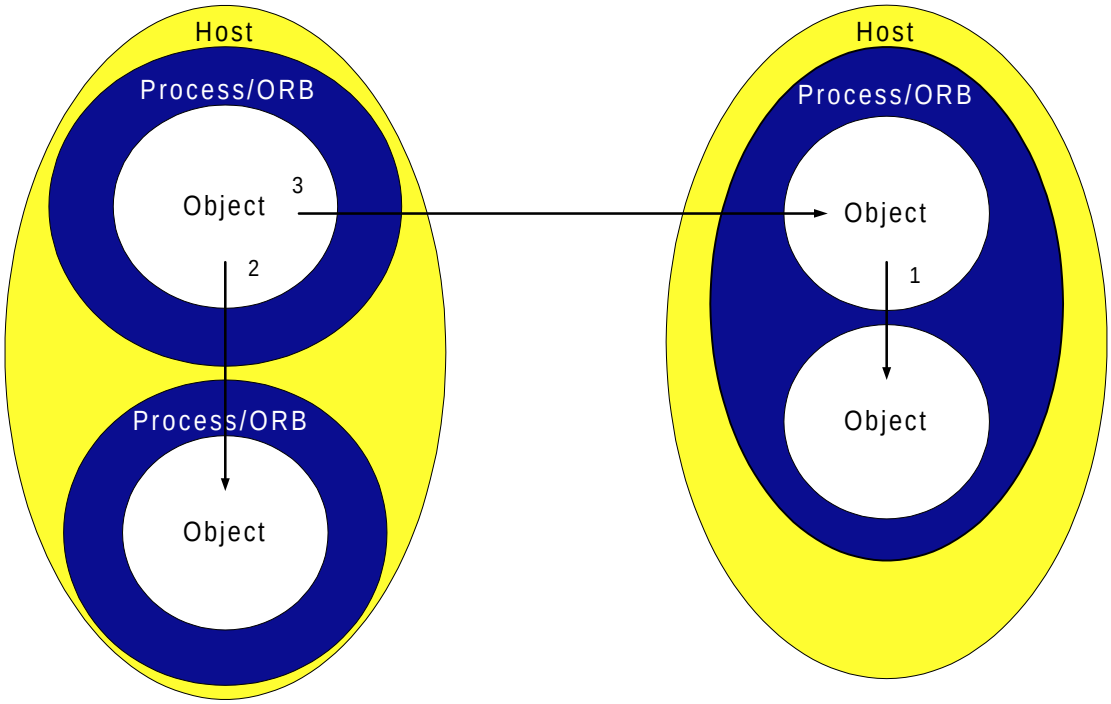


Figure 7-2. Boundaries Crossed by Messages

three ways: 1) from object to object in the same process, 2) from object in one process to an object in another process, and 3) between objects located in different processes, which, in their turn, are running on different hosts. There is a hierarchy of the boundaries: object, process, host. When a boundary is crossed, then all boundaries lower in the hierarchy are also crossed.

Messages cross *object* boundaries when components are co-located in the same address space and use direct method calls through JVM to communicate. Messages cross *process* boundaries when communicating components are co-located in the same host but run in their own processes. In this case, communication takes place through the ORB middleware, which is why we also call these boundaries as *middleware* boundaries. This form of communication, however, can take place using other mechanisms such as IPC [Nutt 1997, Stevens 1993]. Finally, messages cross *host* boundaries when components reside on separate hosts; this involves middleware and communication subsystem overhead.

CAAS can be deployed in many different configurations. When composing CAAS configurations, the main choice is the boundaries crossed between different components. We wanted to measure a wide range of boundary crossing configurations. On the one end of the range is a configuration when all CAAS components are collocated in one process and messages among them cross only object boundaries, which, we expected, would be the most efficient but the quantitative answer was not known. To highlight this, the corresponding CAAS configurations (shown in A, B, and D in Figure 7-3) end with word "Object." On the other end is the composition, in which all CAAS components are running on different hosts, which should yield the best flexibility and the worst overall performance. Again, we wanted to give a quantitative answer about the performance. We decided not to measure such a configuration because it seemed unlikely that anybody would use the service in this way. Instead, we tested cases when all the components are in separate processes, as shown in C, E and G. We also anticipated the use of a PE located on a separate host in case a legacy policy engine is utilized as a PE (configurations F and G).

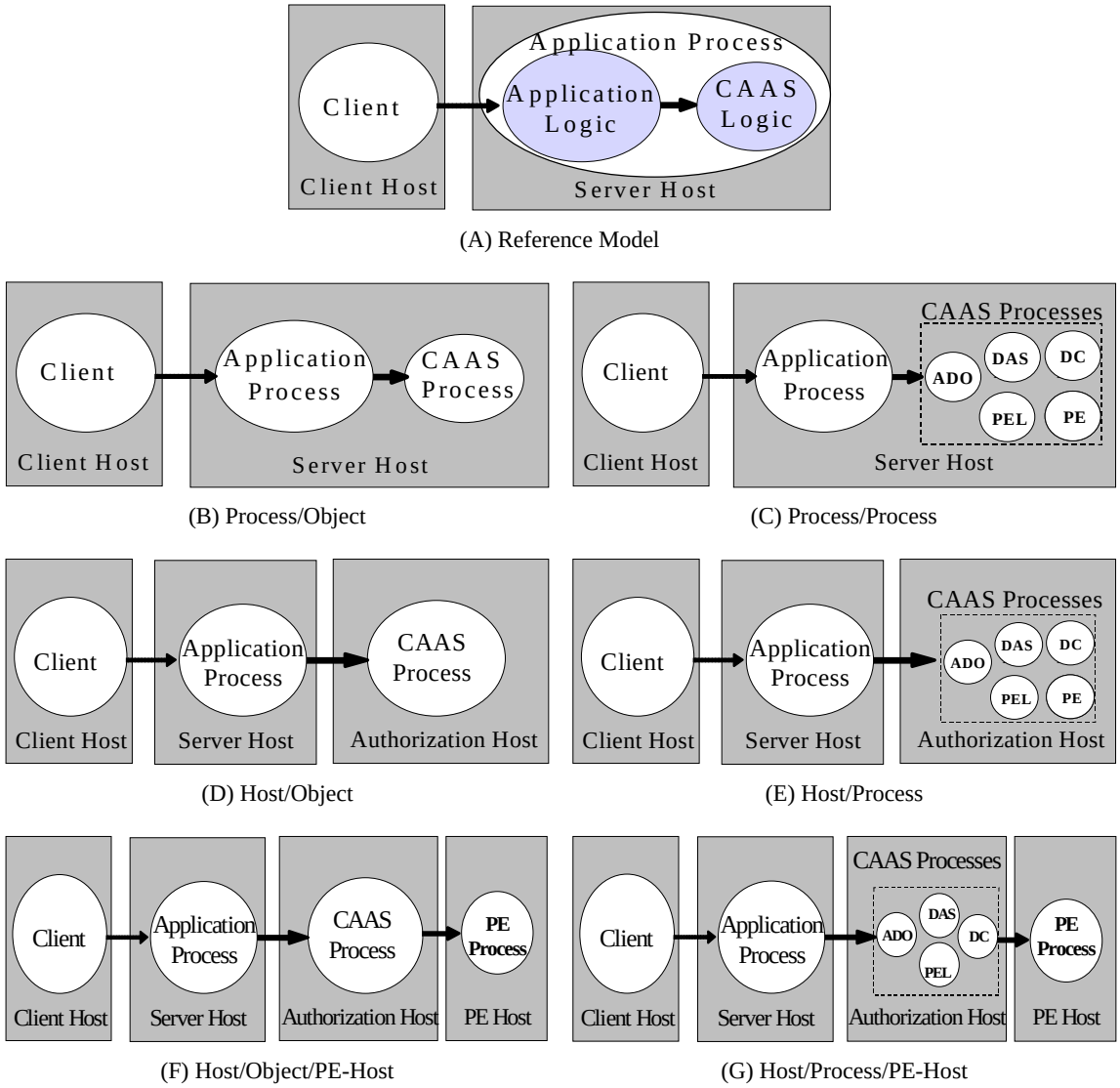


Figure 7-3. Reference Model and Experimental CAAS Configurations

We expected that application performance is affected not only by the type of boundaries the messages among CAAS components cross but also by the communication overhead associated with the messages between the AS and CAAS. This is why we measured the performance for configurations where CAAS is located on the same (B and C) and different (D--G) hosts as the application. To stress this difference, the names of the corresponding configurations begin with either “Process” or “Host.”

In order to produce relative performance measurements we needed a reference configuration that would have the authorization logic, with the same computational complexity as in all other configurations, coupled with the application logic. For this, we simulated our Reference Model by co-locating all CAAS components within the application process as shown in Figure 7-3-A. Our reasoning was based on the assumption that even though the code responsible for application and authorization logic could be highly coupled, it can be re-arranged into the equivalent code in such a way that it will allow for every computer operation to identify whether it contributes to application or authorization performance overhead. Once identified, it should be possible to encapsulate the authorization instructions into a separate application module.

Having the rationale behind CAAS configurations outlined, let us walk through and explain each of them. With *Process/Object* configuration, AS and CAAS are co-located as independent processes in the same server host, and CAAS components are co-located within the same process as illustrated in Figure 7-3-B. Messages between AS and CAAS are transmitted via ORB middleware (process boundaries) whereas CAAS components communicate using native method calls using the JVM (object boundaries). Figure 7-3-C shows *Process/Process* configuration where CAAS components are deployed in their own processes (process boundaries). In *Host/Object* configuration shown in Figure 7-3-D, CAAS components are co-located in the same process; however, AS and CAAS are on different hosts. That is, messages between AS and CAAS are delivered through the ORB middleware and communication subsystem (host boundaries) while messages among CAAS components cross only object boundaries.

In *Host/Process* (Figure 7-3-E), AS and CAAS are on different hosts, and CAAS components are in their own processes in the same host. Figure 7-3-F illustrates *Host/Object/PE-Host* configuration. This configuration is similar to *Host/Object* except that *PE* component runs in a different host. Communication among CAAS components incur object and host boundaries. Finally, in *Host/Process/PE-Host* configuration (Figure 7-3-G), *PE* is located in a host other than the authorization host while the other CAAS components run in different processes co-located in the authorization host. It is important to note that when two components exchange messages through process boundaries, message passing involves middleware overhead and possibly context switch overhead at the host where the two reside. Host boundaries, on the other hand, do not involve such context switch overhead since the communicating components do not compete with each other for execution time.

This configurability allows developers and administrators to deploy CAAS in a way to obtain maximum performance (by avoiding ORB middleware and network overhead), or flexibility (by having any component in any system in the network). For example, administrators may deploy CAAS using *Host/Object* configuration to avoid middleware and network overhead. However, in an organization where one or more PE components are remotely located (perhaps in a different subnet), CAAS can be deployed using *Host/Object/PE-Host* or *Host/Process/PE-Host* configurations. *Host/Process* or *Process/Process* configurations can be used to deploy CAAS components developed by third parties, which are not enabled to run in the same address space with other components. In a real scenario, we expect to see most components be co-located in the same process or host while one or more components, possibly PE, be deployed in remote locations.

After defining the measurement model, the reference and experimental configurations, the next question was what environment for conducting experiments should be used. This is discussed in the next section.

7.3 Test Environment

Our test environment was composed of 4 Gateway E-4200 400MHz Pentium III PC's running Windows NT Workstation 4.0 service pack 4. Each workstation had 128MB of physical memory, 139MB of swap space and its performance properties were set to maximum boost for foreground applications. Also, each workstation was equipped with an Intel PRO/100+ Management network adapter. These workstations interoperated on an 100Mb Ethernet with one hub, and connected to the rest of the campus network through a 100Mb switch. Furthermore, during testing we used JDK 1.1.7 and Visibroker 3.3 ORB, and all java classes and jar files were located on the local hard-drives. We used CORBA Naming service located on a separate host to discover the CAAS components and application. We carried out the performance measurements only when network utilization was less than 1% to minimize the effects of unrelated network load.

7.4 Experiment Procedure

Our experiment setting consisted of a client, an AS, and an instance of CAAS composed of one Access Decision Object (ADO), a Policy Evaluator Locator (PEL), a Dynamic Attribute Service (DAS), a Decision Combinator (DC), and a Policy Evaluator (PE). The goal of the performance measurements was to estimate a worst case performance penalty experienced by clients when CAAS serves authorization requests. We measured the response time T_c experienced by the client when external access control is implemented

using CAAS. Then, the response time T_e was measured. Using these two numbers, we calculated the response time increase *I* percentagewise using Equation 1. The DC object implemented logical AND combination policy while the PE object always granted access.

This procedure was repeated using all six configurations described in Section 7.2. Other parameters for our performance measurements were application processing (or business) logic time *B* and the number of authorization requests *N* generated for each client request. Application processing time represents delays experienced by an AS while serving client requests and enforcing authorization decisions returned by ADO. It does not include processing time incurred by CAAS. Although we used one client during the experiment, in an actual system, a client request can trigger any number of authorization requests by AS. This was simulated using a variable number of authorization requests per each client request.

It was an open question what authorization policies should be used for performance experiments. Since our goal was to measure a worst case performance penalty relative to the Reference Model, we used computationally least expensive combination and evaluation policies. This is because more complex authorization policies would increase computation overhead without increasing middleware and communication overhead, provided that no new inter-component messages are introduced. The increase in the computational overhead would occur within embedded authorization logic for the Reference Model as well as within CAAS while communication overhead would remain unaltered. The change can be illustrated by Equation 2, where Δ is the increase associated with the additional computational complexity of authorization logic. This means that $I' < I$, because $T_c > T_e$ and $\Delta > 0$.

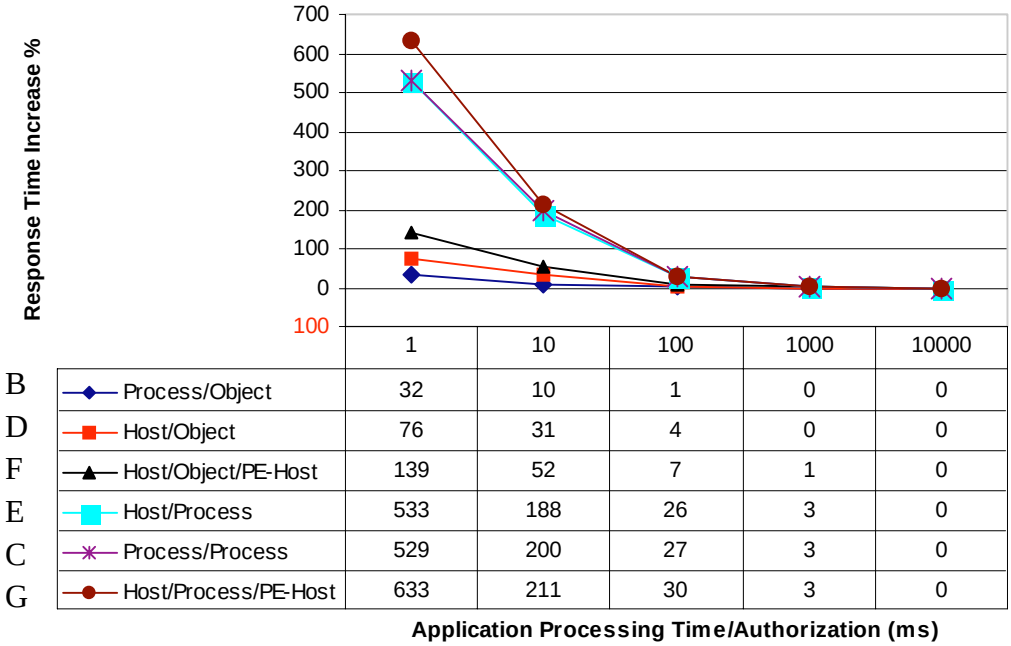
$$I' = \left(\frac{T_c + \Delta}{T_e + \Delta} - 1\right) \times 100 \quad (2)$$

For our performance experiments we did not use caching techniques. Caching was not considered since it reduces communication overhead and therefore reduces relative response time increase *I*. We did not utilize secure communications for remote invocations because the overhead due to the communication security is something we cannot control. Moreover, communication protection is application and implementation dependent. Different applications require different levels of protection, and different security products have different performance. As a result, we decided not to employ communication protection and estimate a worst case response time increase strictly in terms of middleware and communication due to RAD architecture.

7.5 Measurement Results

The measurements were carried out using CAAS configurations shown in Figure 7-3, and the results are illustrated in Figure 7-4. We calculated the increase of the response time as a function of application processing time per authorization request. For example, in the case of configuration *Host/Object* (Figure 7-3-D), the response time increased comparatively to the Reference Model by 31%, when the application was executing application logic for 10 ms each time before it would make an authorization request.

Two groups of CAAS configurations can be observed. The group with the best performance results consists of those configurations, in which all or most CAAS components were process co-located. Even when configured with the PE located on a host, separate from the one with all other CAAS components, CAAS performed better than in any other configuration from the second group. This group consists of CAAS configurations, when



		1	10	100	1000	10000
B	Process/Object	32	10	1	0	0
D	Host/Object	76	31	4	0	0
F	Host/Object/PE-Host	139	52	7	1	0
E	Host/Process	533	188	26	3	0
C	Process/Process	529	200	27	3	0
G	Host/Process/PE-Host	633	211	30	3	0

Figure 7-4. Response Time Increase for Various CAAS Configurations (Error size: ±0.5)

all the components interacted with each other via messages crossing process boundaries. It is worth noting that we used no ORB optimization for interprocess communications on the same host, although some ORBs have them.

The results imply that the amount of time spent executing application logic per each authorization request drastically affects the relative performance experienced by the application client. Its increase in the order of magnitude causes relative performance increase anywhere between 2 and 10 times in both groups. The results also revealed that those applications, which do not actively use the authorization service and spend one second or more executing application logic for each authorization request, are almost insensitive to CAAS configurations. This makes them the primary candidates for employing CAAS. But even more authorization-intensive applications can utilize such an authorization service as CAAS, if all the components are process co-located and the application owners can afford 10% decrease in the performance in return for all the benefits of RAD approach.

7.6 Performance Considerations

Our performance experiments suggest that in order to develop and utilize successfully RAD-based authorization services using current middleware technologies, several critical implications on the overall system performance should be considered. We summarize our findings in Table 7-1.

First, the pattern of using the authorization service by the application must be studied. As Table 7-1 shows, if the usage is mild (i.e. no more than one authorization request in

Limitations on Response Latency Increase	**Authorization Service Usage (app. logic time per request)**		
	Intensive (10ms or less)	**Medium (between 100ms and 1s)**	**Mild (10s or more)**
Strict (5% or less)	Authorization and application functions located in one process	CAAS components in one process	Any configuration or location
Medium (between 10% and 30%)	1.CAAS components in one process 2.CAAS and application on the same host	Any configuration or location	Any configuration or location
Lax (over 30%)	Any configuration or location	Any configuration or location	Any configuration or location

Table 7-1. Recommended CAAS Configurations Depending on Application Requirements

every 10 seconds or so), then such applications (last column) can use RAD-based services configured in any reasonable way. Second, the performance constraints imposed on the application should be used to understand where the application and the service should be located relatively to each other and how the service should be composed. For example, applications that can afford an increase of the response time due to the use of authorization service by more than 30% do not have to be host-collocated with the service, and the service can be configured in any way. On the other hand, applications intensively using the

service by making 100 or more authorization requests per second of application logic execution, and having strict performance constraints affording no more than 5% performance degradation, should have authorization and application logic process co-located.

The configuration and location requirements can be relaxed if some techniques for increasing performance in distributed systems are applied. For example, communication overhead can be minimized by using ORBs with communication layer optimized for objects located on the same host. Another technique is the caching of results previously obtained from the service components. The technique can be very helpful when authorization requests repeat over time.

The deployment and implementation of RAD-based authorization services should take into consideration the interactions among components. That is, described optimization techniques should be applied to components that have a high rate of interaction. For example, evaluations of policies that require more than one PE can be optimized by co-locating corresponding PEs with the appropriate DC in the same process.

Performance in the presence of concurrent requests is another aspect that should be taken into account. Although processing of concurrent requests were not part of our performance measurements, it is an aspect that warrants further research. Concurrency is not a trivial issue to handle in component-based systems. Safety preservation, the insurance that all objects in a system maintain valid states in the presence of concurrent access, requires the avoidance of read/write and write/write conflicts [Lea 1996]. To address this issue, we decided in the current implementation of CAAS to use fully synchronized methods. Although this property does not guarantee the system to be free of liveliness failures such

as deadlocks and resource starvation, it does guarantee consistency of values at the level of Java language object. Synchronized object instances are ready to be used in concurrent settings [Lea 1996] but this introduces unnecessary synchronization, which can affect overall run time performance because invocations of synchronized methods are more expensive, than to regular ones. If operations on CAAS components are synchronized, the granularity of synchronization should be carefully considered. Otherwise it can cause threads to block and unblock unnecessarily. The research issue, which is beyond the scope of this work, is the design of a RAD-based service optimized for concurrent access by multiple applications.

7.7 Conclusions

The main question about the feasibility of RAD approach, after its functional sufficiency, is whether RAD-based authorization services can deliver required performance. We have used CAAS to study the performance aspects of RAD architecture and found that there is no simple answer to the question. The experiments suggest that the two main factors affecting the performance are the ratio of the application execution time to the number of authorization requests and CAAS distribution configuration. Due to the variations of these factors the overall response time experienced by the application clients can increase as high as 600% and drop to as little as 1%. We identified several groups of applications that differ in their use of authorization service and the performance constraints. For each group, we determined what is required in order to assure adequate performance when RAD-based authorization service is used.

The service performance can be improved further if well-known techniques for optimizing distributed systems are used. For instance, utilization of the ORBs that optimize communications between objects located on the same host have the potential to significantly improve the response time. In situations when this is not possible, co-locating most CAAS components in the same address space in order to avoid middleware overhead will improve the performance as well. Also, caching the results previously obtained from various CAAS components will enhance the performance. These optimizations should focus on components with high rate of interaction such as DC and PE components. However, the performance might degrade when such properties as security of middleware communications are imposed.

Our measurements results are not only relevant to CORBA-based systems using RAD approach. The performance results for *Process/Object* and *Host/Object* configurations (Figure 7-3-B,D) can be used to estimate the response time increase for authorization servers in general.

8 Conclusions

Existing middleware technologies are necessary but not sufficient for effectively protecting the resources of distributed enterprise applications. In this dissertation, we proposed a two-tier approach that allows a comprehensive solution to the problem.

Foremost, we showed the adequacy of the CORBA authorization mechanism for the support of $RBAC_0$--$RBAC_3$ models and developed a framework for implementing them using CORBA Security. This delivers all the advantages of the external, scalable and yet comparatively fine-grain AC of CORBA Security along with the well studied powerful modeling concepts of RBAC. But our solution does not stop here because there are applications with more advanced requirements.

For those applications which require finer granularity than operation level and/or protection according to the policies that are difficult or impossible to model using just RBAC, we developed an architecture, RAD, for furnishing authorization decisions to such applications. It was shown via modeling, prototypical implementation, and performance experiments that the architecture features a number of important characteristics. These include separation of application and authorization logic, arbitrary granularity of protected resources, the use of information specific to the application domain, policy-neutrality, inherent consistency of AC enforcement across multiple applications, and high adaptability to various changes experienced by the enterprise environment.

Achieved via the use of either the CORBA authorization mechanism or a RAD-based service, the separation of authorization and application logic simplifies the development of both distributed systems and their security functions, and therefore makes it easier to enhance their quality. Equally important, it paves the way for uniformly utilizing authorization mechanisms across (heterogeneous) system boundaries, as well as for centralizing enterprise security administration and management, traditionally time consuming, costly and error prone processes.

By defining the state of the CORBA protection system, mapping it into RBAC models and developing RAD approach for application-level authorization, we created a structural foundation for modeling authorization architectures, which are central to the design of secure distributed enterprise applications.

8.1 Open Problems

Although our approach addresses the needs of most applications, the problem of engineering access control for distributed enterprise application resources is far from being solved. There are numerous open questions and opportunities for future research. Here we suggest some.

We developed two separate yet related steps. Each addresses the needs of a particular application group. An important question is how to integrate AC administration in uniform way if both solutions are employed. We can see two distinct ways. One is the use of a RAD-based service to furnish authorization decisions not only to application systems but also to middleware layers, and maybe even network, DBM and operating systems. Another fol-

lows the approach of policy agents where authorization rules are administered in a central location and then propagated into middleware and application AC mechanisms with the help of agents. Which of these two, or maybe some other, approaches is more attractive?

When authorization decisions are delegated to a service, one more potential performance and availability bottleneck emerges. New research is needed in order to understand how and what distribution techniques can be applied for achieving performance scalability and availability of authorization services based on RAD architecture.

The RAD approach is valuable for the solutions based on most middleware technologies. We implemented and conducted performance experiments with a CORBA-based prototype. Because information enterprises usually use more than one middleware technology, it is interesting to see how complex a design of multi-technology authorization service could be?

As with any complex software systems, the composition of applications with RAD-based services is not only to make constituent components work together, but also to ensure that the composition as a whole behaves consistently and guarantees certain end-to-end properties. Although this goal is beyond the scope of this dissertation, it is critical to model and design such compositions with the properties guaranteed even before the actual systems are deployed and composed.

The RAD approach has been proposed in the context of access control. However, we believe the approach can be applied to other security functionalities, for instance audit, non-repudiation, and communication protection. Can the architecture be re-used without any

significant changes? If not what should be different? Can a unified solution be proposed that would uniformly and comprehensively support the decisions regarding not only AC but most security functionalities that tend to be re-implemented in applications?

Bibliography

[Abrams 1991] M. Abrams, J. Heaney, O. King, L. J. LaPadula, M. Lazear, and I. Olson, "A Generalized Framework for Access Control: Towards Prototyping the Orgcon Policy," In Proceedings of National Computer Security Conference, 1991, pp. 257-266.

[Abrams 1990a] M. D. Abrams, K. W. Eggers, L. J. LaPadula, and I. W. Olson, "A Generalized Framework for Access Control: an Informal Description," The MITRE Corporation, McLean, Virginia, USA MP-90W00043, August 1990.

[Abrams 1989] M. D. Abrams, A. B. Jeng, and I. M. Olson, "Generalized Framwork for Access Control: An Informal Description," The MITRE Corporation, Springfield, VA, USA MTR-89W00230, September 1989.

[Abrams 1990b] M. D. Abrams, L. J. LaPadula, and I. M. Olson, "Building Generalized Access Control on UNIX," In Proceedings of USENIX workshop on UNIX Security, Portland, Oregon, USA, 1990, pp. 65-70.

[Amoroso 1994] E. Amoroso, *Fundamentals of Computer Security Technology*. Prentice Hall, 1994.

[Anderson 1972] J. Anderson, "Computer Security Technology Planning Study," Air Force Electronic Systems Division ESD-TR-73-51, Vols. I and II, 1972.

[Ashley 1997] P. Ashley, "Authorization for a Large Heterogeneous Multi-Domain System," In Proceedings of Australian Unix and Open Systems Gorup National Conference, 1997.

[Awischus 1997] R. Awischus, "Role Based Access Control with Security Administration Manager (SAM)," In Proceedings of the Second ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1997, pp. 61-68.

[Barkley 1995] J. Barkley, "Implementing Role-based Access Control Using Object Technology," In Proceedings of The First ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1995, pp. 93-98.

[Barkley 1999] J. Barkley, K. Beznosov, and J. Uppal, “Supporting Relationships in Access Control Using Role Based Access Control,” In Proceedings of ACM Role-based Access Control Workshop, Fairfax, Virginia, USA, 1999, pp. 55-65.

[Barkley 1998] J. Barkley and A. Cincotta, “Managing Role/Permission Relationships Using Object Access Types,” In Proceedings of The Third ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1998, pp. 73-80.

[Bell 1975] D. E. Bell and L. J. LaPadula, “Secure Computer Systems: Unified Exposition and Multics Interpretation,” MITRE, Bedford, MA, USA, Technical Report ESD-TR-75-306, March 1975.

[Benantar 1996] M. Benantar, R. Guski, and K. M. Troidle, “Access control systems: From host-centric to network-centric computing,” *IBM Systems Journal*, vol. 35(1), pp. 94-112, 1996.

[Bertino 1996a] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati, “Supporting Periodic Authorizations and Temporal Reasoning in Database Access Control,” In Proceedings of 22th International Conference on Very Large Data Bases, Mumbai (Bombay), India, 1996, pp. 472-483.

[Bertino 1996b] E. Bertino, S. Jajodia, and P. Samarati, “Supporting Multiple Access Control Policies in Database Systems,” In Proceedings of the IEEE Symposium on Research in Security and Privacy, Oakland, California, 1996.

[Beznosov 1997] K. Beznosov, “Applicability of CORBA Security to the Healthcare Problem Domain,” Object Management Group corbamed/97-09-11, September 1997.

[Beznosov 1998a] K. Beznosov, “Issues in the Security Architecture of the Computerized Patient Record Enterprise,” In Proceedings of Second Workshop on Distributed Object Computing Security, Baltimore, Maryland, USA, 1998.

[Beznosov 1998b] K. Beznosov, “Requirements for Access Control: US Healthcare Domain,” In Proceedings of Third ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1998, pp. 43.

[Beznosov 2000] K. Beznosov, “Information Enterprise Architectures: Problems and Perspectives,” School of Computer Science, Florida International University, Miami technical report 2000-06, June 2000.

[Beznosov 1999a] K. Beznosov and Y. Deng, “A Framework for Implementing Role-based Access Control Using CORBA Security Service,” In Proceedings of Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1999, pp. 19-30.

[Beznosov 1999b] K. Beznosov, Y. Deng, B. Blakley, C. Burt, and J. Barkley, “A Resource Access Decision Service for CORBA-based Distributed Systems,” In Proceedings of Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999, pp. 310-319.

[Blakley 1999] B. Blakley, *CORBA Security: an Introduction to Safe Computing with Objects*, First ed. Addison-Wesley, 1999.

[Bloomer 1992] J. Bloomer, *Power Programming with RPC*. O'Reilly & Associates, 1992.

[BullSoft 1995] BullSoft, “AccessMaster,” Bull Soft, 1995.

[Burrows 1990] M. Burrows, M. Abadi, and R. Needham, “A Logic of Authentication,” *ACM Transaction on Computer Systems*, vol. 8(1), pp. 18-36, 1990.

[CA 1998a] CA, “CA-ACF2 for OS/390,” Computer Associates International, 1998.

[CA 1998b] CA, “CA-Top Secret for OS/390,” Computer Associates International, 1998.

[CA 1999] CA, “Unicenter TNG: Product Information,” Computer Associates International, 1999.

[Caswell 1995] D. L. Caswell, “An Evolution of DCE Authorization Services,” *Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company*, vol. 46(6), pp. 49--54, 1995.

[CIST-NRC 1999] CIST-NRC, *Trust in Cyberspace*. Committee on Information Systems Trustworthiness, National Research Council. National Academy Press, 1999.

[Curry 1992] D. A. Curry, *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.

[DeBoever 1997] L. R. DeBoever, “Concept of “Highly Adaptive” Enterprise Architecture,” In Proceedings of Enterprise Architecture Conference, 1997.

[DEC 1989] DEC, “Guide to VAX/VMS System Security --- Version 5.2,” Digital Equipment Corporation, 1989.

[DHHS 1998] DHHS, “Security and Electronic Signature Standards; Proposed Rule,” 45 CFR Part 142, Department of Health and Human Services, 1998.

[DHHS 1999] DHHS, “Standards for Privacy of Individually Identifiable Health Information; Proposed Rule,” Department of Health and Human Services, 1999.

[Eddon 1999] G. Eddon, “The COM+ Security Model Gets You out of the Security Programming Business,” *Microsoft Systems Journal*, vol. 1999(11), 1999.

[Epstein 1995] J. Epstein and R. Sandhu, “NetWare 4 as an Example of Role-Based Access Control,” In Proceedings of Proceedings of the First ACM Workshop on Role-Based Access Control, Gaithersburg, Maryland, USA, 1995, pp. 71-82.

[Espinal 2000] L. Espinal, K. Beznosov, and Y. Deng, “Design and Implementation of Resource Access Decision Server,” Center for Advanced Distributed Systems Engineering (CADSE) - Florida International University, Miami technical report 2000-01, January 2000.

[Filman 1996a] R. Filman and T. Linden, “Communicating Security Agents,” In Proceedings of The Fifth Workshop on Enabling Technologies: Infrastructure for Collabarative Enterprises, Stanford, CA, USA, 1996, pp. 86-91.

[Filman 1996b] R. Filman and T. Linden, “SafeBots: a Paradigm for Software Security Controls,” In Proceedings of New Security Paradigms Workshop, Lake Arrowhead, CA USA, 1996, pp. 45-51.

[Fowler 1997] M. Fowler, *Analysis Patterns: Reusable Object Models*, First ed. Addison Wesley Longman, 1997.

[Gamma 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1995.

[Gittler 1995] F. Gittler and A. C. Hopkins, “The DCE Security Service,” *Hewlett-Packard Journal*, vol. 46(6), pp. 41-48, 1995.

[Giuri 1998] L. Giuri, “Role-Based Access Control in Java,” In Proceedings of Proceedings of the Third ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1998, pp. 91-99.

[Gligor 1986] V. Gligor, C. Burch, R. Chandersekaran, L. Chanpman, M. Hecht, W. Jiang, G. Luckenbaugh, and N. Vasudevan, “On the Design and the Implementation of Secure Xenix Workstations,” In Proceedings of

IEEE Symposium on Security and Privacy, Oakland, CA, 1986, pp. 102-117.

[Gong 1997] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," In Proceedings of The USENIX Symposium on Internet Technologies and Systems, Monterey, California, 1997, pp. 103-112.

[Grampp 1984] F. T. Grampp and R. H. Morris, "UNIX Operating System Security," *AT& Bell Laboratories Technical Journal*, vol. 63(8), pp. 1649-1672, 1984.

[Grand 1998] M. Grand, *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, vol. 1. Wiley Computer Publishing, 1998.

[Grimes 1997] R. Grimes, *Professional DCOM Programming*. Wrox Press Inc., 1997.

[Grimm 1999] R. Grimm and B. Bershad, "Providing Policy-Neutral and Transparent Access Control in Extensible Systems," *Lecture Notes in Computer Science*, pp. 317-338, 1999.

[Grimshaw 1998] A. S. Grimshaw, M. J. Lewis, A. J. Ferrari, and J. F. Karpovich, "Architectural Support for Extensibility and Autonomy in Wide-Area Distributed Object Systems," Department of Computer Science, University of Virginia CS-98-12, 1998.

[Grimshaw 1997] A. S. Grimshaw and W. A. Wulf, "The Legion Vision of a Worldwide Virtual Computer," *Communications of the ACM*, vol. 40(1), pp. 39-45, 1997.

[Grimson 2000] J. Grimson, W. Grimson, and W. Hasselbring, "The System Integration Challange in Health Care," *Communications of the ACM*, vol. 43(6), pp. 48-55, 2000.

[Hailpern 1990] B. Hailpern and H. Ossher, "Extending Objects to Support Multiple Interfaces and Access Control," *IEEE Transactions on Software Engineering*, vol. 16(11), pp. 1247-1257, 1990.

[Hale 1999] J. Hale, P. Galiasso, M. Papa, and S. Shenoi, "Security Policy Coordination for Heterogeneous Information Systems," In Proceedings of Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999, pp. 219-228.

[Heydon 1994] A. Heydon and J. D. Tygar, "Specifying and Checking UNIX Security Constraints," *Computing Systems*, vol. 7(1), pp. 9-12, 1994.

[Hommes 1990] R. Hommes, “VMS Security Architecture,” In Proceedings of DECUS Europe Symposium, Cannes, France, 1990.

[HP 1996] HP, “HP Adds Value to DCE Security Framework with Praesidium Authorization Server,” *DCE application development trends Magazine*, 1996.

[IBM 1976] IBM, *Resource Access Control Facility (RACF). General Information.* IBM Red Books, 1976.

[IEEE] IEEE, *IEEE P1003.6.1 Standard for Information Technology: Portable Operating System Interface (POSIX): Protection, Audit, and Control Interfaces*. IEEE Computer Society Press.

[IETF 1993] IETF, “RFC 1510, The Kerberos Network Authentication Service, V5,” Internet Engineering Task Force, 1993.

[Jonscher 1995] D. Jonscher and K. R. Dittrich, “Argos -- A Configurable Access Control System for Interoperable Environments,” In Proceedings of IFIP WG11.3 Ninth Annual Working Conference on Database Security, Rensselaerville, NY, 1995, pp. 39-66.

[Kaijser 1998] P. Kaijser, “A Review of the SESAME Development,” *Lecture Notes in Computer Science*, vol. 1438, pp. 1-8, 1998.

[Karger 1991] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, “A Retrospective on the VAX VMM Security Kernel,” *IEEE Transactions on Software Engineering*, vol. 17(11), pp. 1147-1165, 1991.

[Karjoth 1998] G. Karjoth, “Authorization in CORBA Security,” In Proceedings of Fifth European Symposium on Research in Computer Security (ESORICS), 1998, pp. 143-158.

[Kleinoder 1996] J. Kleinoder and M. Golm, “MetaJava: An Efficient Run-Time Mete Architecture for Java,” In Proceedings of Fifth IEEE International Workshop on Object-Orientation in Operating Systems, Seattle, WA, USA, 1996.

[Kong 1995] M. M. Kong, “DCE: An Environment for Secure Client/Server Computing,” *Hewlett-Packard Journal*, vol. 46(6), pp. 6-15, 1995.

[Lai 1999] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers, “User Authentication And Authorization In The Java Platform,” In Proceedings of Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999, pp. 285-290.

[Lampson 1991] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, “Authentication in Distributed Systems: Theory and Practics,” In Proceedings of ACM Symposium on Operating Systems Principles, Asilomar Conference Center, Pacific Grove, California, 1991, pp. 165-182.

[Lampson 1971] B. W. Lampson, “Protection,” In Proceedings of 5th Princeton Conference on Information Sciences and Systems, Princeton, 1971, pp. 437.

[LaPadula 1990] L. J. LaPadula, “Formal modeling in a Generalized Framework for Access Control,” In Proceedings of Computer Security Foundation Workshop III, 1990, pp. 100-109.

[Lea 1996] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 1996.

[Linn 1993] J. Linn, “Generic Security Service Application Program Interface,” Internet Engineering Task Force, Internet Draft RFC 1508, September 1993.

[Linn 1997] J. Linn, “Generic Security Service Application Program Interface,” IETF RFC 2078, January 1997.

[Luckenbaugh 1986] G. L. Luckenbaugh, V. D. Gligor, L. J. Dotterer, and C. S. Chandersekaran, “Interpretation of the Bell-LaPadula Model in Secure Xenix,” In Proceedings of DoD-NBS Conference on Computer Security, 1986.

[Maes 1987] P. Maes, “Computational Reflection,” in *Artificial Intelligence Laboratory*. Vrije Universiteit Brussel, 1987.

[McCauley 1979] E. J. McCauley and P. J. Drongowski, “KSOS -- The Design of a Secure Operating System,” In Proceedings of National Computer Conference, 1979.

[McInerney 1999] M. J. McInerney, *Windows NT Security*. Prentice Hall, 1999.

[McMahon 1995] P. McMahon, “Making the Internet Safe for Business,” *ICL Systems Journal*, vol. 10(2), 1995.

[Meyers 1997] W. J. Meyers, “RBAC Emulation on Trusted DG/UX,” In Proceedings of Proceedings of the Second ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1997, pp. 55-60.

[Microsoft 1998] Microsoft, “DCOM Architecture,” Microsoft, 1998.

[Molva 1992] R. Molva, G. Tsudik, E. V. Herreweghen, and S. Zatti, “KryptoKnight Authentication and Key Distribution System,” In Proceedings of Euro-

pean Symposium on Research in Computer Security, Toulouse, France, 1992.

[Mowbray 1997] T. J. Mowbray and R. C. Malveau, *CORBA Design Patterns*. Wiley Computer Publishing, 1997.

[Mowbray 1995] T. J. Mowbray and R. Zahavi, *The Essential CORBA: Systems Integration Using Distributed Objects*. Wiley Computer Publishing, 1995.

[Mullender 1990] S. J. Mullender, G. v. Rossum, A. S. Tanenbaum, R. v. Renesse, and H. v. Staveren, "Amoeba: A Distributed Operating System for the 1990s," *Computer*, vol. 23(5), pp. 44-53, 1990.

[NCSC 1987] NCSC, "A Guide to Understanding Discretionary Access Control in Trusted Systems," National Computer Security Center NCSC-TG-003, September 30 1987.

[Neuman 1993] B. C. Neuman, "Proxy-Based Authorization and Accounting for Distributed Systems," In Proceedings of International Conference on Distributed Computing Systems, Pittsburgh, Pennsylvania, 1993.

[Neuman 1994a] B. C. Neuman and T. Ts'o, "Kerberos: an Authentication Service for Computer Networks," *IEEE Communications Magazine*, vol. 32(9), pp. 33-38, 1994.

[Neuman 1994b] B. C. Neuman and T. Y. Ts'o, "Kerberos: an Authentication Service for Computer Networks," University of Southern California, Information Sciences Institute ISI/RS-94-399, 1994.

[Notargiacomo 1995]L. Notargiacomo, "Role-Based Access Control in ORACLE7 and Trusted ORACLE7," In Proceedings of the First ACM Workshop on Role-Based Access Control, Gaithersburg, Maryland, USA, 1995, pp. 65-69.

[NSF 1999] NSF, "Information Technology Research Program Requirements," National Science Foundation, 1999.

[Nutt 1997] G. Nutt, *Operating Systems: A Modern Perspective*. Addison-Wesley, 1997.

[OMG 1996a] OMG, "CORBAservices: Common Object Services Specification," Object Management Group, 1996.

[OMG 1996b] OMG, "Security Service Specification," Object Management Group, 1996.

[OMG 1997] OMG, “Clinical Observations Access Service RFP,” Object Management Group December 1997.

[OMG 1998a] OMG, “Interoperable Naming Service, Joint Revised Submission,” Object Management Group, document orbos/98-10-11, October 1998.

[OMG 1998b] OMG, “Person Identification Service,” Object Management Group, specification corbamed/98-02-29, February 1998.

[OMG 1999a] OMG, “The Common Object Request Broker: Architecture and Specification,” Object Management Group, Specification formal/99-10-08, 1999.

[OMG 1999b] OMG, “IDL to Java Language Mapping,” Object Management Group, Specification formal/99-07-53, 1999.

[OMG 1999c] OMG, “Resource Access Decision Facility,” Object Management Group OMG document number: corbamed/99-05-04, May 1999.

[OSF 1996] OSF, “Authentication and Security Services,” Open Software Foundation, 1996.

[Parker 1995] T. Parker and D. Pinkas, “SESAME V4 - Overview,” SESAME December 1995.

[Pedrick 1998] D. Pedrick, J. Weedon, J. Goldberg, and E. Bleifield, *Programming with VisiBroker: A Developer’s Guide to Visibroker for Java*. Wiley Computer Publishing, 1998.

[Pfleeger 1989] C. P. Pfleeger, *Security in Computing*. Prentice-Hall, 1989.

[Postel 1982] J. B. Postel, “RFC 821: Simple Mail Transfer Protocol,” University of Southern California Information Sciences Institute RFC 821, August 1982.

[Postel 1983] J. B. Postel, “TELNET Protocol Specification,” DDN Network Information Center, Request for Comments 854, May 1983.

[Postel 1985] J. B. Postel, “File Transfer Protocol,” DDN Network Information Center, Request for Comments 959, October 1985.

[Quarterman 1985] J. S. Quarterman, A. Silberschatz, and J. L. Peterson, “4.2BSD and 4.3BSD as Examples of the UNIX System,” *ACM Computing Surveys*, vol. 17(4), pp. 379-418, 1985.

[Riechmann 1997] T. Riechmann and F. J. Hauck, “Meta Objects for Access Control: Extending Capability-based Security,” In Proceedings of New Security Paradigms Workshop, Langdale, Cumbria, UK, 1997, pp. 17-22.

[Riechmann 1998] T. Riechmann and F. J. Hauck, “Meta Objects for Access Control: A Formal Model for Role-based Principals,” In Proceedings of New Security Paradigms Workshop, Charlottesville, VA USA, 1998, pp. 30-38.

[Rubin 1999] W. Rubin and M. Brain, *Understanding DCOM*. P T R Prentice Hall, 1999.

[Ryutov 2000a] T. Ryutov and C. Neuman, “Access Control Framework for Distributed Applications (Work in Progress),” Internet Engineering Task Force, Internet Draft draft-ietf-cat-acc-cntrl-frmw-03, March 9 2000.

[Ryutov 2000b] T. Ryutov and C. Neuman, “Generic Authorization and Access control Application Program Interface: C-bindings,” Internet Engineering Task Force, Internet Draft draft-ietf-cat-gaa-bind-03, March 9 2000.

[Ryutov 2000c] T. Ryutov and C. Neuman, “Representation and Evaluation of Security Policies for Distributed System Services,” In Proceedings of DARPA Information Servability Conference Exposition, Healton Head, South Carolina, 2000.

[Saltzer 1974] J. H. Saltzer, “Protection and the Control of Information Sharing in Multics,” *Communications of the ACM*, vol. 17(7), pp. 388-402, 1974.

[Sandhu 1996] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, “Role-Based Access Control Models,” *IEEE Computer*, vol. 29(2), pp. 38-47, 1996.

[Sandhu 1998a] R. Sandhu and Q. Munawer, “How to Do Discretionary Access Control Using Roles,” In Proceedings of ACM Workshop on Role-based Access Control, Fairfax, Virginia, USA, 1998, pp. 47-54.

[Sandhu 1998b] R. Sandhu and J. S. Park, “Decentralized User-Role Assignment for Web-based Intranets,” In Proceedings of the Third ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1998, pp. 1-12.

[Sandhu 1994] R. Sandhu and P. Samarati, “Access Control: Principles and Practice,” *IEEE Communications Magazine*, vol. 32(9), pp. 40-48, 1994.

[Schiller 1988] J. I. Schiller, S. P. Miller, B. C. Neuman, and J. H. Salzer, “Project Athena Technical Plan - Kerberos Authentication and Authorization System,” 1988.

[Schmidt 1999] D. C. Schmidt, “Dove: A Distributed Object Visualization Environment,” *C++ Report*, vol. 11(3), 1999.

[Schmidt 1998] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design of the TAO Real-time Object Request Broker,” *Computer Communications*, vol. 21(4), 1998.

[Simon 1997] R. Simon and M. E. Zurko, “Adage: An Architecture for Distributed Authorization,” OSF Research Institute, Cambridge 1997.

[Soley 1996] R. M. Soley and C. M. Stone, *Object Management Architecture Guide*, 3 ed. John Wiley & Sons, 1996.

[Stevens 1993] W. R. Stevens, *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1993.

[Sumner 1999] M. Sumner, “Critical Success Factors in Enterprise Wide Information Management Systems Projects,” In Proceedings of ACM SIGCPR Conference on Computer Personnel Research, 1999, pp. 297 - 303.

[Tardo 1991] J. J. Tardo and K. Alagappan, “SPX: Global Authentication Using Public Key Certificates,” In Proceedings of IEEE Symposium on Research in Security and Privacy, Oakland, California, USA, 1991, pp. 232-244.

[Thomas 1994] R. K. Thomas and R. S. Sandhu, “Conceptual Foundations for a Model of Task-based Authorizations,” In Proceedings of IEEE Computer Security Foundations Workshop, Franconia, NH, USA, 1994, pp. 66-79.

[USA 1996] USA, “Health Insurance Portability and Accountability Act, Public Law 104–191,” US Goverment, 1996.

[Varadharajan 1998] V. Varadharajan, C. Crall, and J. Pato, “Authorization in Enterprise-wide Distributed System: A Practical Design and Application,” In Proceedings of 14th Annual Computer Security Applications Conference, 1998.

[Walker 1980] B. J. Walker, R. A. Kemmerer, and G. J. Popek, “Specification and Verification of the UCLA Unix Security Kernel,” *Communications of the ACM*, vol. 23(2), pp. 118, 1980.

[Weiderhold 1992] G. Weiderhold, “Mediators in the Architecture of Future Information Systems: A New Approach,” *IEEE Computer*, vol. 25(3), pp. 38-49, 1992.

[Wilson 1997] W. Wilson and K. Beznosov, “CORBAmed Security White Paper,” Object Management Group corbamed/97-11-03, November 1997.

[Wong 1997] R. K. Wong, “RBAC Support in Object-Oriented Role Databases,” In Proceedings of the Second ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1997, pp. 109-120.

[Woo 1992] T. Y. C. Woo and S. S. Lam, “Authentication for Distributed Systems,” *Computer*, vol. 25(1), pp. 39-52, 1992.

[Woo 1993a] T. Y. C. Woo and S. S. Lam, “Authorizations in Distributed Systems: A Formal Approach,” In Proceedings of The 13th IEEE Symposium on Research in Security and Privacy, Oakland, CA, USA, 1993, pp. 33-50.

[Woo 1993b] T. Y. C. Woo and S. S. Lam, “Authorizations in Distributed Systems: A New Approach,” *Journal of Computer Security*, vol. 2(3), pp. 107-136, 1993.

[Woo 1993c] T. Y. C. Woo and S. S. Lam, “Designing a Distributed Authorization Service,” University of Texas at Austin, Computer Sciences Department TR93-29, September 1993.

[Woo 1993d] T. Y. C. Woo and S. S. Lam, “A Framework for Distributed Authorization,” In Proceedings of Conference on Computer and Communications Security, Fairfax, Virginia, USA, 1993, pp. 112-118.

[Woo 1998] T. Y. C. Woo and S. S. Lam, “Designing a Distributed Authorization Service,” In Proceedings of IEEE INFOCOM, San Francisco, 1998.

[Wreder 1998] K. Wreder, K. Beznosov, A. Bramblett, E. Butler, A. D'Empaire, E. Hernandez, E. Navarro, A. Romano, M. Tortolini-Taylor, E. Urzais, and R. Ventura, “Architecting a Computerized Patient Record with Distributed Objects,” In Proceedings of Health Information Systems Society Conference, 1998, pp. 149-158.

[Wulf 1996] W. A. Wulf, C. Wang, and D. Kienzle, “A New Model of Security for Distributed Systems,” In Proceedings of New Security Paradigms Workshop, Lake Arrowhead, CA USA, 1996, pp. 34-43.

[Yoder 1997] J. W. Yoder and J. Barcalow, “Archictectural Patterns for Enabling Application Security,” In Proceedings of Pattern Languages of Programming, Monticello, Illinois, USA, 1997.

[Zachman 1997] J. A. Zachman, “Enterprise Architecture: The Issue of the Century,” *Database Programming and Design*, pp. 44-53, 1997.

[Zurko 1998] M. E. Zurko, R. Simon, and T. Sanfilippo, “A User-Centered, Modular Authorization Service Built on an RBAC Foundation,” In Proceedings of Annual Computer Security Applications Conference, Phoenix, Arizona, 1998.

VITA

KONSTANTIN BEZNOSOV

	Born, Novosibirsk, Siberia, Russia
1987-1989	Military Service, Siberia, Russia
1991-1994	Assistant System Administrator and Analyst Information Technology, Budker Institute of Nuclear Physics Akademgorodok, Siberia, Russia
1993	B.S., Physics Novosibirsk State University Akademgorodok, Siberia, Russia
1994-1998	Research Assistant High Performance Database Research Center School of Computer Science (SCS) Florida International University (FIU) Miami, Florida, USA
1997	M.S., Computer Science FIU Miami, Florida, USA
1997-1999	Information Security Architect Information Technology Baptist Health Systems of South Florida Miami, Florida, USA
1998-1999	Co-chair, Security Special Interest Group Object Management Group (OMG)
1998-2000	Research Associate Center for Advanced Distributed System Engineering SCS, FIU Miami, Florida, USA
1999	Program Committee member OMG Workshop on Distributed Object Computing Security July 12-15, Baltimore, Maryland, USA

PUBLICATIONS

J. Barkley, K. Beznosov, and J. Uppal, “Supporting Relationships in Access Control Using Role Based Access Control,” In Proceedings of ACM Role-based Access Control Workshop, Fairfax, Virginia, USA, 1999, pp. 55-65.

K. Beznosov, Y. Deng, B. Blakley, C. Burt, and J. Barkley, “A Resource Access Decision Service for CORBA-based Distributed Systems,” In Proceedings of Annual Computer Security Applications Conference, Phoenix, Arizona, USA, 1999, pp. 310-319.

K. Beznosov and Y. Deng, “A Framework for Implementing Role-based Access Control Using CORBA Security Service,” In Proceedings of Fourth ACM Workshop on Role-Based Access Control, Fairfax, Virginia, USA, 1999, pp. 19-30.

K. Beznosov, “Requirements for Access Control: US Healthcare Domain,” position paper, In Proceedings of Third ACM Workshop on Role-Based Access Control, 1998, p. 43.

K. Beznosov, “Taxonomy of CPR enterprise security concerns at Baptist Health Systems of South Florida,” Baptist Health Systems of South Florida, 1997.

K. Beznosov, “CPR Security CORBA-based Security and Intranet Services: Object Technology Group Position Paper,” Baptist Health Systems of South Florida, 1997.

K. Beznosov, “Issues in the Security Architecture of the Computerized Patient Record Enterprise,” In Proceedings of Second Workshop on Distributed Object Computing Security, Baltimore, Maryland, USA, 1998.

L. Espinal, K. Beznosov, and Y. Deng, “Design and Implementation of Resource Access Decision Server,” Center for Advanced Distributed Systems Engineering, FIU, Miami, Technical Report 2000-01, January 2000.

OMG, “Resource Access Decision Facility,” Object Management Group, corbamed/99-05-04, May 1999.

K. Wreder, K. Beznosov, A. Bramblett, E. Butler, A. D'Empaire, E. Hernandez, E. Navarro, A. Romano, M. Tortolini-Taylor, E. Urzais, and R. Ventura, “Architecting a Computerized Patient Record with Distributed Objects,” In Proceedings of Health Information Systems Society Conference, 1998, pp. 149-158.

W. Wilson and K. Beznosov, “CORBAmed Security White Paper,” Object Management Group, document corbamed/97-11-03, November 1997.