Authorization for Metacomputing Applications

G. Gheorghiu, T. Ryutov and B.C. Neuman
Information Sciences Institute
University of Southern California
4676 Admiralty Way suite 1001
Marina del Rey, CA 90292
grig, tryutov, bcn@isi.edu
(310)822-1511 (voice) (310)823-6714 (fax)

Abstract

One of the most difficult problems to be solved by metacomputing systems is to ensure strong authentication and authorization. The problem is complicated since the hosts involved in a metacomputing environment often span multiple administrative domains, each with its own security policy. This paper presents a distributed authorization model used by our resource allocation system, the Prospero Resource Manager [8]. The main components of our design are Extended Access Control Lists, EACLs, and a General Authorization and Access API, GAA API. EACLs extend conventional ACLs to allow conditional restrictions on access rights. In the case of the Prospero Resource Manager, specific restrictions include limits on the computational resources to be consumed and on the characteristics of the applications to be executed by the system, such as name, version or endorser. The GAA API provides a general framework for applications to access the EACLs. We have built a prototype of the system.

1. Introduction

Metacomputing is sometimes defined as the abstraction of geographically dispersed computing and communication resources (e.g. supercomputers and high-speed networks) into a single metacomputer [2]. Ideally, the user of the system is presented with a consistent and familiar interface that hides the geographic scale, the complexity and the heterogeneity.

A metacomputing system usually crosses administrative domains and involves a very large number of computing resources. Such systems have particularly sensitive requirements for security. This is one of the most difficult requirements to satisfy, due to the large scale and heterogeneity of the resources involved. The problem is complicated by the variety of representations and by the application of access control policies across multiple administrative domains.

This paper describes the authentication and authorization mechanisms and policies used by the Prospero Resource Manager (PRM [8]), a scalable resource allocation system that manages processing resources in metacomputing environments. PRM uses Kerberos [9] to achieve strong authentication and integrates a new distributed authorization model. Because different administrative domains might use different security services for authentication of principals (e.g. DCE, X.509), we designed the system to be extensible, allowing a variety of security services to be used instead of or in addition to Kerberos. The model is based on two ideas:

1. Extended Access Control Lists (EACL): conventional Access Control Lists (ACL) are extended with an optional field added to each ACL entry specifying restrictions on authorized rights. In the case of PRM, the attributes include strength of authentication, limits on the physical resources managed by the system (e.g. CPU load, memory usage) and characteristics of applications that the users are willing to run on their processors (e.g. name, version, endorser).

2. General Authorization and Access API (GAA API): we defined a common API to facilitate authorization decisions for applications. PRM invokes the GAA API functions to determine if a requested operation or set of operations is authorized or if additional checks are necessary.

Ease of use and configurability are important issues to be considered for any resource management system. For this reason, we developed a scalable mechanism based on the Prospero Directory Service to facilitate the management of the extended access control lists.

The paper is organized as follows. Section 2 describes the Prospero Resource Manager. Section 3 presents the motivation for the authorization model applied to metacomputing applications. Section 4 discusses the two components of the distributed authorization model: the EACL framework and the GAA API. Section 5 shows how the model is adapted and integrated within PRM. Section 6 describes the management of the EACL using the Prospero Directory Service. Section 7 discusses related work.

2. The Prospero Resource Manager

The design of the Prospero Resource Manager was guided by the concept of the Virtual System Model, in which resources of interest are readily accessible and those of less interest are hidden from view [8]. PRM applies this concept to the problem of allocating resources in large scale systems by dividing the functions of resource management between three types of managers: the system manager, the job manager and the node manager. Each manager makes scheduling decisions at a different level of abstraction and this separation of management enables PRM to scale as the number of managed resources increases.

Throughout the paper, we will use the term *node* to denote a processing element, be it a processor in a multiprocessor environment or a workstation whose resources are made available for running jobs. A *job* consists of a set of communicating tasks, running on the nodes allocated to the job. A *task* consists of one or more threads of control of an application, together with the address space in which they run.

2.1. The system manager

In PRM, the total collection of processing resources is divided into subsets which correspond usually to administrative domains. Each subset is managed by a *system manager* which is responsible for allocating its resources to jobs as needed. The system managers themselves can be organized in a hierarchical manner in order to avoid bottlenecks and ensure scalability.

The system manager maintains information about the characteristics of each resource it manages, together with the mapping from resources to jobs. The system manager receives status updates from node managers (e.g. availability, load information) and uses them to make allocation decisions. The system manager also responds to resource requests from job managers.

2.2. The job manager

The *job manager* offers a single point of contact for applications to request necessary resources. It hides from the application the complexity of managing the resources that have been allocated by the system manager to a particular job.

When a job is initiated, the job manager locates system managers (by using the Prospero Directory Service if available or from a configuration file) and sends resource requests. If the response from the system manager is affirmative, then the job manager allocates the resources to the tasks in the job and contacts the node manager for each resource in order execute the tasks on the appropriate processors. If a system manager refuses the request, for example when the job manager is not authorized to make the request or when there are no resources available, then the job manager contacts other system managers which can satisfy the request.

2.3. The node manager

Each resource in the system is managed by a *node manager* which is informed by the system manager about job managers that are authorized to use the resource. When the node manager receives a request from an authorized job manager, it responds by loading and executing the requested program. The node manager sends messages to the job manager upon termination or failure of tasks and to the system manager about the availability of the node for future assignments.

3. Motivation for a New Authorization Model

Metacomputing systems cover large networks connecting mutually suspicious domains, which are independently administered.

Consider the following scenario. A user logs onto a machine and wants to perform a computation on a remote machine residing in a different security domain. Let us identify the security issues to be considered:

- Establishing a trust relationship of the users between different security domains. The domain security manager must maintain an authorization database listing principals authorized to request resources belonging to this domain.
- Access control and authorization policies to protect server resources. In a wide area network, it is unlikely that sites would make their resources available to others if there are no means of protection. There should be a flexible mechanism to represent user-defined security policies, such as:
  - type and amount of resources that the node is willing to allocate, e.g. memory, processors, terminal access, access to the local files and directories, network connections

  - applications that can be run on the node, e.g. name of application, version, platform, endorser
  - requirement of payment or accounting for the resources consumed

- Enforcement of the security policies. There should be a mechanism for monitoring execution of the program on a particular node to ensure that the program keeps strictly to the limits imposed by the local administrators

Specification of security policies for principals from multiple administrative domains poses additional problems:

- There are multiple mechanisms for authentication of users in different domains. Therefore, there may be no single syntax for specification of principal names

- Similarly there may be no standard security policy representation. Administrators of each domain might use domain-specific policy syntax and heterogeneous implementations of the policies

Therefore our goal was to design a flexible and expressive mechanism for representing and evaluating authorization policies. It should be general enough to support a variety of mechanisms based on public or secret key cryptosystems and provide integration of local and distributed security policies.

4. Overview of the Model

Our model is designed for a system that spans multiple administrative domains where each domain can impose its own security policies. It is still necessary that a common authentication mechanism be supported between two communicating systems. The model we present enables the syntactic specification of multiple authentication policies, but it does not translate between heterogeneous authentication mechanisms.

4.1. The Extended Access Control List (EACL) framework

EACLs extend the conventional ACL concept by using conditional authorization as an extension to authorization policies, implemented as restrictions (or conditions, we use these words interchangeably) on authentication and authorization credentials. An EACL is associated with an object and lists principals allowed to access this object and the type of access granted.

The objects to be protected in PRM are hosts, but our model is suitable for applications in which the objects are files, physical devices like printers or faxes etc.

4.1.1. Notation

We will use the Backus-Naur Form to denote the elements of our EACL language. Square brackets, `[ ]`, denote optional items and curly brackets, {}, surround items that can repeat zero or more times. A vertical line, `|`, separates alternatives. Items inside single quotes are the terminal symbols. The wild-card symbol `"*"` is used in an EACL just as in the UNIX environment.

4.1.2. EACL : Specification Format

An EACL consists of a set of EACL entries. Each EACL entry represents access control policies directly associated with a particular principal entity. An EACL entry specifies a principal or a list of principals, a set of granted and/or denied access rights, and optionally, any associated conditions.

```
eacl_entry::= principal {principal}
  access_rights {condition}
  {access_rights {condition}} ';'
```

4.1.3. Specification of Principals

The principal is specified according to the following format:

```
principal::=
 principal_type sec_mech  principal_ID |
 'ANYBODY'
principal_type::='HOST' | 'USER' |
 'GROUP' | 'APPLICATION'
```

where `sec_mech` and `principal_ID` are alphanumerical strings.

Different administrative domains might use different authentication mechanisms, each having a particular syntax for specification of principals. For example, an application may use Kerberos V5 [9] as an authentication service. Kerberos V5 provides secret-key based authentication and the format of the Kerberos V5 principal name is `user_name/instance@realm`. Other domains may use DCE to obtain the user's identity credentials, usually identified by a User ID and Group ID. Another domain might use client authentication in SSL, based on public-key cryptography, where principals are identified by a global name, syntactically tied to the X.500 directory. In our model, the syntax of `principal_ID` is defined according to the underlying `sec_mech`, but is tagged to identify the name space.

Principals can be aggregated into a single entry when the same set of access rights and conditions applies to all of them. `ANYBODY` is used to represent all principals regardless of authentication. Examples of principal entities are:

```
    ANYBODY
    USER KERBEROS.V5 kot@ISI.EDU
    HOST IPaddress 164.67.21.82
    GROUP DCE 8
    APPLICATION CHECKSUM 0x75AA31
```

4.1.4 Specification of Access rights

Access rights are specified using the format:

```
access_rights::='<' tag ':' ['-'] value
 { tag ':' ['-']value } | '*' '>'
```

where `tag` and `value` are alphanumerical strings.

Access rights are names for types of access to the protected object. All operations defined on the object are grouped by type of access to the object they represent, and named using a tag. The right is granted when there is no `"-"` preceding the right specification, otherwise the right is denied. The meaning of access control rights is application specific.

4.1.5. Specification of Conditions

Conditions specify the type-specific policies under which an operation can be performed on an object. The format used for specifying access rights conditions is as follows:

```
condition::= type ':' value
```

where `type` and `value` are alphanumerical strings.

A condition is interpreted according to its type. Conditions can be categorized as generic or specific. A condition is generic if it is interpreted by the GAA API. For example: time of day, authentication mechanism, required endorsement. Specific conditions are interpreted by the application: CPU load , memory usage, applications that are to be loaded on the node.

4.1.6. EACL evaluation

The authorization language we presented supports authorization models based on the closed world model, when all rights are implicitly denied. Authorizations are granted by an explicit listing of positive access rights. The open world model, which is based on implicit granting of all rights and listing of only negative authorizations, can be represented in our model by including `ANYBODY *` as the final entry in an EACL. This will grant everybody all rights regardless of authentication. Denial of rights is then specified using negative rights in entries earlier in the ACL.

If one allows both negative and positive authorizations in individual or group entries, inconsistencies must be resolved according to different resolution rules. The design approach we adopted allows the ordered interpretation of EACLs. An ordered evaluation approach is easier to implement, it allows only partial evaluation of EACL and resolves the authorization conflicts. Evaluation of ordered EACL starts from the first to the last in the list of EACL entries. The resolution of inconsistent authorization is based on ordering: the authorizations or denials that have already been examined take precedence over later ones. Other interpretations are possible, but we found that for many such policies, resolution of inconsistencies was either NP-Complete or undecidable.

4.2. GAA API

The GAA API is used by applications to decide whether the subject is authorized for access. In this subsection we provide a brief description of the GAA API routines.

4.2.1. GAA API functions

1) `gaa_get_object_eacl`
This function is called before other GAA API routines which require a handle to the object EACL on which to operate. It returns a handle to the object EACL.

2) `gaa_check_authorization`
This function tells the application server whether the requested operation is authorized, or if additional application-specific checks are required. It returns the code `YES` (indicating authorization) if all requested operations are authorized, `NO` (indicating denial of authorization) if at least one operation is not authorized, `MAYBE` (indicating need for application-specific checks) if there are some unevaluated conditions and additional application-specific checks are required. A list of conditions is also returned, each condition being marked as evaluated or not evaluated, and if evaluated, marked as met or not met. The time period during which the authorization is granted is returned as a condition that may be used by the application.

If no operation was specified as an input, a list of authorized rights is returned as a condition that must be checked by the application. This allows the application to discover access control policies associated with the target object. The application must understand the conditions that are returned unevaluated, otherwise it rejects the request. If the application understands the conditions, it checks them against the information about the request, the target object, or other environmental conditions to determine whether the conditions have been met.

4.2.2 GAA API Security Context

The security context is an argument passed to the GAA API. Some of its constituents follow:

Identity Verified authentication information, such as principal ID for a particular security mechanism. To determine which entries apply, the GAA API checks if the specified principal ID appears in an EACL entry that is paired with a privilege for the type of access requested.

Authorization Attributes Verified authorization credentials, such as group membership, group non-membership and proxies.

Delegated Credentials Delegation is supported through inclusion of delegated credentials, such as those supported by *restricted proxies* [6].

Evaluation and Retrieval Functions for Upcalls These functions are called to evaluate application-specific conditions; request additional credentials and verify them.

5. Applying the Distributed Authorization Model to PRM

We will first discuss the integration of the EACL framework into PRM and then we will show how PRM makes use of the GAA API to enforce the policies expressed through the EACL.

5.1. EACL conditions specific to PRM

Our experience with deploying PRM on a wide scale has shown that administrators are more willing to grant access to their workstations if they can restrict access to users or organizations they trust. Administrators must also be able to specify restrictions on the specific applications that will run on their systems. These restrictions are important in the context of movement of executable or interpreted content between different systems and platforms, i.e. what is usually known as “mobile code”. We have therefore introduced EACL conditions specific to this type of policy:
- name of application:
```
  application_name :  matlab
```
- name of interpreter, in case the application is written in an interpreted language:
```
  interpreter_name :  Tcl
```
- platform the application runs on:
```
  application_platform :  Solaris
```
- version number for the application:
```
  application_version :  1.0
```
- endorser or certifying authority for the application:
```
  application_endorser :  Globus
```
Authorizing a user to run an application on the specific resources is often not detailed enough for system administrators. What is needed is a way to impose and enforce limits on the physical resources consumed by the applications. To specify these limits, PRM uses specific EACL conditions:
- CPU load, expressed as maximum percentage of the CPU time that an application is allowed to use:
```
  cpu_load :  20%
```
- memory usage, expressed as maximum size in Kbytes that a process can occupy in main memory:
```
  mem_usage :  1024
```
- machine idle time, expressed as minimum interval in minutes that the machine has to be idle before any application managed by PRM is allowed to run:
```
  idle_time :  30
```

5.2. Using the GAA API in PRM

5.2.1. Creation of the GAA API security context

For communications, PRM uses calls to the Asynchronous Reliable Delivery Protocol (ARDP), which handles several security services including authentication, integrity and payment. ARDP calls the Kerberos library through a security API, requesting the principal’s identity, which is placed into the security context and is passed to the GAA API. Figure 1 shows the flow of control: the system manager calls ARDP requesting the principal’s identity (1); the request and the verification of the principal’s identity credentials take place (2, 3, 4, 5); ARDP places the principal’s authentication credentials in the security context (6a) and returns it to the system manager (6); the system manager calls the GAA API (7); the security context, containing the verified principal’s identity is being passed into the GAA API (7a).

When additional security attributes are required for the requested operation, the list of required attributes is returned and obtained by the application. The application or transport may add an upcall function to the security context which is passed to the GAA API and used to request required additional credentials. Such additional credentials are requested, verified, and added to the security context by this upcall function.

5.2.2. Authorization Walk-through

Here we present two authorization scenarios. First, let’s consider a request from user Joe to run `matlab` on the host `kot.isi.edu` on Monday at 7:30 PM. Assume that this host has the following ordered EACL stored in the Prospero directory service:

```
USER  kerberos.v5  joe@ISI.EDU
    <HOST : load > time_window : 6AM-8PM,
                   cpu_load : 20% ;
GROUP kerberos.v5  operator@ISI.EDU
```

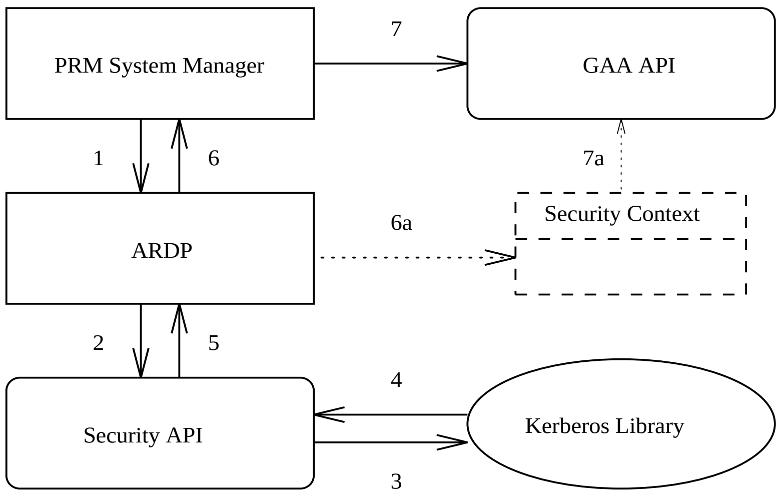

Figure 1. Creation of the GAA API security context

```
        USER  kerberos.v5  tom@ISI.EDU
        <HOST : * > <DEVICE : power_down> ;
ANYBODY <HOST : load> time_day : sat-sun,
                      time_window : 6AM-8PM,
                      cpu_load : 10% ;
```

When a job manager contacts a system manager with the request for resources, the system manager calls the `gaa_get_object_eacl` function to obtain a handle to the EACL of `kot.isi.edu`. The upcall function for retrieving the EACL for the specified object from the Prospero virtual file system is passed to the GAA API and is called by `gaa_get_object_eacl`, which returns the EACL handle. The system manager calls ARDP, which handles authentication as explained in Figure 1 and section 5.2.1. If Joe is authenticated successfully, then the verified identity credential is placed into the security context, specifying Joe as the Kerberos principal `joe@ISI.EDU`.

The `gaa_check_authorization` function is called by the system manager, which asks if Joe is authorized to load `matlab` to `kot.isi.edu`. In evaluating the EACL, the first entry applies. It grants the requested operation, but there two conditions that must be evaluated. The first condition `time_window : 6AM-8PM` is generic and is evaluated directly by the GAA API. Since the request was issued on Monday at 7:30 PM this condition is satisfied. The second condition `cpu_load : 20%` is PRM-specific. If the security context passed by PRM defined a condition evaluation function for upcall, then this function is invoked and if this condition is met then the final answer is `YES` (authorized).

During the execution of the task the node manager is monitoring if the task is abiding to the limits imposed on the local resources and authorization time. If the corresponding upcall function was not passed to the GAA API, the answer is `MAYBE` and the set of conditions is returned. Conditions are marked as either evaluated or not evaluated. In our example `time_window : 6AM-8PM` was evaluated and met; `cpu_load : 20%` was not evaluated and should be checked by PRM.

Next, we present an authorization scenario where additional credentials are needed. Let's consider a request from user Joe to run `matlab` on the host `kot.isi.edu` on Monday at 8:30 PM. In EACL evaluation, the first entry applies but does not grant this operation, since the first condition is not met. The temporary answer is `NO` (not authorized). The second entry grants this permission. If the security context defines a credential retrieval function for upcall, then this function is invoked and if either a group "operator" membership credential or delegated credential from user Tom for Joe is obtained, then the final answer is `YES`. If the credential retrieval upcall function was not passed to the GAA API, the answer is `NO`.

6. Managing the EACL using the Prospero Directory Service

We have mentioned in section 2 that PRM deals with scalability issues by splitting the task of managing the resources across the three types of managers. Our goal in designing a mechanism for the management of the EACL files was to enable easy sharing of a default authorization policy among node managers, while allowing customization of the policy at the level of individual hosts.

We use the Prospero Directory Service [7] to store the information associated with the EACL files. The EACL files themselves are objects stored in the Prospero directory service.

The following scenario shows how the management of the files is accomplished:

1. The administrator of the domain whose resources are managed by a system manager running on host A creates an EACL file describing the default authorization policy which applies to the domain.

2. The administrator registers with the Prospero server. We supply a script which takes as input the location of the EACL file and creates a Prospero object representing a link to the file, together with two attributes for the link:

```
SYSTEM_MANAGER A
EACL_DEFAULT True
```

3. If the administrator of a particular host B in the domain managed by A wants to specify a local authorization policy different from the default one, a similar procedure is followed, except that the link to the local EACL file is created with the following attributes:

```
NODE_MANAGER B
EXTEND_DEFAULT Prepend/Append/Replace
```

(`Prepend` if the local policy is prepended to the default policy, `Append` if the local policy is appended to the default and `Replace` if the local policy completely replaces the default)

4. When a system manager is contacted by a job manager with a request for resources, it first authenticates the user, as was explained in the authorization scenario in section 5. Before requesting resources from a node manager running on a particular node B, the system manager retrieves the EACL file associated with that node by looking for a link with attribute `NODE_MANAGER = B`. If no such link is found, the default EACL file provided for the domain will be used and it will be obtained by retrieving a link with attributes `SYSTEM_MANAGER = A` and `EACL_DEFAULT = True`. If a link with `NODE_MANAGER = B` is found, then a second query is issued for the value of the attribute `EXTEND_DEFAULT`. If the value is `Prepend` or `Append`, the system manager will have to retrieve the default EACL file first, and then prepend or append to it the contents of the EACL file for node B (note that the distinction between the two cases `Prepend/Append` is necessary because the EACL evaluation takes into account the order of the EACL entries). If the value is `Replace`, then only the EACL file for node B will be retrieved and used.

5. After retrieval of the EACL file, evaluation of the conditions listed in the file follows, as detailed in the authorization scenario from section 5. If all the conditions are met, the job manager is allowed to use the resources on that particular host.

6. During the execution of tasks on a particular host, the node manager periodically checks whether the task is abiding to the limits imposed on the local resources. If it is not, then the task is interrupted and the job manager is notified.

7. Related Work

Nagaratnam and Byrne [5] present a model for Internet user agents to control access to client resources. This model protects client machines from hostile downloadable content and allows the client to selectively grant access to trusted agents. The authenticity of the code is based on digital signatures of principals certifying it. All access control requests are mediated by calling a security manager component and decisions are based on the user's access control specifications stored in the policy database.

The model is restricted to using the Javakey utility as an authentication mechanism based on public key digital signatures, while our model is general enough to use a variety of security mechanisms based on public or secret key cryptosystems.

Another disadvantage of that model is the duplication of common information. Each user has to maintain a database of any principals specified in the policy database and their public keys, as well as specification of groups. These databases should be properly integrity-protected. In contrast, PRM uses Kerberos to achieve strong authentication. The authentication database is maintained centrally by the KDC and stored on a physically secure machine. Our model also supports a group certification mechanism. A group server maintains and provides group membership information, and issues group membership and non-membership certificates. The certificates are placed into the GAA API security context and checked by the GAA API when making authorization decisions. There is no need for each user to maintain authentication and group specification databases locally.

The Generalized Access Control List framework described by Woo and Lam [10] presents a language-based approach for specifying authorization policies. The GACL model supports only system state-related conditions within which rights are granted, such as current system load and maximum number of copies of a program to be run concurrently. This may not be sufficient for distributed applications. Our model allows fine-grained control over the conditions.

Both restricted proxies [6] and the use-condition model [4] allow conditions and privilege attributes to be embedded in authorization credentials or certificates. These mechanisms can be readily integrated with the authorization model presented here: the restrictions or conditions caried in the proxy or certificate are evaluated by the GAA API in addition to the restrictions in the matching EACL entry.

The CRISIS architecture [1] provides ACLs that are related to the type of the protected object. For example, file ACLs list principals allowed read, write or execute access to the file, whereas node ACLs contain principals allowed to run jobs on the node. CRISIS ACLs do not support constraints on the resources that principals are allowed to consume. The emphasis of our work is on providing a general framework for representing security policies and facilitating authorization decisions for applications. Our model provides a uniform authorization mechanism that is capable of supporting different operations and different kinds of protected objects.

The Tivoli Management Environment (TME 10) is a commercially available system from IBM which takes a role-based approach to security [3]. TME roles are named capabilities, containing a list of objects and access permissions to those objects. Objects can have default access and can be associated with more than one role. Each role will have a different level of access to the object. Roles are defined to support a particular job function within an organization, e.g. customer support or management. Groups are assigned roles, thus giving members of those groups access capabilities to the objects assigned to those roles. The TME security model can be easily expressed by our EACL framework:

1) An EACL is associated with each object to be protected. Default access to the object is represented by including `ANYBODY default_rights` as the last entry in the EACL.

2) The object's EACL will contain entries listing groups and a set of access rights, granted by TME roles assigned to each group.

For example, in TME the group `Support_users` is assigned the `Customer_support` role which grants RWE permissions to file `/cust_supp_dir/*` In our system, an EACL associated with the object `/cust_supp_dir/*` will have the following entry:

```
GROUP sec_mech support_users FILE:R
FILE:W FILE:E ;
```

TME lacks flexibility in supporting user-defined security policies. It has a fixed predefined set of object types and generic access permissions that are available on each object type. In addition, the TME model requires the creation of a new role to include each new combination of objects and access rights. This becomes very cumbersome for systems where a large number of operations exist on various objects.

8. Conclusions

By extending the traditional Access Control Lists with restrictions on authorized rights, and by using General Authorization and General Authorization and Access API, it becomes possible to support a flexible distributed authorization mechanism allowing applications and users to define their own access control policy types, and integrating local and distributed security policies. The problem of translation of the policies is addressed by using general or PRM-specific evaluation functions. In this paper, we have omitted discussion of many practical details due to space limitation. A prototype of the presented model has been developed at the Information Sciences Institute of the University of Southern California.

Acknowledgments

This research was supported in part by the Defense Advanced Research Projects Agency under the Scalable Computing Infrastructure (SCOPE) Project, TNT, Contract No. DABT63-95-C-0095, Security Infrastructure for Large Distributed Systems (SILDS) Project, Contract No. DABT63-94-C-0034, and by a grant from Xerox Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Intelligence Center and Fort Huachuca Directorate of Contracting, the Defense Advanced Research Projects Agency, the U.S. Government, or Xerox Corporation.

References

[1] E. Belani, A. Vahdat, T. Anderson, and M. Dahlin. The CRISIS wide area security architecture. *Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas*, January 1998.

[2] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, Summer 1997.

[3] IBM. TME 10 security management. http://www.tivoli.com/redbooks/html/sg242021/2021fm.htm, October 1997.

[4] W. Johnston and C. Larsen. A use-condition centered approach to authenticated global capabilities: Security architectures for large-scale distributed collaboratory environments. LBNL Report 38850.

[5] N. Nagaratnam and S. Byrne. Resource access control for an Internet user agent. *Proceedings of the third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon*, June 1997.

[6] B. C. Neuman. Proxy-based authorization and accounting for distributed systems. *Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh*, May 1993.

[7] B. C. Neuman, S. Augart, and S. Upasani. Using Prospero to support integrated location-independent computing. *Proc. Symp.on Mobile and Location-Independent Computing, Cambridge, MA*, pages 29–34, August 1993.

[8] B. C. Neuman and S. Rao. The Prospero Resource Manager: A scalable framework for processor allocation in distributed systems. *Concurrency: Practice and Experience*, June 1994.

[9] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, pages 33–38, September 1994.

[10] T. Woo and S. Lam. A framework for distributed authorization. *Proc. ACM Conference on Computer and Communications Security, Fairfax, Virginia*, November 1993.