DATA & KNOWLEDGE ENGINEERING

Consistent policy enforcement in distributed systems using mobile policies ☆,☆☆

Susan Chapin *, Don Faatz, Sushil Jajodia, Amgad Fayad

The MITRE Corporation, 1820 Dolley Madison Boulevard, McLean, VA 22102-3481, USA

Abstract

This paper briefly traces the evolution of information system architectures from mainframe-connected terminals to distributed multi-tier architectures. It presents the challenges facing developers of multi-tier information systems in providing effective consistent data policy enforcement, such as access control in these architectures. Finally, it introduces ''Mobile Policy'' (MoP) as a potential solution and presents a framework for using mobile policy in the business logic tier of multi-tier information systems.
© 2002 Elsevier Science B.V. All rights reserved.

Keywords: Security; Access control; Mobile policy; *n*-Tier architecture

1. Introduction

Multi-tier architectures separate application functions into three or more tiers, a presentation tier that handles interface with the user, one or more business logic tiers, and a data tier. Typically, the client or presentation tier is reduced to no more than a Web browser and the database management system (DBMS) is returned to its primary function of storing data (see Fig. 1). Business logic is moved from the client and the database to a middle tier, which consists of a Web server and application code [3,4,11,14]. The application code is hosted on a different platform
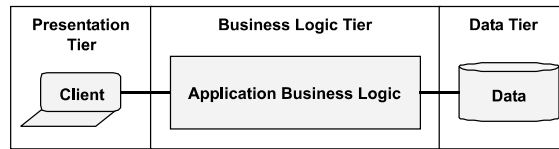
Fig. 1. Multi-tier architectures.

from the DBMS. The application code may request data from many DBMSs and may implement functions much more complex than satisfying requests by clients to access data in the DBMS. Support for multiple clients and databases is shown in Fig. 2.

Multi-tier architectures require changes in the way security and other policies are managed. Mechanisms are needed that can achieve consistent policy across elements of a distributed environment and that support flexible policies that address needs other than access control. The need for consistent policy management across distributed components is analogous to the need for consistent transaction management across distributed components. The need for flexible policies arises from the complex functionality of many multi-tier applications. While control over access to data remains a very important policy, support for other types of policies, such as requiring certain files to have a copyright notice or be sanitized in some way before they are returned to certain clients, is also needed.

Specific requirements for making policies consistent across different components include making policies mobile, so that they may travel with the data from one component to another rather than being applied before the data are released from the DBMS, and making security contexts mobile, so that references to users and roles have the same meaning to all components. Making policies flexible requires enabling those who manage data to define the kinds of policies supported, rather than relying on DBMS vendors. Traditional data tier policy management does not support these needs well, for several reasons: policies defined by DBMS vendors, are limited to access control policies; access control is applied by the DBMS at the time access to the data is requested, requiring that the source of the request be known to the DBMS; and traditional data tier users, roles, and policies are all locally defined within the DBMS based on local security context.

We propose an application framework that extends the capabilities for policy management in multi-tier applications without interfering with existing DBMS-based access controls. In our
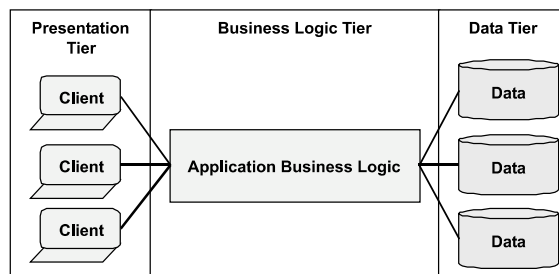

Fig. 2. Multi-tier architectures for multiple clients and databases.

framework, policy management is decomposed into three separable functions: defining policies, associating policies with data, and applying policies. Security context management is enhanced by including third-party mechanisms, such as digital certificates provided by a public key infrastructure (PKI) and attribute certificates, that can be referred to by all components. Almost any policy can be supported, limited only by the ability of developers to implement the policy in software. The proposed framework does not interfere with existing access control policy mechanisms. The framework allows policies to be applied in any tier of the application, determined by the application developers in conjunction with the database administrators. As of this writing we have developed our proposed framework design in sufficient detail to support a proof-of-concept prototype.

The rest of this paper is organized as follows. Section 2 describes the components of multi-tier architectures and the needs for policy management in multi-tier architectures. Section 3 describes the limitations of existing mechanisms for dealing with policy management in multi-tier architectures. Section 4 presents an overview of our proposed framework and the functions of the components, standardized vocabularies and method interfaces, and how the framework separates duties among development subgroups. It also describes how security contexts, as well as policies, can be shared among application components. Section 5 summarizes what we have achieved and what we want to achieve in the future.

2. Multi-tier architectures

Multi-tier architectures are an evolution of approaches to connecting a user to services informed by data. Architectures for such applications must provide three functions: data storage and retrieval, application business logic, and interface with the user.

The first approach, historically speaking, was for clients to connect to servers using dumb terminals. All three functions were provided by the server, usually in a single integrated application. Terminal to server architecture is shown in Fig. 3. Later the availability of intelligent workstations allowed the application to be split between the client and the server. Data storage was provided by the server, client interface was provided by the client component, and application business logic was provided by both components, with functionality divided according to the needs of the application. Client/server architecture is shown in Fig. 4.

More recently, a third approach has become common, largely in response to the development of Web capabilities. Application-specific client components that support business logic, called *fat clients* or *thick clients*, are difficult and expensive to deploy and maintain. Administrators would like to replace them with *thin clients* that can be deployed once and used to support multiple applications. In multi-tier architectures thin clients are responsible for interfacing with the user,
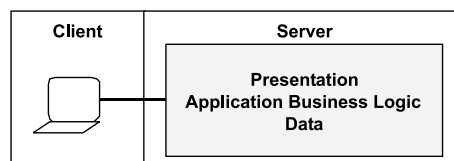

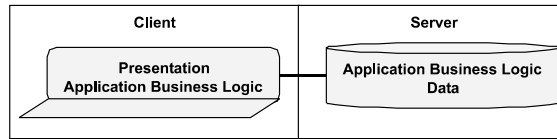Fig. 3. Terminal to server architecture.

Fig. 4. Client/server architecture.

but do not themselves determine what information to display. All business logic is offloaded to server components. Multi-tier architectures are shown in Figs. 1 and 2.

Web browsers are the most typical of these thin clients; they are automatically available on most user workstations and can be used to support many different applications. While it is true that Web browsers also support the automatic download of thick client software, in the form of mobile code such as ActiveX and Java Applets, many enterprises prefer to turn off mobile code functionality in response to concerns about security and network bandwidth.

Thin clients are responsible only for *presentation*, which consists of displaying information to users and collecting information from users. Web browser thin clients are connected to Web servers, instead of directly to a DBMS server. The Web server is associated with an application server capability, which supports the application business logic that determines what the Web browser should present to and collect from the user. This application server environment is the *business logic tier*.

At the data store end, the *data tier*, there are good reasons for moving business logic out of the data storage systems, the DBMSs, into the business logic tier. As long as you have a Web application server, you have a platform that supports business logic. Moving all the business logic to this platform, instead of developing it within the DBMS, has a number of advantages for development support and run-time effectiveness.

Advantages for development support include:

- One application can access multiple databases on multiple DBMSs, without requiring coordination among developers of code based on the multiple DBMS platforms.
- An application can call legacy applications and databases without requiring the legacy systems to be extensively modified.
- One database can be referenced by more than one application.
- The business logic application can be a coordinated, single application written on a single platform.
- DBMSs were developed to store data and provide it on request. They do this extremely well. They are less praiseworthy as application development platforms. Compared to integrated development environments that support C++, Visual Basic, Enterprise Java Beans (EJB), and other component or object-oriented environments, DBMS development environments are limited in developer support and language capabilities.

Advantages for run-time effectiveness include:

- enhanced scalability by supporting many clients with few resources,
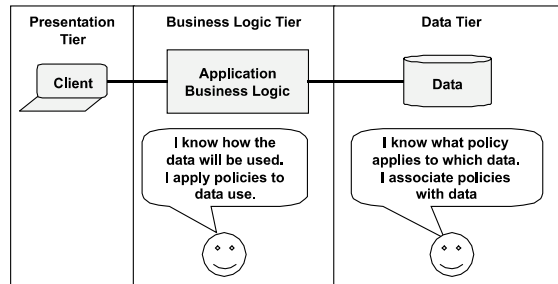- enhanced efficiency through the use of database connection sharing.

Fig. 5. Multi-tier architecture policy management.

The downside of multi-tier application architectures is that they can be quite complex. The presentation tier can consist of any one of a number of browser products on any one of a number of platforms. The data tier can consist of multiple databases in different DBMSs on different platforms. The business logic tier can consist of multiple components on multiple different platforms.

The problem is that all these different components need to be composed together into a single application. Where security is involved, the composition must be seamless and reliable. Composing policies in multi-tier applications can be an issue, because the developers of the different components of the business tier and the different databases have different responsibilities and knowledge about the policies that should apply. The distribution of knowledge and responsibilities is illustrated in Fig. 5. We address policy composition in multi-tier architectures by providing a framework that allows the developers of each component to concentrate on the policy management functions that are properly their responsibilities.

We develop our solution for policy composition in Sections 2.1.–2.3. We discuss aligning the authority for policy with the responsibility, managing identity credentials, and making security contexts mobile. We then discuss how with our solution policies can be extended beyond access control.

2.1. Aligning the authority for policy with the responsibility

An architecture that reserves policy application to the DBMS requires extensive communication among various subgroups within the enterprise. Managing a policy has three components, and each component is, ultimately, the responsibility of different enterprise subgroups:

- *Policy specification* is properly the responsibility of enterprise management.
- *Policy association* is the process of associating enterprise policy with data. It is properly the responsibility of the data owners, usually represented by the database administrator (DBA).
- *Policy application* is the process of applying the policy to data at the appropriate time. It is properly the responsibility of the business logic developer or whoever is in charge of the point(s) at which the data are used. Applying the policy at time of use is an ongoing activity; the data may be considered to be ''used'' when it is accessed within the DBMS, but it is also ''used'' within the application, whether the application code is located within the DBMS or

in a separate middle tier component, and whether the application uses the data immediately or holds on to it for several days before use.

Policy enforcement is only complete when all three elements, definition, association, and application, work harmoniously together. The problem is that building a coordinated response to policies can require close cooperation among those responsible for each component. It is not that any of the policy management problems are inherently impossible to solve. The problem is that they require cooperative design decisions affecting application code in both the middle tier and the DBMS, and these two portions of the application may be developed by different groups of people on different time schedules. The result is a greater risk of miscommunication and decreased assurance that the resulting product will correctly implement the policy.

A mechanism is needed that decouples the development of software that implements the policy from the process of associating the policy with data and the development of software that applies the policy at time of use.

2.2. Managing identity credentials

Management of identity credentials may present design problems because in multi-tier applications the client does not connect directly to the DBMS. Most typically, such application architectures are built from a Web browser on the client, a Web server and one or more applications in the middle tier, and one or more data stores in the data tier. Problems with authentication in the middle tier include communicating user credentials from the client through the middle tier to the DBMS, authenticating all applications in the chain from the client to the DBMS, and supporting high-performance architectures.

Communication of user credentials is a problem because, if authentication in the DBMS is to be based on the identity of the user who originates a request for access, then a mechanism is needed to pass the credentials from the originating client through the business logic tier(s) to the DBMS.

One possible mechanism for communicating credentials is for the middle tier to impersonate the client. With this mechanism, the middle tier opens a connection to the DBMS while impersonating the originating client, and the DBMS uses traditional DBMS authentication and authorization mechanisms. Requirements for this mechanism are (a) the DBMS must trust the middle tier both to authenticate the actual client and to impersonate the actual client and not some other entity, and (b) the client, the middle tier, and the DBMS must share a security context. [1]

Another possible mechanism is for the middle tier to pass the client's digital certificate to the DMBS. With this mechanism the middle tier and the DBMS both trust the PKI to authenticate the actual client, in effect sharing the same security context, and the DBMS trusts the middle tier to pass it the certificate belonging to the actual client and not some other identity.

---

[1] A security context defines the semantic meaning of an identity and its attributes. Without a shared security context, the DBMS might understand the identity ''Jenny Blaise'' to point to a sales manager in the enterprise, while the middle tier understands the identity ''Jenny Blaise'' to point to a reporter for a tabloid newspaper.

With either mechanism, chain of delegation is a problem because, in architectures where the DBMS and the client are not in direct communication, the DBMS has no choice but to trust intermediary applications either to transmit the credentials presented by the actual client or to impersonate the client. For this reason, the DBMS must authenticate all intermediary applications as well as the originating client.

Performance can be a problem because traditional DBMS authentication occurs when a connection is established and the authentication identity persists only as long as the connection is maintained. However, establishing and maintaining a connection to a DBMS is expensive in terms of time and resources. In systems where support for a large number of simultaneous connections is important, such as e-commerce systems, any authentication/authorization mechanism that depends on a dedicated connection to the DBMS for each originating user is an unacceptable solution. High-performance multi-tier architectures typically rely on opening a dedicated connection between the middle tier and the DBMS, and using that connection to access the DBMS as required to satisfy requests from multiple clients. [2]

For these reasons, many currently fielded multi-tier systems do not attempt to authenticate the originating client in the DBMS, or even to inform the DBMS of the identity of the originating client. Instead, the middle tier connects to the DBMS under its own identity, and the DBMS trusts it with a set of authorizations associated with the middle tier application rather than the originating client.

To deal with these two problems, a mechanism is needed that allows a more granular level of control in multi-tier architectures, even though that control may not be as strong as traditional access control applied to client/server architectures, because it will be stronger than the practical alternatives for multi-tier architectures.

2.3. Making security contexts mobile

Traditional DBMS policies depend on the DBMS identifying and proving the unique identity of the client who is requesting access to the data. The client usually represents a user, though it may be an application running under its own identity. In either case, the authenticating system has prior knowledge of all potential clients. In our area of discussion, DBMS authentication, this knowledge is traditionally stored directly in the DBMS. Other possible implementations include storing the identities and authorizations of potential clients in an external trusted directory and developing code in the DBMS to use this external information for authentication.

There are two problems with this approach:

- The pool of potential clients may include clients whose identity is not known to the system managers before the clients attempt to connect.
- If a chain of impersonation is involved, then the DBMS concept of user identity must be synchronized with the middle tier's concept of user identity.

---

[2] To be more accurate, we should discuss connections from individual components of the middle tier, rather than the middle tier as a whole. However, this distinction is not relevant to the argument being presented, however significant it may be to middle tier application developers.

2.3.1. Pool of potential clients

For some applications, the pool of potential clients may be unknowable because expansion of the Internet has led to expansion of application client bases. Many e-commerce applications more-or-less consider everyone in the world to be a potential client; this set of potential clients is simply too large to predefine each potential identity. Extranet applications include employees of other enterprises in the potential client base; employees of other enterprises are not readily known to database administrators. Even within an enterprise, the pool of predefined clients needs to be synchronized with the enterprise's master file; solutions exist for this but they may be restrictive.

Because for many applications the number of potential clients can be too large to predefine, a mechanism is needed that associates the client with some characteristic that can be known before the client makes a request that requires access to data. This characteristic might be identity if the client is a previous customer, or it might be some other attribute that places the customer in a predefined group or role.

2.3.2. Synchronizing concepts of user identity

To deal with these problems, a mechanism is needed that identifies users consistently in all tiers. Today the most promising such mechanism is use of identity and attribute digital certificates validated by a trusted PKI. Attribute certificates facilitate role-based access control (RBAC) by providing a digitally signed list of user attributes [5,6]. Other mechanisms may be developed in the future. Whatever mechanisms are available, system architectures need the flexibility to make use of them.

One way to maintain security context support is to have a mapping between the identity and attributes that the DBMS requires for access to a particular piece of data and the identity and attributes that the application logic understands. For example, if the DBMS requires the attribute "manager" for data access and the application logic's equivalent term for "manager" is "supervisor", then a mapping between "manager" and "supervisor" can insure that context is maintained between the DBMS tier and the application server tier (see [5,6] for additional details).

2.4. Making policies general

"Policy" is often taken to mean "security policy", and "security" is often taken to mean "access control", and all access control is often expected to be handled by the same mechanisms. Although security policies are important policies, and access control is important to an enterprise's overall security, these are not the only policies that an enterprise may want to enforce, and not all policies, access control or other types, are equally important.

2.4.1. Different types of policies

A substantial part of most middle-tier application development involves implementing various kinds of policies. Some policies are enterprise policies that are specified by enterprise management as rules that must be followed. Examples of enterprise policies include requirements for ensuring that files have the proper copyright notice before they are released outside the enterprise, degrading the resolution of certain images before they are released to specified classes of clients, and scanning files for viruses before they are used.

Other rules are local to the application but span both the business logic tier and the data tier. It is a bit of a stretch to call these application rules "policies", but it is convenient for our discussion because they share many of the characteristics of enterprise policies. In particular, they may be as critically important and as much in need of assurance that they are working correctly as enterprise policies, and can equally well be handled by our proposed framework.

An example of one of these other "policies" is a rule that defines the confidence that the middle tier application can have in the accuracy of a data item retrieved from a DBMS. Imagine an application that controls airplane takeoffs. One of the data items it needs is the amount of fuel in the plane's tank. The rule might be that the confidence level of this type of data is a function of metadata, such as the time since the data elements were last updated, rather than something that can be derived from the data value itself. The application as a whole, including both the middle tier and the DBMS, needs a mechanism to calculate the confidence factor and get that information to the middle tier before the middle tier releases the plane for takeoff, or some considerable unpleasantness might ensue.

A mechanism is needed that supports any policies that may be applicable to data, using the same techniques for any policy, without requiring that the set of policy types be predefined by DBMS vendors.

2.4.2. Different levels of criticality

A characteristic of this expanded definition of policies is that not all policies are equally critical. Some types of policies may be less critical than others in an enterprise; for example, the need to check files for copyright notice may be less critical than protecting write access to the salary file. Even within access control, some data may need to be protected more carefully than others. For example, the author of a document may wish it to be restricted to only a small group of people while it is under development, but the accidental release of the partially written document to other employees would not have as severe consequences as the accidental release of the company's product source code to the general public.

Therefore, a mechanism that is not deemed sufficiently secure for one policy may still be acceptable, and very valuable, for other policies. The requirement is that the mechanisms must not interfere with each other.

3. Related work

As information system architectures have moved from client-server to multi-tier, administration of security policy has become difficult. Generally, each component in a multi-tier architecture that has policy enforcement responsibilities maintains its own policy information that must be manually configured. Hence, it becomes the responsibility of system administrators to assure that all policies are consistent.

Several research efforts are currently under way to centralize the administration of policy. The Open Group's Adage project [12,17] is a typical example of this research. A central policy definition and storage capability is used by administrators to define and store all the policies needed throughout the distributed system. These policies are then translated into policy information suitable for the various enforcement mechanisms used throughout the system. Systems like Adage

assume a single central authority that defines all policy. Further, they assume that this single authority has administrative control of all elements of the multi-tier distributed system.

Mobile policy takes a different view, assuming a much more distributed definition of policy and administrative control of the system. Mobile policy allows policy to be defined by an authority close to the system element that is responsible for the information being controlled. Then, that system element shares the policy with other system elements that use its data.

The notion of mobile policy is not particularly new. Several approaches to sharing policy information have been developed. However, none is as general as the approach proposed here.

One problem common to all attempts to centralized policy definition and storage is the need for a semantically rich policy specification language capable of representing all policies that may apply within the multi-tier system. Such a language is very difficult to define and has so far eluded researchers. Mobile policy tries to avoid this problem by encapsulating policy in an executable module. These modules can be coded using any programming or policy definition language that the policy administrator chooses. Instead of defining an all-powerful magic policy language, the problem is transformed into defining a shared vocabulary of inputs to and outputs from policy modules. These vocabularies should be more tractable than a general-purpose policy language.

Information labeling, as typified by NIST FIPS PUB 188 [13], is a mechanism for sharing with data consumers the policy that should be applied to data. However, when using labels, it is assumed that all system elements that exchange data already share a common definition of policies that might apply to data. The label is a pointer to the particular policy to be applied to a piece of data. For example, a label of classified does not define policy for a piece of data but instead tells the recipient to enforce his policy for classified data when using this piece of information.

The policy server in Secure Computing Corporation's (SCCs) Distributed Trusted Operating System (DTOS) [1,7,8] is an example of mobile policy similar to the approach proposed here even though it requires a central policy specification authority. DTOS is a high-assurance version of the Mach microkernel operating system. In microkernel operating systems, a very small kernel is built (the microkernel) and many of the services associated with traditional operating systems are added as servers on top of the microkernel. SCC wanted to implement discretionary access control as a server outside the kernel. The kernel would enforce policy, but the policy being enforced would be defined outside the kernel by a policy server and could be easily modified. Each time a server running on the microkernel attempted to access a microkernel-controlled resource, the microkernel would consult the policy server to determine if the requested access was allowed.

As might be expected, the need for the microkernel to check with the policy server on every resource access request introduced significant overhead and severely degraded performance of the operating system. To deal with this, SCC added an "access vector cache" to the microkernel. When a process first requests access to a microkernel-controlled resource, the microkernel contacts the policy server and gets an access vector for that process from the policy server. The access vector defines all of the resources that the subject process is allowed to use. On subsequent requests from that process for access to microkernel-controlled resources, the microkernel consults the cached access vector instead of contacting the policy server. This significantly improves performance.

Access vectors are a form of mobile policy and were the first approach considered in this work. However, access vectors are well suited for use in an operating system but not as a policy mechanism for data in a multi-tier information system. Access vectors in DTOS are a fixed-length

bit field. This is possible because the set of resources being controlled by the microkernel is static and completely defined before the system begins operation. Unfortunately, in a distributed system the set of data available for use or the set of potential users is not static nor is it predefined before the system begins operation.

SCC did provide a capability not addressed in the mobile policy framework presented here that may be useful and should be considered for future work. Access vectors are considered cached copies of actual policy information. The policy information in the policy server is the definitive definition of the policy to be enforced. As such, the policy in the server could change during system operation invalidating one or more cached access vectors. SCC provided a mechanism for the policy server to notify the microkernel that policy changes have occurred. When this happens, the microkernel invalidates the entries in the access vector cache and contacts the policy server on the next request for access to a resource to get a new access vector consistent with the new policy. The framework for mobile policy presented here does not yet address policy changes.

While the mobile policy framework presented here was being developed, the Object Management Group's (OMGs) Common Object Request Broker Architecture (CORBA) medical systems (CORBAmed) domain task force (DTF) was developing a framework for access control decision making within business objects called Resource Access Decision (RAD) [9,10]. CORBA business objects are essentially business logic tier components. RAD deals only with access control decisions and does not define either where access control policy comes from or how it is administered. However, the RAD framework does include several of the elements found in the mobile policy framework presented here.

The next section describes the framework for use of mobile policy in multi-tier information systems.

4. The proposed framework

We call our proposed framework MoP, for mobile policies. MoP allows the separation of policy definition, policy association, and policy application into separate operations that can be performed by different people without requiring them to share the details of the policy with each other. Policies, once defined, are associated with data in the database. When data move from one component or tier to another, any associated policies travel along with the data until the policies are applied. Mobile policies are shown in Fig. 6.

4.1. System overview

The primary goal of the framework is to support separation of duty among application component developers by moving policy from the database to the application along with the data while minimizing the knowledge that must be shared between data tier developers and business logic tier developers. Secondary goals are to minimize effort on the part of application developers, support assurance that the system works as intended, work harmoniously alongside existing policy mechanisms, support multiple application and DBMS platforms, and minimize the impact on performance.
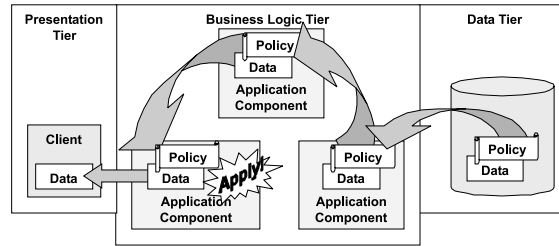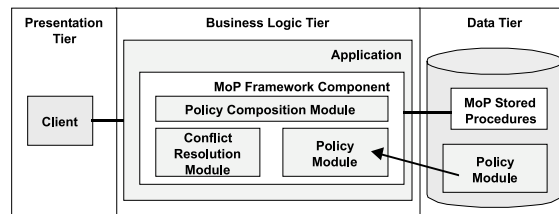
Fig. 6. Mobile policy.

Fig. 7. MoP component types.

The framework consists of five code component types and a set of standards for how they are used and developed. The component types are:

- Policy module. Policy modules implement policy rules.
- Policy composition module. Deals with issues such as the order in which policies are to be applied. Bonati et al. [2] propose an algebra for security policies with a translation mechanism to logic programs in order to facilitate policy composition even in the case where the policies are expressed in different formats.
- Conflict resolution module. Conflict resolution modules resolve conflicts among policy modules.
- MoP stored procedures. The MoP stored procedures implement MoP framework component logic that resides within each DBMS.
- MoP framework component. The MoP application component implements the mechanisms for using the framework within an application.

The component types are shown in Fig. 7.

Standards for the use and development of the components are the glue that makes the system work. MoP specifies two kinds of standards, *interface* standards that specify how one component may call another, and *vocabulary* standards that represent the minimal knowledge that must be shared among the developers of systems that use MoP.

4.2. Function of components

This section describes the MoP component functions.

4.2.1. Policy module

A *policy module* is an executable code module written for the platform of choice of the application. Each policy module implements one specific policy rule. For example, a policy module may determine whether a requested access is granted based on user identity, or whether access is granted based on the type of connection between the client and the application, or it may add the correct copyright notice to a file. Thus, each policy module is a self-contained package with a limited, specific function, which has the nice benefit that it simplifies validation of correct behavior.

Policy modules are classified into types by the end function they perform, not by the rule that governs how they perform it. The three examples above include only two policy types: *determine whether access is granted* and *add a copyright notice*. The two access granting rules, one of which looks at user identity and the other of which looks at the client connection, would be implemented in two separate policy modules, both of which are of type "access grant."

All policy modules of the same function type return the same output parameters with the same syntax and semantics. An application programmer needs to know what the module does in order to determine whether the module is applicable to the planned use of the data, and what output parameters the module returns and what they mean in order to code an appropriate response, but the application programmer does not need to know the policy rule the module implements.

In contrast, not all policy modules of the same type require the same input parameters. All policy modules implement a method that returns a list of input parameters. The application must be able to accept a list of parameters selected from a predefined set and return a value for each.

The scope of a policy module may vary. It may be developed and used within a single application, department, or enterprise, or eventually there may be well-known policy module components that are widely available.

We envisage that policy modules will come from three sources. At first, policy modules will be custom-built by enterprise developers to implement enterprise policies. Later, if MoP becomes widely used, policy modules that implement common policies may become commodity items. Finally, we plan to investigate automatically extracting existing DBMS access control specifications from the DBMS and creating MoP policy modules dynamically when access to data is requested.

To summarize, a policy module is an executable code module that implements a single rule, has a well-known type and set of output parameters, and produces a list of required input parameters.

4.2.2. Conflict resolution module

Multiple policy modules may be associated with the same data set. If it should happen that more than one policy module of the same type is associated with the same dataset, then any conflicts must be resolved before the correct single output parameter set is defined. This conflict resolution is performed by a *conflict resolution module*.

Conflict resolution modules can be simple or complex, depending on the policy module type (see [2]). For example, a conflict resolution module for access grant policy modules might be very simple, just the logical AND of Boolean values meaning OK or Not OK, while the conflict resolution module for copyright notices might be very complex, requiring choosing among several sets of alternative copyright statements based on some external information.

We assume that different policy module types are independent of each other. Any interactions between, say, a copyright notice rule and an access grant rule we consider to be idiosyncratic, complex, and outside the scope of the MoP framework. MoP, of course, does not prevent the application developer writing code to resolve any such conflicts.

Conflict resolution module development is closely linked to policy module development. Conflict resolution modules implement the resolution of conflicts among policy rules, and therefore conflict resolution rules are in effect policy rules.

The scope of conflict resolution modules can vary as widely as the scope of policy modules, from application-specific to well-known published conflict resolution module components.

4.2.3. DBMS stored procedures

When data are accessed, the MoP application component needs to retrieve the policy modules associated with the data. Two DBMS stored procedures provide this capability. One receives a SQL request and returns identifiers associated with relevant policies, the other receives a policy module identifier and returns the specified policy module.

MoP therefore requires three or more database queries instead of one for each data request: access the data, request relevant policy module identifiers, and request each needed policy module.

The MoP application component makes all these requests within the same transaction, thereby eliminating potential synchronization difficulties.

The separation of function not only supports flexibility but also decreases performance overhead by allowing the application to make only those requests it actually needs and to make them in any order. For example, for READ requests the policy may be run before the data are retrieved, because the result of the policy may make retrieving the data unnecessary, or the application may first retrieve data and review it to determine which of the associated policies are relevant to its intended use of the data before requesting the policy modules.

Separating the request for policy module identifiers from the request for specific policy modules allows the application to cache policy modules and to request only those policy modules it actually needs, a potentially significant performance enhancement.

4.2.4. Application framework component (MoP)

The MoP application framework component encapsulates the MoP implementation details that are not application dependent. The MoP component exposes methods that support accessing data, identifying relevant policy modules, retrieving relevant policy modules, and running selected policy types.

As of this writing, the application is responsible for setting up pointers to permanent objects (in the current version, the permanent objects are caches and connections to databases), providing an object that actualizes parameters, and calling the MoP retrieve time and MoP apply time methods.

4.3. MoP shared vocabularies

MoP shared vocabularies are the heart of our solution for sharing policy among developers responsible for different application tiers while minimizing the knowledge they must share with

each other. Encapsulating policy rules into components allows us to reduce the semantics that must be shared from understanding policy logic, which requires a "magic language" and is very difficult, to understanding a small set of shared vocabulary items, which is a relatively easy and familiar technology. Each vocabulary item has an identifier, a syntax, and a semantic meaning.

MoP uses three shared vocabularies: policy module types, output parameters, and input parameters.

In the *policy module types* vocabulary, each term specifies what the policy module does, such as add a copyright notice or determine whether access is granted. The policy module type vocabulary is the most pervasive of the MoP vocabularies. Module type terms must be understood by all participants in policy management except for DBMS developers, including policy makers and policy module developers, conflict resolution module developers, data store administrators, and application developers. The policy module type is important to DBAs determining the applicability of policy modules to data and to application programmers determining the applicability of policy modules to their intended use for the data.

In the *output parameters* vocabulary, each term specifies both syntax and meaning of a parameter returned from a policy module. Output parameters are the same for all policy modules of the same type. The application uses the output parameters to apply the policy. Output parameter terms must be understood by policy makers and policy module developers, conflict resolution module developers, and application developers. The output parameter vocabulary is important because for many policy types, such as access control, the application must be prepared to take action based on returned output parameters.

In the *input parameters* vocabulary, each term specifies an input parameter needed by the policy module. The application provides a method that accepts a list of input parameters and returns a list of matching values. Input parameter terms must be understood by developers of policy modules and by application developers. The input parameter vocabulary is important because two modules with the same function may have different input parameters.

The scope of the shared vocabularies can vary in the same way the scope of a policy module can vary. Vocabulary terms can be shared among developers of a single application, developers within a department or an organization. If MoP becomes widely available, some vocabulary term definitions may be standardized across the industry.

4.4. Allocation of responsibilities

Supporting separation of duty by allocating specific policy management responsibilities to different development groups is MoP's prime benefit. With MoP, each group of developers needs to understand only the subset of policy management that falls properly within the group's purview.

MoP allocates responsibilities to policy makers, database administrators, DBMS developers, and application developers. MoP does not impose any requirements on the client tier.

Policy makers specify the policy rules that are implemented by MoP policy modules. They also have the ultimate responsibility for locating or creating policy modules that implement the rules

and conflict resolution modules that implement the resolution of conflicts among policy rules. Policy makers must understand:

- policy module function type semantics,
- policy module input and output parameter semantics and syntax,
- policy and conflict resolution rules,
- policy module and conflict resolution module interfaces.

DBMS developers create stored procedures that implement the two DBMS functions required by MoP, returning identifiers for the policy modules associated with a data access request and returning a policy module on request. Database administrators must understand:

- policy module function type semantics,
- which policy modules are to be associated with which data.

Database administrators create and install the MoP stored procedures into the DBMS, insert policy modules identified by the policy makers, and associate policy modules with data. Mechanisms for these functions will vary from DBMS to DBMS. This process is out of the scope of MoP. DBMS developers must understand:

- DBMS development environment,
- stored procedure interfaces.

Application developers call the MoP application component, pass it required parameters, and use policy module outputs to apply policy. Application developers must understand:

- policy module function type semantics,
- policy module input and output parameter semantics and syntax,
- MoP application component interfaces.

4.5. The implementation

We are using a prototype implementation of the MoP components to validate our framework design as we develop it. We do not consider any portion of our design complete until it has been included in the prototype.

The current prototype uses Microsoft Visual Basic Enterprise 6.0, the Common Object Module (COM) and Microsoft Access 8.0. Early work has focused on building the MoP application component, using stubs for database support and policy modules, and a demonstration application that exercises each feature of the MoP application component.

Our target databases are Oracle and SQL Server. Access does not provide stored procedures or sophisticated policy management mechanisms, but its functionality is adequate to support work on the MoP application component.

The current demonstration application, showing interior functions of the MoP application component, is shown in Fig. 8.
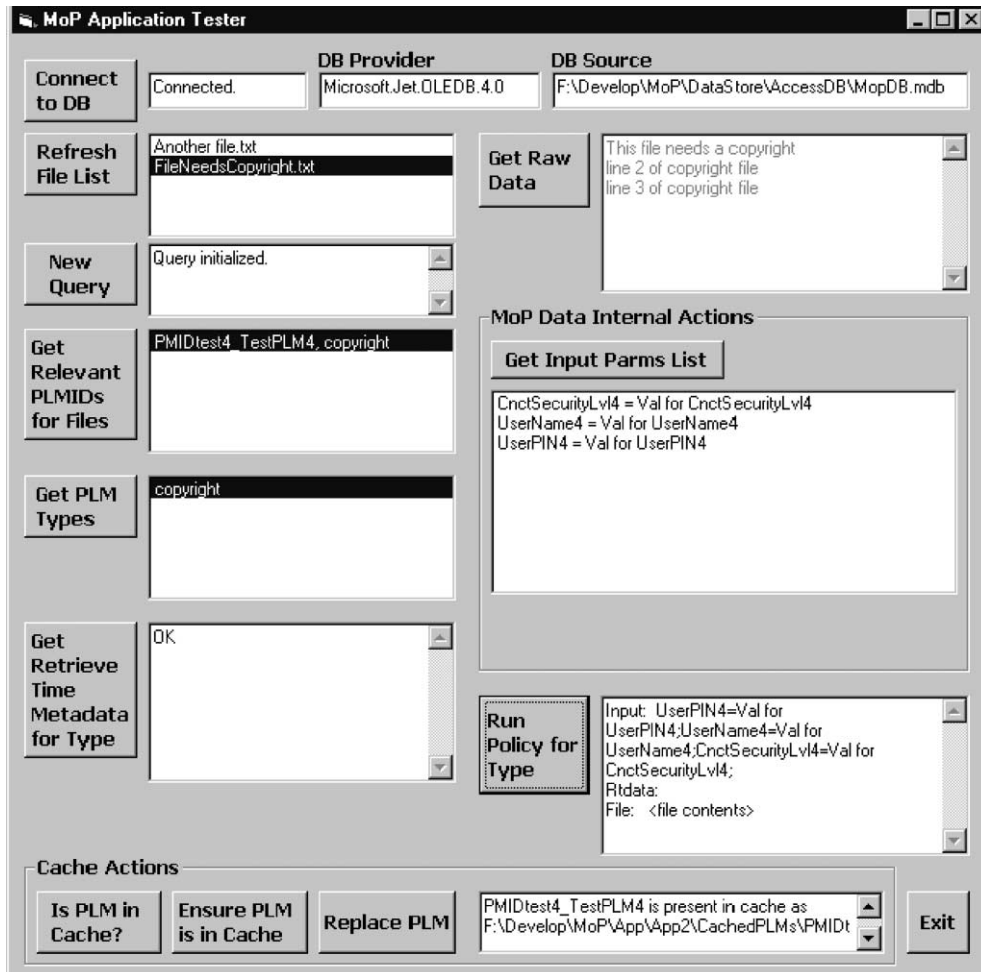
Fig. 8. MoP demonstration application.

5. Conclusions and future work

This paper proposes the use of mobile policy in multi-tier information systems. Specifically, it separates policy administration from policy enforcement. Policy is specified and administered at the element of a distributed system where the data being controlled by policy is defined. That policy is then shared with consumers of the data so that they can enforce the appropriate policy when using the data.

In this paper, mobile policy is proposed as a means for making DBMSs more composable. This is analogous to the capability provided by the X/Open Distributed Transaction Processing (DTP) model [15,16]. X/Open DTP allows DBMSs to participate in transactions managed by an external transaction monitor. It essentially opens up the DBMSs transaction processing protocol to allow two-phase commit across the DBMS and other software components. Mobile policy attempts to

provide the same capability to the access control mechanisms of the DBMS while simultaneously extending it to handle a broader collection of data handling policies beyond access control.

Although we have not completed work on the basic MoP framework, we have identified a number of enhancements that we would like to add once the basic framework is complete: dynamically generated policy modules, dynamic determination of conflict resolution metadata, a policy composition module that manages relationships among different policy modules, and support for associating policy modules with subsets of retrieved data.

Dynamically generated policy modules are interesting because they would eliminate parallel implementations of the same policy. DBMS systems already have a mechanism that associates access control policies with data. We would like to develop a mechanism that extracts the access control information relevant to an SQL query and packages it as a MoP policy module. In addition to convenience value, automatic generation of MoP policy modules potentially could enhance assurance because the information would not have to be associated with the data twice, once as a policy module and once as DBMS access control lists.

Dynamic determination of conflict resolution metadata is interesting because it would simplify the task of policy module developers. As it stands today, MoP requires linked code development in policy modules on one type and their associated conflict resolution modules. We think it would be desirable to provide a cleaner interface so that policy module and conflict resolution module development can be more independent.

Support for associating policy modules with subsets of retrieved data is interesting because it would support applications, such as data warehouses, where a large block of data is retrieved all at once and stored internally in database table format. Later, when the data are to be used, the application extracts subsets of the data for each specific use. MoP as currently designed does not support this kind of application.

Before our framework can be shown to be useful in production environments, a number of issues need to be addressed: performance, multi-platform application support, and assurance.

Performance is an issue, because a prime reason for using multi-tier architectures is to gain enhanced scalability and efficiency. If making policies mobile slows processing down any appreciable amount, any benefits will not be worth the cost.

Multi-platform support is an issue because another prime reason for using multi-tier architectures is to gain application development flexibility. If the MoP application component can be called only by COM applications, and not by EJB or CORBA applications, its usefulness will be limited.

Assurance is an issue because many MoP policies are security policies. A mechanism for implementing security policies that cannot itself be shown to meet enterprise requirements for security will not be very useful.

References

[1] D.L. Black, D.B. Golub, D.P. Julin, R.F. Rashid, R.P. Draves, R.W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, D. Bohman, Microkernel operating system architecture and mach, Journal of Information Processing 14 (1995) 4.

[2] P. Bonati, S. De Capitani di Vimercati, P. Samarati, A Modular Approach to Composing Access Control Policies, in: Proceedings of the ACM Computer and Communications Security Conference, 2000.

[3] B.J. Cox, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley, 1986.

[4] T. Digre, Business object component architecture, IEEE Software 15 (5) (1998).

[5] V. Doshi, A. Fayad, S. Jajodia, R. MacLean, Using attribute certificates with mobile policies in electroniccommerce applications, in: Proceedings of 16th Annual Computer Security Applications Conference, New Orleans, LA, December 2000, pp. 298–307.

[6] A. Fayad, S. Jajodia, D. Faatz, V. Doshi, Going beyond MAC and DAC using mobile policies, in: Proceedings of 16th IFIP International Conference on Information Security, Paris, France, June 2001.

[7] S.E. Minear, Providing Policy Control Over Object Operations in a Mach Based System, Secure Computing Corporation, Roseville, MN, April 1995.

[8] S.E. Minear, Controlling Mach Operations for use in Secure and Safety-Critical Systems, Secure Computing Corporation, Roseville, MN, June 1994.

[9] Object Management Group (OMG), Resource Access Decision (RAD), OMG document corbamed/99-03-02, March 1999.

[10] Object Management Group (OMG), Transaction Service Specification.

[11] B. Plotkin, S. Garone, Apple's WebObjects: Innovative Engineering + Internet Standards = Industrial-Strength Web Application Server, IDC, Framingham, MA.

[12] R. Simon, M.E. Zurko, Separation of duty in role-based environments, in: Proceedings of the 10th Computer Security Foundations Workshop, June 1997.

[13] US Department of Commerce/National Institute for Standards and Technology, Standard Security Label for Information Transfer, FIPS PUB 188, September 1994.

[14] G. Wiederhold, Mediators in the architecture of future information systems, IEEE Computer 25 (3) (1992) 38–49.

[15] X/Open Company Ltd., Distributed Transaction Processing: The TxRPC Specification, X/Open document P305, Reading, UK.

[16] X/Open Company Ltd., Distributed Transaction Processing: The XATMI Specification, X/Open document P305, Reading, UK.

[17] M.E. Zurko, R. Simon, T. Sanfilippo, A user-centered, modular authorization service built on an RBAC foundation, in: Proceedings of IEEE Symposium on Security and Privacy, May 1999.

Susan Chapin is a Lead INFOSEC Engineer at The MITRE Corporation, where she has worked for the Center for Integrated Intelligence Systems since 1992. She has degrees from Harvard University and San Diego State University, and has worked as a software developer and information systems security engineer since 1976. Her e-mail address is schapin@mitre.org.

Don Faatz is a Principal Information System Security Engineer with The MITRE Corporation in McLean Virginia. His work is focused on architectures for secure distributed information systems. He has a BS and ME in Computer Systems Engineering from Rensselaer Polytechnic Institute and is pursuing a Ph.D. in Information Technology at George Mason University.

Sushil Jajodia is Principal Scientist at The MITRE Corporation in McLean, Virginia. He is also the BDM Professor and Chairman of the Department of Information and Software Engineering and Director of Center for Secure Information Systems at the George Mason University, Fairfax, Virginia. He received his Ph.D. from the University of Oregon, Eugene. His research interests include information security, temporal databases, and replicated databases. He has authored four books, edited seventeen books, and published more than 250 technical papers in the refereed journals and conference proceedings. He received the 1996 Kristian Beckman award from IFIP TC 11 for his contributions to the discipline of Information Security, and the 2000 Outstanding Research Faculty Award from GMUs School of Information Technology and Engineering. Dr. Jajodia has served in different capacities for various journals and conferences. He is the founding editor-in-chief of the Journal of Computer Security, and serves on the editorial boards of ACM Transactions on Information and Systems Security and International Journal of Cooperative Information Systems. He is the consulting editor of the Kluwer International Series on Advances in Information Security. The URL for his web page is http://isse.gmu.edu/~csis/faculty/jajodia.html.

Amgad Fayad leads the advanced security research section at the MITRE Corporation. His interests include security service application programming interfaces (API), suspicious user confinement, access and release control, and penetration testing methodologies. He recently taught courses at George Mason University on C++ programming and discrete mathematics. He holds an M.S. degree in computer sciences from Purdue University, West Lafayette, Indiana.