Using Views for Customizing Reusable Components in Component-Based Frameworks

Anca-Andreea Ivan
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
ivan@cs.nyu.edu

Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University, New York, NY 10012
vijayk@cs.nyu.edu

Abstract

Increasingly, scalable distributed applications are being constructed by integrating reusable components spanning multiple administrative domains. Dynamic composition and deployment of such applications enables flexible QoS-aware adaptation to changing client and network characteristics. However, dynamic deployment across multiple administrative domains needs to perform cross-domain authentication and authorization, and satisfy various network and application-level constraints that may only be expressed in terms meaningful within a particular domain.

Our solution to these problems, developed as part of the Partitionable Services Framework, integrates a decentralized trust management and access control system (dRBAC) with a programming and run-time abstraction (object views). dRBAC encodes statements within and across domains using cryptographically signed credentials, providing a unifying and powerful mechanism for cross-domain authorization and expression of network and application constraints. Views define multiple implementations of a reusable component, thus enriching the set of components available for dynamic deployment and enabling fine-grained, customizable access control. We describe the run-time support for views, which consists of a view generator (VIG) and a host-level communication resource (Switchboard) for creating secure channels between pairs of components. We present a simple mail application to illustrate how dRBAC, views, and Switchboard can be used to customize reusable components and securely deploy them in heterogeneous environments.

1. Introduction

Increasingly, scalable distributed applications are being constructed by integrating reusable component services spanning multiple administrative domains. Grid frameworks like Globus [9], or component frameworks like DCE [27], DCOM [30], and CORBA [24] provide infrastructural support to ease construction of such component-based applications, allowing services to register with a common substrate that provides basic services — discovery, resource management, security.

Although most such frameworks rely on static component linkages, a growing number of systems (Active Frames [23], Eager Handlers [33], Ninja [28], Active Streams [4], CANS [12], Partitionable Services [16], Conductor [21] and a recent version of Globus [9]) advocate a more dynamic model, where components are combined at run-time, based on the current state of the environment and QoS requirements of the clients. This dynamic model enables applications to flexibly and dynamically adapt to changes in resource availability and client requests. For example, low bandwidth can be masked by deploying a cache component close to the client. Similarly, security-aware applications can deploy an encryptor/decryptor pair to protect sensitive data crossing insecure links.

However, dynamic component-based frameworks must overcome several challenges before these benefits become possible. On top of the fact that component deployments need to span multiple administrative domains, necessitating cross-domain authentication and authorization among dynamically created principals (a problem that grid applications also encounter and address), dynamic frameworks must address two additional issues. First, they need to work with a set of *reusable components*, selecting among and specializing these components as appropriate for the environment and client QoS requirements. Second, the component selection and specialization process must be cognizant of various application and environment constraints *which may only be expressed in terms meaningful within a particular domain*. For example, nodes in the network may be required to have certain software packages, or component code may need to be signed by somebody the node owner trusts. We believe that frameworks that provide flexibility in component selection and expressiveness for cross-domain constraints are likely to see wider usage than others.

This paper describes our solutions to these issues, developed in the context of the Partitionable Services Framework (PSF) [16]. PSF is a dynamic component-based framework which allows applications to flexibly adapt to heterogeneous environments by assembling and deploying their constituent components as required by the network characteristics and the client's QoS requirements.

In this paper, we integrate a decentralized trust management and access control system called *dRBAC* with a programming and run-time abstraction called *object views* to refine assumptions of PSF-like component-based framework: (1) the availability of a reusable set of customizable components, without detailing how these arise; (2) application and network-level constraints expressed using the same terms; and (3) a relatively simple authentication and access control model for applications based on client and node credentials. dRBAC encodes statements within and across domains using cryptographically signed credentials, enabling expression and resolution of diverse application and environment constraints. Views define multiple implementations of a reusable component, enriching the set of components available for dynamic deployment and thereby increasing the likelihood of a successful deployment satisfying constraints. In addition, dRBAC and views together provide a unifying mechanism for cross-domain authentication and authorization, supporting single sign-on and fine-grained access control. We also describe the run-time support required for deploying component views, which consists of a view generator called VIG and a host-level communication resource called Switchboard for creating secure channels between pairs of components. To illustrate how dRBAC, views, and Switchboard work together to facilitate customization and secure deployment of reusable components in a heterogeneous environment, the paper presents a case study of a simple component-based mail service.

The rest of the paper is structured as follows. In Section 2., we review PSF, the context for this work, and a security-aware component-based mail application, originally introduced in [16]. Sections 3. and 4. describe the dRBAC trust management system and the object views abstraction, and their use in PSF. We discuss related work in Section 5. and conclude in Section 6..

2. Background

2.1. PSF - Partitionable Services Framework

In order to allow applications to flexibly adapt to heterogeneous environments, PSF relies on four elements: (1) a *declarative specification* of application and environment characteristics, (2) a *monitoring* module, (3) a *planning* module, and (4) a *deployment* infrastructure.

Similar to the Corba Component Model [25], components are modeled as entities that *implement* and *require* typed interfaces, each of which is associated with a set of properties. The environment itself is modeled in terms of nodes and links that possess their own set of properties, and are additionally capable of influencing the implemented interface properties of deployed components. Such modeling of application and network behaviors permits the use of type compatibility to define what constitutes a valid application configuration: two components can be linked to each other if one implements interfaces the other requires. The current PSF implementaion works with Java-based components. However, PSF can be easily extended to other models for expressing component functionality (e.g. WSDL [31] in web services) and connectivity (e.g. matching web services at the method level).

The *planning* module is responsible for selecting amongst valid application configurations the satisfy the level of service requested for the deployment while factoring in application and network-level constraints, updates to which are tracked by the *monitoring* module. Our current planner, Sekitei [18], combines regression and progression techniques from classical AI planning to cope with general constraints and network scale concerns. The output of the planner is a sequence of component deployments, realized using the *deployment infrastructure* which securely instantiates, links, and executes the components on the given nodes.

2.2. Component-based mail application

We will use a security-aware mail application throughout the paper to illustrate how dRBAC and views work in PSF. The main components of this application are: *mail clients* with different capabilities, a *mail server* that manages the mail accounts for all users, *view mail server* components that can be replicated as a cache close to the client, and *encryption/decryption* components that ensure the privacy of all messages sent over insecure links.

The mail application offers different levels of QoS, where each level is defined by the number of processed requests and the message privacy. PSF ensures that clients receive the required level of service by assembling and deploying components, as described in Section 2.1.. For example, PSF adapts to low available bandwidth by placing a *view mail server* close to the client and to insecure links by placing <*encryptor*/*decryptor*> pairs.

Based on the mail application, we build the following scenario: the mail service is used by a company (*Comp*) to provide e-mail facilities to its members, across three sites: the main office in New York, a branch office in San Diego, and a partner organization (*Inc*) in Seattle. The three sites compare to LANs, with fast and reliable links, connected to each other by high latency and insecure WAN links. Sections 3. and 4. use this scenario to describe how dRBAC and views work in PSF.

3. dRBAC: Decentralized Role-Based Access Control

3.1. dRBAC features and implementation

dRBAC [5] is a PKI-based trust management and role-based access control system originally developed for expressing and enforcing security policies in coalition environments spanning multiple administrative domains. Such environments are characterized by partial trust and the absence of central policy roots. dRBAC credentials, called *delegations*, express the mapping of an equivalence class of access rights in one trust domain to members of another equivalence class, possibly in another trust domain. Each of these equivalence classes is represented by a dRBAC *role*. These delegations potentially include attenuation of valued attributes. A summary of relevant features of dRBAC follows; a more complete description appears in [5].

Each dRBAC delegation is cryptographically signed by its issuer. Additional credentials may be required as evidence of the issuer's authorization to administer the rights proved by the delegation. As with other role-based access control systems, dRBAC delegations may be transitively chained to form proof graphs indirectly authorizing a required class of access rights. A dRBAC credential can be tagged with expiration dates and also may additionally require online validation monitoring from an authorized "home" which is aware of any revocation of the delegation. Similar to other distributed trust management engines (SPKI [7], KeyNote [2], PolicyMaker [3]), dRBAC supports third party delegations and linked namespaces.

Table 1 presents the three types of dRBAC credentials: self-certifying, third-party, and assignment delegations. The self-certifying and third-party delegations allow an Issuer entity to give the permissions associated with an Entity.Role role to a different entity or role (Subject). The difference between them is based on whether the owner of that role is also the Issuer. An Issuer entity uses the assignment delegation to give the *right of assignment* for Entity.Role to another entity (Subject) located outside the Issuer's space. The assignment delegations permit the usage of private roles outside the defining domain. The ($'$) mark indicates that the Subject is allowed to assign Entity.Role to other Subjects.

Using dRBAC, a trust-sensitive component C can determine if a set of dRBAC credentials X gives some subject S the set of access rights represented by a role R continuously over some duration. To do this, C presents the public identity of S, a set of required access rights R, and the credentials

Table 1. dRBAC delegation types.

Self-certifying	[Subject $\rightarrow$ Issuer.Role] Issuer with $Attr_1$=Val_1, $Attr_2$=Val_2, ...
Third-party	[Subject $\rightarrow$ Entity.Role] Issuer with $Attr_1$=Val_1, $Attr_2$=Val_2, ...
Assignment	[Subject $\rightarrow$ Entity.Role '] Issuer with $Attr_1$=Val_1, $Attr_2$=Val_2, ...

X to a dRBAC implementation. The dRBAC module first authenticates the signatures and establishes validity monitors for all the credentials in X. Authorization is granted if the dRBAC module can construct a graph (proof) from valid and authenticated credentials in X that "proves" that S possesses the rights required by R.

dRBAC credentials are stored in a distributed repository. To assist in collecting dRBAC credentials that authorize a particular role, dRBAC contains a mechanism that relies on *discovery tags* associated with credential subjects and objects. These tags identify an entity as "searchable from subject" or "searchable from object", permitting queries about credentials involving the entity to be directed as appropriate to its home node.

3.2. Use of dRBAC in PSF

dRBAC is used in two ways in PSF. Its first use is conventional, for authenticating and authorizing various entities in the framework—clients, components, and network resources—even when these entities span multiple administrative domains. The second use is somewhat novel, for translating between application and network-level constraints each of which are expressed in terms meaningful only within their respective domains. It is this latter use that motivated the use of a general trust management system like dRBAC as opposed to existing grid security architectures such as GSI [10] or CAS [19].

Cross-domain authentication and authorization The security requirements of PSF are described as follows. Clients requesting access to an interface must first be authenticated and then authorized to receive an appropriate level of service. In particular, the planing module takes into consideration the client credentials, the component credentials, and network resource credentials to generate a deployment that achieves the desired level of service and is realizable. The latter entails component and network resource authorization: a node is authorized to host a component, and a component is authorized to execute on a node. Additionally, deployed components may make their own requests for their required interfaces, which triggers this process recursively.

The challenge in achieving the above results from the fact that component deployments span multiple administrative domains, and components can be accessed by anonymous clients as well as deployed on nodes that the component developer is not aware of a priori. dRBAC provides mechanisms by which each administrative domain can issue independent credentials to its clients, components, and network resources, and yet these credentials can be combined to permit cross-domain authorization decisions. The latter is enabled using dRBAC delegations, which provides a mechanism for mapping roles in other domains to roles in the current one. This allows domains/resource owners to set their own security policy, independent of who is likely to access them. Clients belonging to other domains are authorized for a service as long as they present credentials that prove their possession of a role local to the service's domain. Instantiated components receive their own set of credentials permitting use of similar mechanisms for servicing their requests.

The trust management solution to cross-domain authentication and authorization generalizes the approach adopted in Globus-like systems [10, 19], which rely on the translation between a system-wide "grid credential" (virtual organization-level credential in CAS) and local accounts to authorize and enforce security policy for client requests. Our approach offers advantages of scalability (multiple policy roots are permitted), easier configuration (local policy need not include translation between grid and local credentials, which is automatically inferred), and finer-grained control (the rights afforded a request can be modulated to the credentials associated with it as opposed to the account these translate to). We defer a detailed discussion of the latter advantage to Section 4..

Expressing application and network constraints The second use of dRBAC is motivated by the observation that dRBAC credentials are just *statements* about entities within and across administrative domains, whose authenticity can be cryptographically verified. Thus, a dRBAC credential that grants the permissions associated with an Object role to a Subject role can also be interpreted as the statement that "it is true that Subject **is an** Object".

This interpretation allows the use of dRBAC credentials to encode various application and network-level properties and constraints on these properties, which drive the deployment process. Properties associated with application components and network resources are encoded using dRBAC credentials. Constraints are specified in terms of dRBAC system queries: "is X a Y?" (more precisely, the constraint is that X must possess role Y). Note that by design, dRBAC permits properties and constraints to be defined in terms of local names, relying on *role mapping* delegations to define translations across domains.

3.3. Example: dRBAC use in mail application

In this section, we will use the mail application to explain how PSF uses dRBAC to:

1. Authorize clients before accessing a service.
2. Authorize nodes before choosing them for component deployment. This step also includes mapping the network properties onto application specific properties.
3. Provide the necessary credentials, such that nodes can authorize components before executing them.

We start with the scenario described in Section 2.2., and assume that each site (New York, San Diego, Seattle) is running PSF. Beside the main modules – registrar, monitor, planner, deployer – the framework has a security module (*Guard*) that manages the site security by generating certificates, defining roles, creating access control lists, authenticating, and authorizing. We assume that (i) *NY-Guard* is responsible for the correct use of the mail application and all clients located in New York, (ii) *SD-Guard* manages the San Diego clients even though they should be considered as belonging to the same logical domain as the New York clients, and (iii) *SE-Guard* manages all clients from Seattle. Table 2 presents an example of dRBAC credentials that ensures correct authorization of clients, nodes, and components.

Client authorization. The goal is to allow clients to use credentials defined by their local *Guard* for both local and cross-domain authorization. PSF achieves this goal by using dRBAC to find a mapping from a role to another role, even if the roles are created by different domais. For example, Bob works in San Diego and holds credential (11) created by *SD-Guard*, which associates the role `Comp.SD.Member` with Bob's identity. If Bob wants to access the mail service, he should be authorized as one of the roles defined by *NY-Guard* (e.g. `Comp.NY.Member` or `Comp.NY.Partner`). In this case, dRBAC proves that Bob is `Comp.NY.Member` by presenting credentials (2) and (11).

Node authorization. The node authorization process consists of two steps: (i) the actual authorization of the node, and (ii) the transformation of the node properties into properties meaningful to the application. The second step decides whether the node is useful during adaptation or not. The first part of the node authorization can be easily achieved in a similar way to the client authorization.

The interesting question is how to transform the node properties generated by one administrative domain into properties that are meaningful to the application. The challenge arises from the fact that the component developer has no a priori knowledge of the node(s) where the component may be deployed. For example, the policy of the mail application is expressed only in terms meaningful to the programmer's domain, and states that Dell machines installed

Table 2. The roles and certificates generated by the Guard modules

New York		
User Auth.	(1)	[Alice → Comp.NY.Member] Comp.NY
	(2)	[Comp.SD.Member → Comp.NY.Member] Comp.NY
	(3)	[Comp.SD → Comp.NY.Partner '] Comp.NY
Node Auth.	(4)	[Dell.Linux → Mail.Node with Secure={true,false} Trust=(0,10)] Mail
	(5)	[Dell.SuSe → Mail.Node with Secure={true,false} Trust=(0,7)] Mail
	(6)	[IBM.Windows → Mail.Node with Secure={false} Trust=(0,1)] Mail
	(7)	[Comp.NY.PC → Dell.Linux] Dell
Component Auth.	(8)	[Mail.MailClient → Comp.NY.Executable with CPU=100] Comp.NY
	(9)	[Mail.Encryptor → Comp.NY.Executable with CPU=100] Comp.NY
	(10)	[Mail.Decryptor → Comp.NY.Executable with CPU=100] Comp.NY
San Diego		
User Auth.	(11)	[Bob → Comp.SD.Member] Comp.SD
	(12)	[Inc.SE.Member → Comp.NY.Partner] Comp.SD
Node Auth.	(13)	[Comp.SD.PC → Dell.SuSe] Dell
Component Auth.	(14)	[Comp.NY.Executable → Comp.SD.Executable with CPU=80] Comp.SD
Seattle		
User Auth.	(15)	[Charlie → Inc.SE.Member] Inc.SE
NodeAuth.	(16)	[Inc.SE.PC → IBM.Windows] IBM
Component Auth.	(17)	[Comp.NY.Executable → Inc.SE.Executable with CPU=40] Inc.SE

with Linux are secure and have a trust level between 0 and 10 (credential 4), Dell machines installed with Suse are secure with a trust level between 0 and 7 (credential 5), and IBM machines installed with Windows are not secure and have a trust level between 0 and 1 (credential 6). Similarly, all node credentials are defined as properties local to their domains. For example, all machines from the San Diego site have a credential (13) generated by Dell, stating that they are running SuSe. PSF decides to deploy a mail component on a node only if dRBAC finds a possible chain of credentials that maps a node credential to a policy credential. In our example, the machines from San Diego can be mapped from credential (13) to credential (5). A similar process can be used to map link properties onto application properties.

Component authorization. The second part of the mutual authorization between a node and a component requires that a node accept the component before allowing it to run. In this case, nodes should be able to authorize components, even if the components might belong to a different domain. In our example, *NY-Guard* creates local credentials for three components that need to be deployed in different domains: `MailClient` in New York, `Encryptor` in San Diego, and `Decryptor` in Seattle. *SD-Guard* defines a `Comp.SD.Executable` role to specify that all components having this role will be allowed to run on the San Diego machines with a limit of 80% in CPU consumption. In a similar way, *SE-Guard* defines a `Inc.SE.Executable` role that restricts the CPU consumption to 40%. Then, both *SD-Guard* and *SE-Guard* associate these roles to the `Comp.NY.Executable` role. This allows *NY-Guard* to generate only local credentials (`Comp.NY.Executable`) for the `MailClient`, `Encryptor`, and `Decryptor` components. Whenever a component is deployed on a node, it presents a chain of credentials. The component is accepted only if the node recognizes the chain of credentials as valid.

4. Views: Customizing Reusable Components

Object views [22] were originally proposed in the context of parallel programming languages supporting a shared object space. In that context, views allowed reduction of coherence traffic by defining a coherence granularity smaller than the object and encapsulating application-specific protocols. PSF employs the same underlying mechanism but for very different goals: to increase flexibility of distributed component deployment in the presence of application- and network-level constraints, and to enable fine-grained access control.

4.1. The views abstraction

Views provide a mechanism by which to define multiple physical realizations of the same logical component. An object is a *view* of another object, called *original object*, if it (1) implements a subset of the functionality of the original object; or (2) works with a subset of the original object's data. We refer to the former as *object views*, and to the latter as *data views*. Most views in practice are likely to exhibit

Table 3. (a) The original Java object.

```
public interface MessageI {
  public void sendMessage(Message mes )
  public Set receiveMessages()
}
public interface AddressI {
  public String getPhone( String name )
  public String getEmail( String name )
}
public interface NotesI {
  public void addNote( String note )
  public boolean addMeeting( String name )
}
public class MailClient
  implements MessageI,
             AddressI,
             NotesI {
  Account[] accounts;
  public void sendMessage(Message mes ){}
  public Set receiveMessages(){}
  public String getPhone( String name ){
    return findAccount( name ).getPhone();
  }
  public String getEmail( String name ){
    return findAccount( name ).getEmail();
  }
  public void addNote( String note ){}
  public boolean addMeeting( String name ){}

  private Account findAccount(String name){
    return accounts.get(name);
  }
}
```

(b) The XML rules to define a view.

```
<View name = ViewMailClient_Partner >
  <Represents name = MailClient >

  <Restricts>
    <Interface name = MessageI  type = local >
    <Interface name = NotesI  type = rmi >
    <Interface name = AddressI  type = switchboard >

  <Adds_Fields>
    <Field name = accountCopy  type = Account>

  <Adds_Methods>
    <MSign> ViewMailClient_Partner(...)
    <MBody> /** constructor body **/

    <MSign> void mergeImageIntoView(byte[])
    <MBody> /** code to merge image into the view **/

    <MSign> void mergeImageIntoObj(byte[])
    <MBody> /** code to merge image into object **/

    <MSign> byte[] extractImageFromView()
    <MBody> /** code to extract image from view **/

    <MSign> byte[] extractImageFromObj()
    <MBody> /** code to extract image from object **/

  <Customizes_Methods>
    <MSign> String addMeeting( String name )
    <MBody> /** new code for method**/
```

characteristics of both object and data views: we focus on such hybrid views in the rest of the paper.

In the context of PSF, views define (a family of) auxillary components that embody different ways of realizing the component functionality. In general, the functionality of the original object can either be completely replicated in the auxillary component, be completely present in the original object (with the auxillary component just serving as a gateway to this functionality), or be somewhere between these two extremes. To assist with view construction, Table 3(b) shows a simple schema that can be used as a guideline, using as example a component from our mail application given in Table 3(a). The `ViewMailClient_Partner` is a restricted version of the `MailClient` component, able to send/receive messages, add notes into a remote diary, and query the address book in a secure fashion. Such a component is useful if clients use untrusted machines (e.g. airport terminal) to check e-mail.

A minimal view is fully described by a name (`ViewMailClient_Partner`), and a represented object (`MailClient`). The minimal view can be enriched by providing a list of implemented interfaces (`MessageI`, `AddressI`, `NotesI`), defining new methods and fields, and copying or customizing existing methods. For each interface, the view description can specify a type (*local*, *rmi*, or *switch*) that indicates how the interface is available to clients. The methods defined by a *local* interface should be available only to clients running in the same JVM as the view. Interfaces can be also required to be only available on the original object. Access to such interfaces is permitted either via RMI (*rmi*) or a secure communication channel called Switchboard (*switch*), as described below. Beside general methods, a view definition must contain descriptions of several special methods: at least one `constructor` declaration, and complete implementations for cache coherence-specific methods. The cache coherence methods describe how the state of the view can merged/extracted into/from the view/object [22].

4.2. Use of views in PSF

Views offer two advantages in the context of the partitionable services framework: (1) they improve the likelihood of successful component deployment in constrained environments; and (2) they provide a finer granularity at which to authorize and enforce access control decisions.

Increased flexibility in component deployment Dynamic component-based frameworks work with a set of reusable components, selecting among and customizing these components as appropriate for the environment and client QoS requirements. Thus, whether or not the planning module is in fact able to come up with a deployment schedule is dependent on the set of available component types.

Views provide a convenient mechanism for enriching the set of components, without requiring onerous application developer input. By merely distributing component functionality between the original and auxillary objects, views increase the likelihood of the planner finding a component deployment in constrained environments. Additional flexibility arises from allowing view properties to be specified at creation time.

Fine-grained, single sign-on authorization By definition, views implement a subset of the functionality of the original object. Thus, restricting access at the level of methods or interfaces is easily achievable by defining appropriate views. Also, views can be customized to have different implementations depending on their intended users, say by selecting appropriate property values. Access control lists can be established, per component, which specify the level of service (the view) associated with a given dRBAC role. As described earlier, such policy can be established using only roles within the local namespace: cross-domain requests are first translated by dRBAC into local roles before any access control decisions are made.

Table 4 depicts the description of some access control rules created for the mail application scenario. All members of the company (`Comp.NY.Member`) are allowed to send/receive messages, access the phone and email directories, and add notes and meetings to their calendar (`ViewMailClient_Member`). The partners (`Comp.NY.Partner`) can execute the same operations, with the exception that the functionality for setting up a meeting is reduced to only requesting the right to set up a meeting (`ViewMailClient_Partner`). All other clients have only the right to browse the email directory (`ViewMailClient_Anonymous`).

Enforcement of these access control decisions comes naturally because views contain only the subset of object state required for their local methods, and must interact with the original object to realize the rest of their functionality (if any). Views permit single sign-on usage, because authentication and authorization decisions can be completed when the view is first instantiated. After that clients are free to access the view they receive, without additional access control. Moreover, by using a secure communication channel between the view and the original object, as described below, requests that are deferred to the original object can also proceed without requiring additional checks.

Table 4. Rules defined to restrict the client's access to the service. These rules are also used for automatic view creation.

Role	View name
`Comp.NY.Member`	`ViewMailClient_Member`
`Comp.NY.Partner`	`ViewMailClient_Partner`
others	`ViewMailClient_Anonymous`

4.3. Run-time support for view deployment

To understand what run-time support is required for view deployment, consider the sequence of actions that take place in PSF in response to a client request for a service interface. This request is passed on to the planning module, along with any client credentials. The planning module is aware of both the component and view specifications, as well as the current state of network resources. The client credentials serve to identify the subset of components that can be used for deployment (based on access control decisions). Once the planning module finds a valid plan that satisfies the client's QoS requirements, the run-time system is responsible for instantiating, downloading, and securely connecting the views.

View instantiation The generation of the code for a view is deferred to the time this view is first deployed. This ensures that despite their flexibility, views incur management costs proportional to their utility.

The view generation is handled by a tool called **VIG** (View Generator), which takes the class file of the represented object and an XML definition of the view and produces a new classfile corresponding to the view. Table 5 illustrates the view generation process by presenting the Java code for the `ViewMailClient_Partner` view, as defined in Table 3.

VIG uses the API's provided by the Javassist [29] toolkit to manipulate Java objects at bytecode level. The view generation process consists of two steps: (i) reading the XML description and the represented object, and (ii) modifying existing method/interfaces and adding new methods to the view as specified by the XML rules. If VIG is unable to generate correct bytecode (e.g. a new method uses a variable that is not defined in the original object or the method), it triggers an error that indicates how the XML rules can be rectified. Therefore, VIG can be used to both generate views at runtime and guide the programmer's effort to write correct XML files.

VIG decides how to generate the actual view bytecode by processing, in order, (1) interfaces, (2) methods, and (3) fields.

(1) *Local* interfaces do not require any processing, and can be copied as is. However, *rmi* and *switchboard* interfaces need to extend `java.rmi.Remote`, respectively `java.io.Serializable`. For all methods defined by interfaces, VIG processes the actual method implementations as described below.

(2) Methods defined as *local* or by *local* interfaces do not change, and can be copied from the represented object into the view. The methods defined by *rmi* or *switchboard* interfaces are transformed into simple RMI, respectively Switchboard calls against the original object. The main problem when copying existing methods from the represented class into the view is to find the correct implementation. This problem arises when there is an inheritance chain from the represented object to a unique super class. Javassist provides the necessary mechanisms to solve this problem by following the inheritance chain and generating views for every class in the chain such that the "extends" relationships between views is similar to the "extends" relationships between the represented classes. Once VIG finds the right method implementation to copy, VIG parses the method code and copies the declarations of all used class fields.

In order to add a new method or customize an existing method, VIG extracts the method signature and body from the XML description of the view. The actual addition or customization is simplified by the fact that Javassist allows the insertion of pure Java code into the view bytecode. Beside general methods, the XML definition needs to provide the descriptions for at least one view constructor and several cache coherence-specific methods that *extract* the view state and *merge* updates into it. Ideally, the code for the cache coherence methods should be generated by VIG. Currently, the method descriptions are provided by the programmer. Our goal is to supply default handlers in an automatic fashion, which can be overridden as necessary. Another requirement for the cache coherence protocol is that all methods should work with the most current image. VIG ensures it by placing `acquireImage` and `releaseImage` method calls at the beginning and the end of every method implemented by the view.

(2) In general, fields are added either because they are used by a method, or because they are declared by the XML description of the view.

Secure channels between components Once the views are generated, the deployment infrastructure issues to the generated view its own set of credentials, downloads them onto their target nodes, and connects them to other components using secure channels. These channels ensure that component interactions possess the desired security properties, and avoid the need for additional access checks after the channel has been established.

These channels are built on top of a novel communication abstraction called Switchboard, which permits the establishment of secure, authenticated, and *continuously* authorized and monitored connections between a pair of components. The latter property distinguishes Switchboard from abstractions like SSL/TLS [11].

Prior to forming a Switchboard connection, the components at either end provide their *authorization suites*—PKI identities (including private keys for authentication), dRBAC credentials to be supplied to the partner, and `Authorizer` objects for evaluating the partner's credentials. Authorizers generate `AuthorizationMonitors`, which inform either partner when the trust relationship changes. When Switchboard connections span multiple hosts, a cipher is established using a key-exchange protocol, and connectivity is monitored using replay-resistant heartbeats that indicate liveness and round-trip latency. Switchboard connections provide a two-way procedure-call (RPC) interface appearing as a custom socket on top of which Java RMI requests can be routed. A previous version of SwitchboardStream that provides secure and monitored transport is described in [6].

The continuous monitoring property of Switchboard connections is crucial for supporting single sign-on in dynamic environments, where client and/or network credentials can change in the middle of long-lived component interactions. Such a change in credentials invalidates the corresponding dRBAC proofs, and results in notification to the `AuthorizationMonitors` at either end of the connection. These monitors can then take appropriate action, including requiring a component to revalidate itself prior to approving future requests.

5. Related Work

The work described in this paper is related to previous efforts that have looked at cross-domain authorization, modeling application and network resource properties to permit their use by automated planning modules, and support for fine-grained access control.

Cross-domain authorization. Component-based frameworks target application deployment in heterogenous environments spanning multiple administrative domains, and thus raise new security issues. Several systems (DCE [27], DCOM [30], CORBA [24], Globus [9] to cite some examples) aim to solve the cross-domain authentication and authorization problems that result in such systems.

Table 5. View source code.

```
public interface MessageI {
  public void sendMessage(Message mes )
  public Set receiveMessages()
}
public interface AddressI extends Serializable {
  public String getPhone( String name )
  public String getEmail( String name )
}
public interface NotesI extends Remote {
  public void addNote( String note ) throws RemoteException
  public boolean addMeeting( String name ) throws RemoteException
}

public class ViewMailClient_Partner implements MessageI, AddressI, NotesI {

  Account[] accounts;
   public Account accountCopy;
   CacheManager cacheManager;
   NotesI notesI_rmi;
   AddressI addrI_switch;

   public ViewMailClient_Partner ( String[] args ) {

     if (System.getSecurityManager() == null)
       System.setSecurityManager(new RMISecurityManager());

     /** rmi code **/
     notesI_rmi = (NotesI) Naming.lookup(...);

     /** switchboard code **/
     addrI_switch = (AddressI) Switchboard.lookup(...);

     /** initialize cache manager **/
     Properties properties = new Properties( Pname[], Pvalue[]);
     cacheManager = new CacheManager( properties, name);

     /** user supplied code **/
     ...
   }
  public void sendMessage(Message mes)  { /** the original code **/ }
  public Set receiveMessages()          { /** the original code **/ }
  public String getPhone(String name)   { return addrI_switch.getPhone(); }
  public String getEmail(String name)   { return addrI_switch.getEmail(); }
  public void addNote(String note)      { notesI_rmi.addNote(); }
  public boolean addMeeting(String name){ /** user supplied code **/ }

   private void mergeImageIntoView( byte[] image )  /** user supplied code **/
   private void mergeImageIntoObj( byte[] image )   /** user supplied code **/
   private byte[] extractImageFromView()            /** user supplied code **/
   private byte[] extractImageFromObj()             /** user supplied code **/
}
```

DCE [27] provides authentication and authorization based on private-key cryptography with a trusted third party. CORBA [24] and the web services infrastructure [15] provide a general interface for authentication and authorization, leaving it up to application programmers on how exactly they choose to implement it. SESAME [17] authenticates users and provides them with an authorization credential (Privilege Attribute Certificate) that can be used for all authorization decisions. Legion system [14] controls heterogeneous, independent, and distributed resources presenting the user with the image of a single, coherent environment. From a security point of view, all resources are considered to be objects residing in a single shared namespace, and are uniquely identified by a Legion Object Identifier (LOID) [8] that contains a public key. Users are authenticated using shortlived Legion credentials [32] generated the first time the user logs on into the system.

Globus General Purpose Architecture for Reservation and Allocation (GARA) [26] system relies on the Globus Grid Security Infrastructure (GSI) [9] to handle all its authentication and authorization problems. GSI assumes the existence of a Public Key Infrastructure (PKI) and a single shared namespace across domains. Recent work has looked into replacing the PK credentials with Kerberos tokens [1]. In GSI, all resource providers (P) have the necessary authentication/authorization information for all possible users (U, thus implying a storage space proportional with $P \times U$. Floww-on work to GSI, CAS (Community Authorization Service) [19] divides the users into communities such that all providers know about communities only. In this way, CAS improves the memory storage to $C \times (P+U)$, where C is the number of communities. dRBAC reduces the memory storage is $P+U+c$, where c represents the number of credentials created to cross domains.

Our dRBAC-based solution differs from the above efforts in not assuming a single policy root (hence namespace) for credentials. This solution offers advantages of scalability (multiple policy roots are permitted), easier configuration (local policy need not include translation between global system-wide and local credentials, which is automatically inferred), and finer-grained control (the rights afforded a request can be modulated to the credentials associated with it as opposed to the local credentials these translate to).

Expressing component and network properties. Most dynamic component-based frameworks ([28, 21, 12, 16]) rely on an application registration step, where complete specifications of the application components are provided to permit automated deployment planning. Our use of dRBAC credentials to model general application and network-level properties and constraints is in marked contrast to other systems [28, 21], which restrict themselves to specifying only a small number of resource consumption properties (e.g., CPU usage, bandwidth consumption) determined a priori.

Granularity of access control. Except systems like Sesame [17] and Adage [34] which only enforce access control at the level of the entire application, most object-based distributed systems permit finer-grained access control.

The Java 2 environment [13] combined with the Java Authentication and Authorization Service (JAAS) [20] uses security managers and policy files to define resources that in principle can support any granularity of access control. Unfortunately, the security manager only checks rights to access JVM resources, like files or sockets. In order to protect other resources, the applications need to implement their own access control mechanisms.

DCE [27] and CORBA [24] enable any level of granularity for access control by letting the applications to define their own notion of resource. DCOM [30] applications can control access to low level obejct only by taking advantage of the API's exposed by the DCOM programatic security. Legion [14] objects must implement a special function, *M*ayI that is called to check credentials every time a user invokes a method on the object.

Our view-based approach enables access control policies to be specified at arbitrarily fine granularity (down to the level of individual methods), and when coupled with the Switchboard secure channel mechanism, additionally provides single sign-on benefits.

6. Summary and Future Work

In this paper, we have described how views and dRBAC can solve the problems of (i) performing cross-domain authentication and authorization, and (ii) satisfying various network and application-level constraints, when properties should only be expressed in terms meaningful within a particular domain. Views define multiple implementations of a reusable component, thus enriching the set of components available for dynamic deployment and providing fine-grained, customizable, and low-level access control to resources. dRBAC is a powerful mechanism for both cross-domain authorization as well as expression of network and application constraints.

One of the main assumptions made in the Partitionable Services framework is that all domains are using dRBAC as their authorization policy implementation. In order to allow each domain to freely choose the policy implementation (e.g. roles, capabilities), the framework should provide a service able to translate between that implementation and dRBAC.

VIG is designed to create views based on a set of simple rules and the original object. Ideally, VIG should automatically generate the entire view code. The current version is able to partially generate the code; the functions to extract (merge) the image of the object (view) need to be specified by the application programmer. In the future, we plan to fully automate the process of creating views based on a few hints from the programmer.

Acknowledgements

This research was sponsored by DARPA agreements F30602-99-1-0157, N66001-00-1-8920, and N66001-01-1-8929; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

References

[1] W. Adamson and O. Kornievskaia. A Practical Distributed Authorization System for GARA. Technical Report 01-14, Center for Information Technology Integration, University of Michigan, 2001.

[2] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust Management for Public-Key Infrastructures. In *Security Protocols International Workshop*, volume 1550, pages 59–63. Springer LNCS, 1998.

[3] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. In *Financial Cryptography*, pages 254–274, 1998.

[4] F. Bustamante and K. Schwan. Active Streams: An Approach to Adaptive Distributed Systems. In *HotOS*, 2001.

[5] E. Freudenthal et al. dRBAC: Distributed Role-based Access Control for Dynamic Coalition Environments. In *ICDCS*, 2001.

[6] E. Freudenthal et al. Switchboard: Secure, Monitored Connections for Client-Server Communication. In *RESH*, 2002.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Spki certificate theory, rfc 2693. In *Network Working Group, The Internet Society*, 1999.

[8] A. Ferrari, F. Knabe, M. Humphrey, S. Chapin, and A. Grimshaw. A Flexible Security System for Metacomputing Environments. In *HPCN*, pages 370–380, 1999.

[9] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *http://www.globus.org/research/papers.html*, 2002.

[10] I. T. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM CCS*, pages 83–92, 1998.

[11] A. Freier, P. Karlton, and P. Kocher. The SSL Protocol, Version 3.0. In *Internet Draft*, 1996.

[12] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS: Composable, Adaptive Network Services Infrastructure. *USITS*, 2001.

[13] L. Gong. Java security: present and near future. *IEEE Micro*, 17(3):14–19, 1997.

[14] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Wide-Area Computing: Resource Sharing on a Large Scale. *Computer*, 32(5):29–37, 1999.

[15] IBM Corporation and Microsoft Corporation. Security in a Web Services World: A Proposed Architecture and Roadmap. *htpp://msdn.microsoft.com/*, 2002.

[16] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogenous Environments. In *HPDC*, 2002.

[17] P. Kaijser, J. Parker, and D. Pinkas. SESAME: The Solution to Security for Open Distributed Systems. In *Computer Communications*, 1994.

[18] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. In *IPDPS*, 2003.

[19] L. Pearlman at el. A Community Authorization Service for Group Collaboration. In *IEEE Workshop on Policies for Distributed Systems and Networks*, 2002.

[20] C. Lai, L. Gong, L. Koved, A. Nadalin, and R. Schemers. User authentication and authorization in the Java platform. In *15th Annual Computer Security Applications Conference*, pages 285–290. IEEE Computer Society Press, 1999.

[21] J. Li, M. Yarvis, and P. Reiher. Securing Distributed Adaptation. In *OpenArch*, 2001.

[22] I. Lipkind, I. Pechtchanski, and V. Karamcheti. Object Views: Language Support for Intelligent Object Caching in Parallel and Distributed Computations. In *OOPSLA*, pages 447 – 460, 1999.

[23] J. Lopez and D. O'Hallaron. Support for Interactive Heavyweight Services. In *HPDC*, 2001.

[24] Object Management Group. CORBA Security Services, Ver. 1.8. *http://www.omg.org/*, 2002.

[25] Object Management Group. CORBA Component Model. *http://www.omg.org/*, 2003.

[26] R. Butler et al. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, 2000.

[27] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE*. O'Reilly & Associates, Inc., 1992.

[28] S. Czerwinski et al. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[29] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of Legacy Java Software. pages 236–255, 2001.

[30] W. Rubin et al. *Understanding DCOM*. Prentice Hall, 1999.

[31] W3C. Web Services Description Language (WSDL) 1.1. *http://www.w3.org/TR/wsdl*, 2003.

[32] W. Wulf, C. Wang, and D. Kienzle. A New Model of Security for Distributed Systems. Technical Report CS-95-34, CS Department, University of Virginia, 1995.

[33] D. Zhou and K. Schwan. Eager Handlers - Communication Optimization in Java-based Distributed Applications with Reconfigurable Fine-grained Code Migration. *3rd Intl. Workshop on Java for Parallel and Distributed Computing*, 2001.

[34] M. E. Zurko, R. Simon, and T. Sanfilippo. A User-Centered, Modular Authorization Service Built on an RBAC Foundation. In *IEEE Symposium on Security and Privacy*, pages 57–71, 1999.