

University of Montana

## ScholarWorks at University of Montana

---

Graduate Student Theses, Dissertations, &  
Professional Papers

Graduate School

---

1995

### Object oriented design of the groupware layer for the Ecosystem Information System

Venugopal V. Hemige  
*The University of Montana*

Follow this and additional works at: <https://scholarworks.umt.edu/etd>

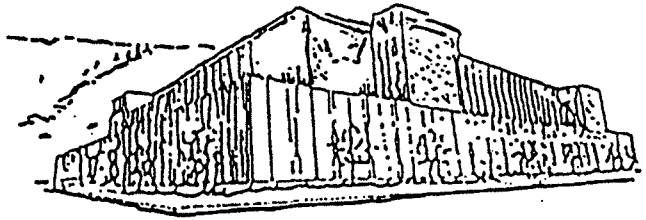
**Let us know how access to this document benefits you.**

---

#### Recommended Citation

Hemige, Venugopal V., "Object oriented design of the groupware layer for the Ecosystem Information System" (1995). *Graduate Student Theses, Dissertations, & Professional Papers*. 5093.  
<https://scholarworks.umt.edu/etd/5093>

This Thesis is brought to you for free and open access by the Graduate School at ScholarWorks at University of Montana. It has been accepted for inclusion in Graduate Student Theses, Dissertations, & Professional Papers by an authorized administrator of ScholarWorks at University of Montana. For more information, please contact [scholarworks@mso.umt.edu](mailto:scholarworks@mso.umt.edu).



Maureen and Mike  
**MANSFIELD LIBRARY**

The University of **MONTANA**

---

Permission is granted by the author to reproduce this material in its entirety,  
provided that this material is used for scholarly purposes and is properly cited in  
published works and reports.

*\*\* Please check "Yes" or "No" and provide signature \*\**

Yes, I grant permission

No, I do not grant permission

Author's Signature

Date

06/03/95

Any copying for commercial purposes or financial gain may be undertaken only with  
the author's explicit consent.

**Object Oriented Design of the Groupware Layer**  
for the  
**Ecosystem Information System**

by

**Venugopal V. Hemige**

**Bachelor of Technology**

**Karnataka Regional Engineering College, India 1993**

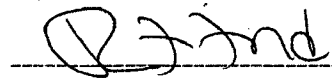
**Presented in partial fulfillment of the requirements**

**for the degree of**

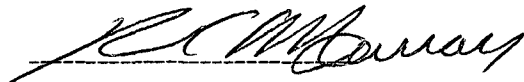
**Master of Science**

**University of Montana**

**1995**



Chairman, Board of Examiners



Dean, Graduate School

June 14, 1995

Date

UMI Number: EP40557

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP40557

Published by ProQuest LLC (2014). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 - 1346

Object-oriented design of the Groupware Layer for the Ecosystem Information System.

Director: Prof. Ray Ford



In a distributed system, sharing of data and other resources is one of the key goals. But while allowing users access to these resources, it is important to make sure that distributed system resources are accessed only by those who are authorized to do so and only in precisely the ways they are authorized to do so. The implementation of EIS has reached a stage where the database is being populated with information by ecosystem researchers from our collaborating laboratories. Before people start using EIS extensively, the primary need is to design and implement the groupware layer appropriate for EIS. This thesis investigates the requirements of the groupware scheme for EIS and presents an appropriate design. The design of a reliable authentication scheme is presented here. The thesis makes use of a scenario-based object-diagram representation to show how different objects in the client/server processes will interact with each other during program execution.

## Acknowledgements

I would like to express my heartfelt gratitude towards Prof. Ray Ford for his relentless support and guidance with my thesis. I particularly appreciate his patience in carefully reading every line of my draft through several iterations before coming up with a draft that could be presentable. I am deeply indebted to Prof. Lynn Churchill and Prof. Nick Wilde without whose guidance and help, this thesis would not have been possible.

Special thanks to Mrs. Kathy Lockridge for her ever-cheering support and help with all the formal details regarding making a thesis possible. Last but not the least, I would like to thank the Department of Computer Science, University of Montana, for all the computing resources and relevant literature that made my thesis possible.

Date: June 02, 1995

Venugopal Hemige

## Table of Contents

<b>Abstract</b> .....	ii
<b>Acknowledgements</b> .....	iii
<b>Table of Contents</b> .....	iv
<b>1. Introduction</b> .....	1
<b>2. Background</b> .....	3
2.1 An Overview of Object-Oriented Modeling.....	3
2.2 The Goal for EIS .....	4
2.3 Object-Oriented Design of EIS.....	5
<b>3. The Groupware Layer</b> .....	9
3.1 Requirement.....	9
3.2 Basic Entities.....	11
3.3 An Example Group_info Object.....	12
3.4 EIS Database Organization.....	17
<b>4. The 3-A Aspects of the Groupware Layer</b> .....	20
4.1 User Authentication.....	20
4.2 Authentication in EIS.....	22
4.2.1 Local Client Authentication.....	23
4.2.2 Remote Server Authentication.....	23
4.3 Authorization.....	26
4.4 Access Control.....	28
<b>5. Scenarios</b> .....	29
5.1 Notations.....	29
5.2 Basic Design.....	30

5.2.1 - 5.2.8 Example Scenarios.....	31-47
<b>6. Summary and Conclusion.....</b>	<b>48</b>
6.1 Implementation Status.....	48
6.2 Directions for Future Research.....	49
<b>Appendix A: Pseudocode for Authenticating a user.....</b>	<b>51</b>
<b>Appendix B: Installation Guide for EIS.....</b>	<b>52</b>
<b>References.....</b>	<b>55</b>



## 1. Introduction

The Ecosystem Information System (**EIS**)[1, 2, 3] is a distributed database containing various types of information of interest to ecosystem modelers and managers. Included in this database are meta-data descriptions for various data sources, datasets, and modeling components. EIS is designed and implemented using object-oriented technology. The current implementation of EIS implements only the “core access technology” demonstrating the potential for sharing interpreted objects via EIS. But it does little or nothing at all in terms of security. Having come to a stage where the software can now be used by ecosystem modelers to populate the database so that it can be shared and accessed by other users on the network, data security has become one of the key issues. The goal of EIS is to allow users anywhere on the network to share distributed resources in a network-transparent manner. In a distributed system, sharing of data and other resources is one of the key goals. But while allowing users access to these resources, the distributed system should allow each user to do those things that he/she is authorized to do, and to prevent him/her from doing things that are not authorized. The work described here is the object-oriented design and prototype implementation of this *groupware layer* for EIS, as an extension to the current prototype system.

In this thesis, I discuss the three important aspects of the groupware layer in EIS and present an object-oriented design of the different classes and objects that encapsulate these aspects. I perform a careful study of the different popular authentication mechanisms available and present a viable authentication for EIS. The security mechanism presented here is aimed at ensuring that only the right users are allowed access to the database and only those services and resources that they are authorized to. Since authentication needs to be done for every service requested by the user, it might be better to cache the groupware information whenever a client is accessing a particular portion of the groupware layer. The entire design process makes use of different scenarios to help the designer analyse the effectiveness of a particular design aspect

and the way to go around implementing it. The scenarios described in this thesis should help the reader understand the design concepts more clearly too.

Briefly summarized, the purpose of this thesis is to :

- design and implement a reliable security mechanism for a distributed system like EIS.
- use object-oriented methodology in the design process.
- provide a strong authentication scheme to ensure that only the right users access the EIS database resources.
- manage the costs involved in making the authentication/authorization requests.
- adopt a scenario-based object-diagram representation to describe how objects will interact with each other during program execution.

## 2. Background

### 2.1 An Overview of Object-Oriented Modeling

Object-oriented design is built upon a sound engineering foundation, whose elements we collectively call the *object model* [4]. The object model encompasses the principles of abstraction, encapsulation, modularity, hierarchy, concurrency, and persistence. The building blocks in any object-oriented design methodology are concepts of class, instance(or object) and method. A class is a description of objects that share a common structure and a common behavior. An object is simply an instance of a class. An object can be considered to have state, behavior and identity. The state of an object is determined by a set of attributes of the object plus the current values of each of these attributes. In general, the internal state of an object is hidden from other objects and hence is not directly accessible. However an object can make parts of its state available to other objects or let other objects perform actions on this state through a set of visible attributes and operations. The client/server model is a good example of an object-oriented modelling where the two main objects under consideration are the client and the server processes. The client/server model is used to describe the use of networks containing two types of processes that have an asymmetric relationship. The client process makes requests for services and the server provides the services on request. Objects that share common attributes and operations are grouped into a class.

Object-Oriented design[4] is an incremental, iterative process in which the products of design, a set of interacting objects, gently unfold over time. We start the object-oriented design process by discovering the classes and objects that form the vocabulary of our problem domain. The process of object-oriented design generally tracks the following order of events:

- Identify the key abstractions in the problem space (the significant classes and objects) and describe the mechanisms that provide the behaviour required of

objects, so that they can work together to achieve some function.

- Identify the semantics of these classes and objects.
- Identify the relationship among these classes and objects.
- Implement these classes and objects.

This is an incremental process: the identification of new classes and objects usually results in the need to refine and improve upon the semantics of and relationships among existing classes and objects. It is also an iterative process: implementing classes and objects often leads us to the discovery or invention of new classes and objects whose presence simplifies and generalizes our design.

Given an object-oriented design for a target system, the next step is to somehow test the design, prior to implementation, to see if it provides the correct set of facilities. A scenario is a description of how objects will interact with each other during program execution to perform a specific activity. A scenario can be represented in many different ways, but one of the most convenient representations is as a diagram showing objects and the exchange of operation calls and results that represent their client/server relationship in this particular scenario.

For a client/server system such as EIS, a set of carefully selected scenarios can help the system designer to visualize different approaches and determine the correctness of the design, its efficiency and how it would function in a real implementation.

## **2.2 Goal for EIS**

Modern ecosystem management and analysis is an application area that demands extensive information sharing between different organizations and different sites within an organization. The goal of the EIS project is to create a network-accessible repository of ecosystem information that provides access to various types of information of

interest to ecosystem modelers and managers in a user-friendly manner, ensures reasonable security, allows the distribution of locally created material and contribution from outside users, and is populated with material sufficient to illustrate both its use and its value to potential users. The second major goal in EIS development is to enhance the level of access provided by Internet-based tools for objects such as numerical datasets, program components and meta-data descriptions for various data sources. Currently available Internet tools support the display of hypertext documents and images in standard formats, but provide access to other types of objects merely as uninterpreted files. Our goal is to develop tools that allow us to construct a web-work of hierarchical dataset descriptions and dataset instances, allowing a potential user to transparently navigate from site to site, browse through dataset descriptions, locate datasets and data transformations of interest, and easily add datasets and dataset descriptions.

### 2.3 Object-Oriented Design of EIS

The base EIS object-oriented design (referred to as EIS 0.9) implements the core of information distribution functionality, but without any access control or other mechanism for security. An object diagram of the EIS 0.9 design is depicted in Figure 1. The core system consists of five primary types of one-per-user or one-per-host objects that implement the necessary services. The three one-per-user objects **GUI**, **OME** and **ORB-client** are encapsulated under one object called **EIS-client**. **EIS-client** is invoked as a client process by a user. On every host, the one-per-host objects - **Object Request Broker (ORB-server)** and **Object Database Manager (ODBM)** are encapsulated under the **EIS-server**. At the heart of the **EIS-server** are two one-per-host objects. The **ORB-server** manages locally generated requests, and resolves such requests by forwarding them to either the local filesystem or to the **ORB-server** on a remote host. Thus the **ORB-server** must also respond to remotely generated object requests for objects stored on its local database. The **ORB-**

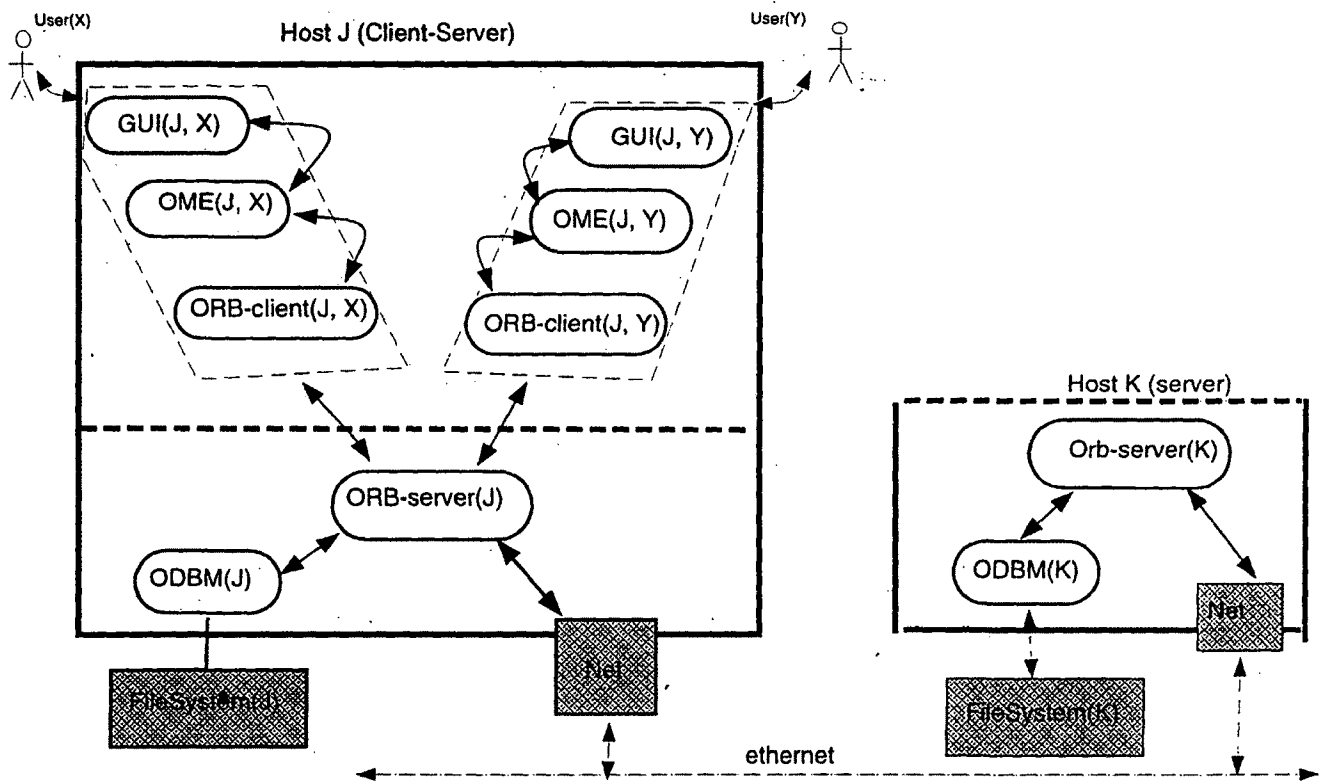


Figure 1: EIS 0.9 Object-Oriented Design

server also manages all communication required to store and retrieve information from the local file system, phrased in terms of messages/responses with the ODBM. The ODBM encapsulates all aspects of file system storage, thus hiding details associated with file naming, directory structure, etc.

The EIS-Client is a set of three one-per-user objects, namely GUI, OME and ORB-client. The user-interface, which involves the X/Motif details, is encapsulated in the GUI. The GUI maintains only enough local state to allow the display to be drawn. In order to allow efficient implementation of operations that modify the display another object, the Object Management Engine (OME), maintains a more complete local state. Thus, the class hierarchy is represented in different forms and in different degrees of detail in different objects. To maintain the local state, the OME uses several smaller objects which are all encapsulated in the OME for our explanation here. The OME routes all its requests for EIS database entities to the ORB-Client, which in turn forwards each request through the ORB-Server to either the local ODBM or across the network to another ORB-Server.

The EIS 0.9 has been implemented in C++, with the GUI front-end written using X/Motif. A public-domain remote procedure call mechanism (SUN RPC) is used for host/host communication. This initial implementation executes on the IBM RS/6000 Unix workstations. The code should be portable to a wide range of Unix workstations within the portability bounds of different C++ compilers, X/Motif library implementations, and the SUN RPC. However, the prototype EIS 0.9 implements only the core access technology demonstrating the potential for sharing interpreted objects via EIS. Since the remote procedure call mechanism used is written in C, an encode/decode method has to be used to convert the C++ objects into a format acceptable to the RPC library. All this is encapsulated within the ORB-Client and ORB-Server. The present implementation of EIS does not yet provide support for robust operation, group-oriented security, or other forms of access regulation. The EIS data-repository is organised hierarchically using an object-oriented framework to or-

der the myriad collection of components used in ecosystem modeling. In collaboration with other ecosystem modeling laboratories, the repository is being populated with datasets and modeling tools from important ecosystem modeling and management applications.



### 3. The Groupware Layer

#### 3.1 Requirements

In a distributed system, sharing of data and other resources is one of the key goals. But while allowing users access to these resources, it is important to make sure that distributed system resources are accessed only by those who are authorized to do so and only in precisely the ways they are authorized to do so. The implementation of EIS has reached a stage where the database is being populated with information by ecosystem researchers from our collaborating laboratories. Before people start using EIS extensively, the primary need is to design and implement the groupware layer appropriate for EIS. The design of the security system for a distributed system like EIS is significantly different from that for a uniprocessor system. A user could be anybody on the network. An EIS group could consist of users from a range of hosts. The access privileges to a hierarchy could be more detailed than the *(read, write, execute)* accesses as described in the Unix system. The owner of a hierarchy should probably be able to specify the access groups for each/some of the groups defined. Also since parts of a hierarchy could be contributed by a certain group of users, they ought to be able to set authorization on their parts of the hierarchy for other users.

Ideally there would be a system supported layer of client/server facilities that allowed servers to “publish” their services yet restrict access to them, and clients to identify themselves and gain access to these services. The Distributed Computing Environment (**DCE**)[6] is a convenient tool for such purposes. The set of DCE *Security Service* facilities provides a robust set of capabilities to ensure that services are made available only to properly designated parties, without inconveniencing legitimate users. DCE Security implementation is based on Kerberos[7, 8] and uses very well known encryption algorithms (Data Encryption Standard - DES). An ideal situation would be to implement EIS using DCE distributed support for security. But unfortunately, while DCE is fast becoming a standard, it is still not widely available

nor completely compatible across different vendor platforms. In order to permit the implementation and use of EIS now, our plan is to implement EIS using the SUN RPC [5] which is a public-domain software portable and compatible across a wide range of Unix Workstations. As regards security, the disadvantage is that the SUN RPC offers an authentication scheme which cannot provide a high level of security for an arbitrary design. The complete functional design of EIS demands this level of security for its groupware scheme. The work described here includes the complete EIS groupware design, and a modified, more limited form that can be implemented securely with only RPC support.

There are three facets of the proposed groupware layer: authentication, authorization, and access control. The three "As" work together to provide security in EIS. Authentication means validating the user's identity. Authorization means determining to what groups this user belongs. Access control is deciding the set of access privileges this user should and should not be allowed. The authorization and access control decisions in the EIS distributed system are encapsulated in the EIS-server. For every operation that a user requests, the EIS-client authenticates the user (determines the user's identity), authorizes the user's membership, then verifies from the concerned EIS-server if the user has the right to perform the access implied by the requested service. Only at the end of this process is the user allowed to perform the requested action.

There is overhead in making authentication/authorization/access requests. In general, information required to resolve these requests may be stored globally, so the overhead is passed along to the concerned EIS-server. If the server is remote from the originating user, considerable performance delays could result. Since such requests are made for every operation requested by the client, the groupware layer is likely to be the most frequently used resource in EIS.

There are three major issues to be addressed in the groupware implementation.

First is managing the costs involved in resolving the 3-A requests. Second is recognizing the level of security and functionality that can be attained without true distributed system support for security, i.e. using only RPC. The third is balancing communication and data replication in the design of features to support the 3-A's. Keeping only a single copy of the pertinent groupware data will lead to excessive communication costs, whereas replicating the data at each client might lead to an inconsistency between the replicated groupware data thereby leading to erroneous results. An appropriate balance between centralized and replication schemes, as described in [9, 10], must be designed to make the 3-A implementation efficient, reliable and robust.

Actually integrating a well thought-out groupware layer with the existing system code will be the final step. All of the new objects are encapsulated within the server, only requiring changes in the ORB-client and ORB-server to handle the communication requests and minor new GUI capability for operations concerning the groupware layer. We shall identify this new EIS system with the groupware layer with a new version number (EIS 1.0) in order to distinguish the new EIS from the existing system (EIS 0.9).

### **3.2 Basic Entities**

Logically, the groupware layer is based on a collection of classes that define the characteristics of active entities such as users, hosts, and groups, of permissions, and of privileges. These classes are defined below, followed by a discussion on the procedure used in EIS 1.0 in handling the 3-A aspects of Authentication, Authorization and Access Control. Figures 2, 3 and 4 show the classes that encapsulate these definitions. Figure 2 describes the abstractions that encapsulate concepts related to authentication and authorization. Figure 3 is a description of the abstractions that encapsulate key concepts in defining access permissions. Figure 4 shows the different classes of users in the EIS system, based on their access privileges. A single instance

of *group\_info* will encapsulate the authentication and authorization operations for a group of hosts. An instance of *priv\_info* will encapsulate the permissions for different groups for each hierarchy, and hence is used for access control.

Figure 5 also suggests where the different parts of the groupware layer fit into the EIS system. Logically, the EIS system as a whole consists of a number of cells. Each cell is made up of a set of hosts. Ideally a host belongs within only one cell but it can be otherwise. One host in each cell is designated as the cell's *main\_server* which is very much like the other EIS servers except that it encapsulates the *group\_info* and the *hier\_id\_list* for that cell. The *group\_info* defines a set of groups for the cell and the scope of a group includes all the hosts within the cell. Every EIS host maintains a database of all the hierarchies that were created on that host. For example, in Figure 5, the hierarchies  $HIER_{M1}$  indicates that this hierarchy is one of the hierarchies stored on host M. Hierarchies are identified by names that are unique within the domain of a cell. Along with each hierarchy, we have a *priv\_info* object that keeps track of the access permissions for different groups to the hierarchy.

### 3.3 An Example *group\_info* Object

An example instance of *group\_info* is shown in Figure 6 to better illustrate how the objects are organized. Cell A is one of the cells in the EIS system as depicted in Figure 5. The *group\_info* consists of:

1. A list of *groups* (EIS\_ADMIN, dasl, eis) and the users belonging to each of these groups
2. A list of *vita* with a cross reference indicating the list of *groups* each user belongs to.

This instance of *group\_info* for cell A is encapsulated within the *main\_server* for the cell, host "cs.umt.edu". Suppose now, user "trish@radiator.cs.umt.edu" creates a hi-

Class	Attributes	Description
<b>user</b>	email-id	Since EIS can be accessed from anywhere on the network, we define a EIS user as a person with a valid email account on a Unix system. Hence a user's attribute is the email-id of the account.
<b>host</b>	IP address	An EIS host is a host on which an EIS server runs as a daemon process.
<b>cell</b>	list<host>	A cell in EIS terms is a group of co-operating EIS hosts, each running its own EIS server. Each cell has a designated <i>main_server</i> host which has the extra attributes described below.
<b>group</b>	group_name, list<user>	A group encapsulates a group_name and a list of users that belong to the group.
<b>vita</b>	user, list<group>	A vita is defined as the list of groups that a user belongs to. It consists of a <i>user</i> attribute and a <i>list&lt;group&gt;</i> attribute.
<b>group_info</b>	eis_admin, list<group>, list<vita>	This contains the information describing which user belongs to each group and what groups each user belongs to. This object encapsulates the <i>authorization</i> process of the groupware layer. The list<vita> is a cross-reference to the list<group> so that the authorization process is faster and more efficient. There should be only one <i>group_info</i> object per EIS cell.
<b>EIS_server</b>	Hierarchies	This is the EIS server process that runs as a one-per-host daemon process. It handles requests from EIS clients and manages the database
<b>main_server</b>	Hierarchies, hierarchy_list, group_info	The <i>main_server</i> in a cell is different from the other EIS servers in that it encapsulates a few additional objects that are absent in other servers. These attributes are the <i>group_info</i> and <i>hierarchy_list</i> (a table of all the hierarchies within the cell).

Figure 2: Abstractions encapsulating Authentication/Authorization

<b>Class</b>	<b>Attributes</b>	<b>Description</b>
<b>priv_info</b>	owner-id, owner_perms, world_perms, list<group_perms>	Priv_info is the object that keeps access right information for different privilege classes.
<b>perms</b>	read, modify, modify_if_extended, delete, delete_id_extended, extend, execute	This object contains the different access rights defineable in a EIS hierarchy.
<b>group_perm</b>	groupname, perms	The permissions for a group.

Figure 3: Abstractions encapsulating Access Control

## **Class      Attributes and description**

**Privilege Classes** The Privilege classes within EIS are EIS\_Admin[Root], Owner, Group(s), World. The EIS\_Admin is the only class of user(s) who are privileged to access/modify the *group\_info*. Every hierarchy has a Owner who can change/add/delete the permission attribute of any of the groups. Each of the different groups including the world can be set to have different access rights.

Figure 4: Privilege Classes

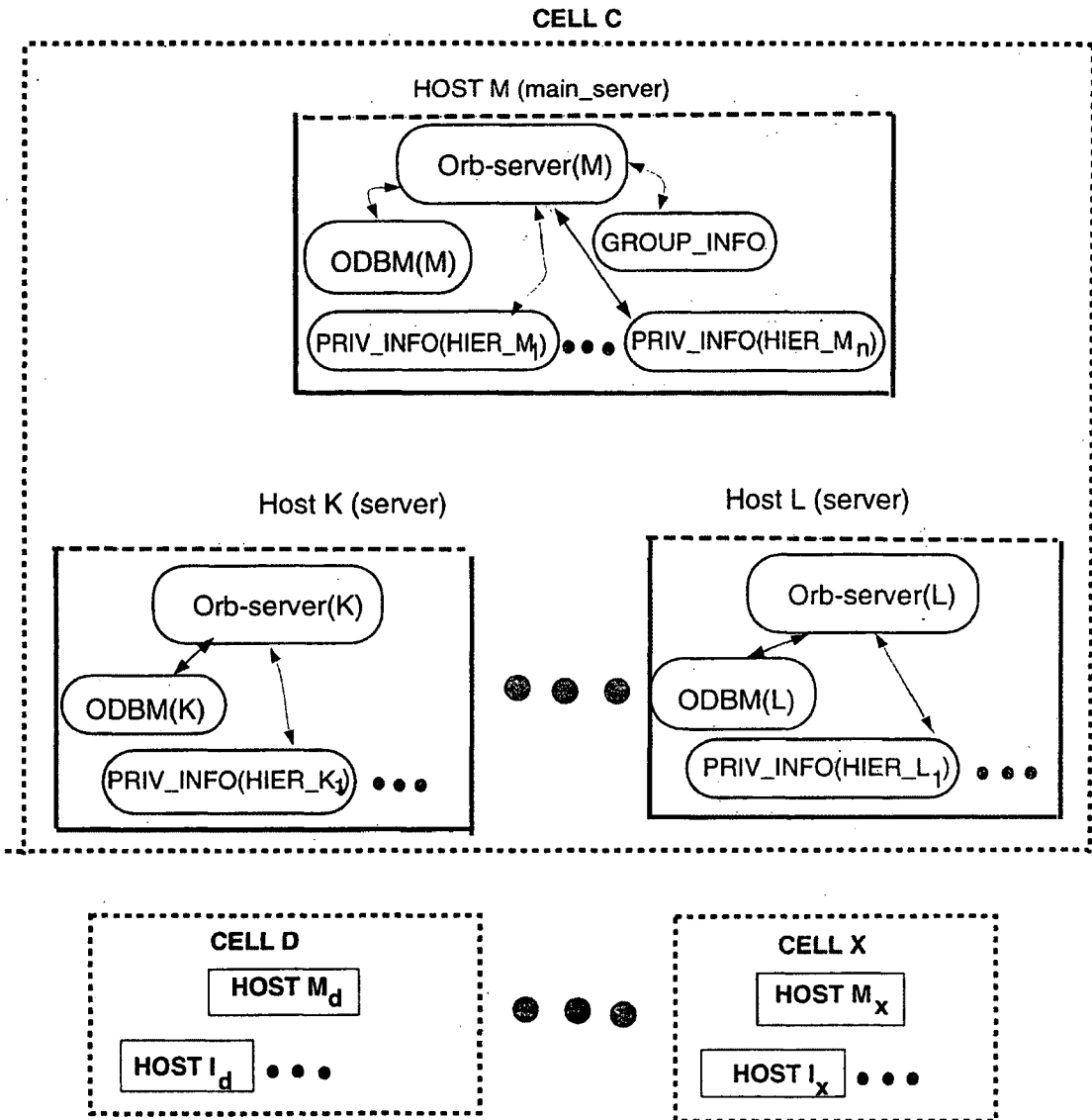


Figure 5: EIS Object-oriented diagram including the Groupware Layer

```

EIS_ADMIN: venu@eisgate.cs.umt.edu, vijayant@eisgate.cs.umt.edu      /* a group */
dasl: ford@cs.umt.edu, trish@radiator.cs.umt.edu, ford@wilfred.umt.edu /* a group */
eis: righter@wru.umt.edu, dthompsn@cs.umt.edu , ford@wilfred.umt.edu /* a group */

venu@eisgate.cs.umt.edu: EIS_ADMIN                                     /* a vita */
vijayant@eisgate.cs.umt.edu: EIS_ADMIN                               /* a vita */
ford@cs.umt.edu: dasl                                               /* a vita */
ford@wilfred.umt.edu: dasl, eis                                       /* a vita */
dthompsn@cs.umt.edu: eis                                             /* a vita */
trish@radiator.cs.umt.edu: dasl                                         /* a vita */
righter@wru.umt.edu: eis                                              /* a vita */

```

Figure 6: An example *group\_info* instance for cell A



Group Name	Permissions
owner<trish@radiator.cs.umt.edu>	rd-m-ex
eis	r---ex
world	r---

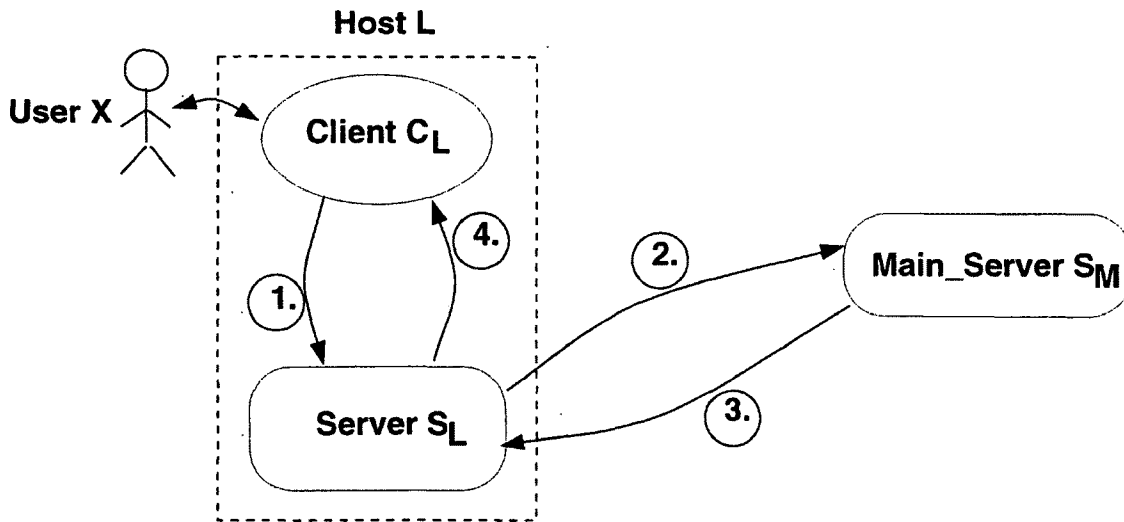
Figure 7: *Priv.info* for hierarchy *habitat\_type*

erarchy named *habitat\_type* on host “radiator.cs.umt.edu”. Then “trish@radiator.cs.umt.edu” becomes the owner of hierarchy “habitat\_type”. Either the owner or the EIS\_ADMIN can specify access permissions to “habitat\_type” for different groups. An example *priv.info* for “habitat\_type” is as shown in Figure 7.

### 3.4 EIS Database Organization

In this section, we discuss scenarios that illustrate the client/server (object) interaction used to implement the basic services provided in EIS. These scenarios are important because they help to illustrate the database organization and reflect upon the way the groupware authentication is designed.

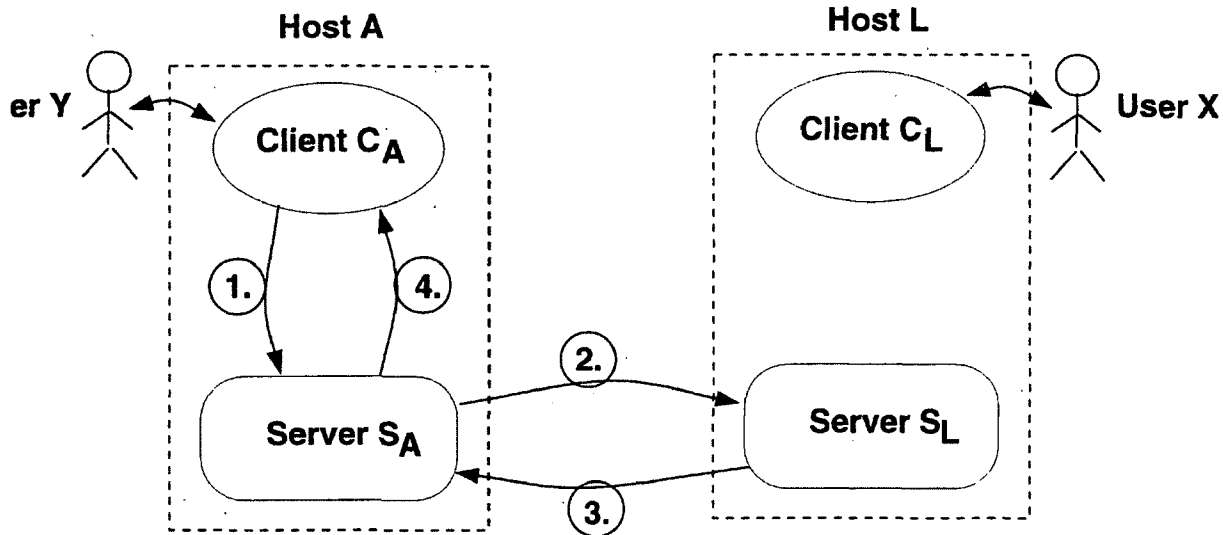
Consider some cell C, in which the *main\_server* is  $S_M$ . The *main\_server*  $S_M$  contains a “hierarchy list” which maintains a list of all hierarchies created within the cell domain. For every hierarchy that is created, there is a *min\_info* object that maintains links to all the classes, instances and methods that together form the hierarchy. The *min\_info* is encapsulated in the database on the host where the hierarchy was originally created. That is when a new hierarchy “ $H_1$ ” is created, its *min\_info* “ $H_{1:min}$ ” is stored on the local host’s database and a link to the hierarchy’s *min\_info* “ $H_{1:min}$ ” is planted in the “hierarchy list” maintained on the *main\_server* for the cell. Individual



1. Save *min\_info* for hierarchy  $H_1$
2. Server  $S_L$  records the *min\_info* for  $H_1$  in its database
3. Server  $S_L$  requests  $S_M$  to add a link to  $H_1$  in the hierarchy list
4. & 5. return ACK.

Figure 8: Create and Save a Hierarchy  $H_1$

class, instance and method definitions that are subsequently added to the hierarchy are encapsulated on the host where they are created. Everytime a class, instance or method is defined that extends a hierarchy, the object's description (called *max\_info* for the object) is stored on the local database. A link to the object's *max\_info* along with some minimal information about the relationship of this object with other objects in the hierarchy is planted in the hierarchy's *min\_info* " $H_{1:min}$ ". Thus whenever a new object is defined on host L for a hierarchy originally created on host K, the object definition is stored on L and a link to this definition is created in the hierarchy's *min\_info* object on host K.



1. Store the *max\_info* for the new class on the local database.
2. Add a link to the *min\_info* of  $H_1$  on host L's database.
3. & 4. return ACK.

Figure 9: Extend a Subclass to hierarchy  $H_1$

Figure 8 depicts the scenario where user X on host L creates a hierarchy  $H_1$ . The client  $C_L$  makes a request to its local server  $S_L$  to save the *min\_info* for  $H_1$  on host L. The server also connects to the cell main server,  $S_M$ , to update the cell hierarchy list to add a link to the new hierarchy  $H_1$ .

Figure 9 describes a scenario where another user Y on some other host A adds a subclass to the hierarchy  $H_1$ . At the time of executing this service request, the user would have already loaded the *min\_info* for  $H_1$ . When the user adds the new subclass, a request is sent to  $S_A$  to add the *max\_info* for the extended class on  $S_A$ 's database. Subsequently,  $S_A$  sends a request "add a link to the new class" to the server " $S_L$ ", which will update the *min\_info* for  $H_1$ .

## 4. The 3-A Aspects of the Groupware Layer

### 4.1 User Authentication

Authentication is a process of verifying a user's identity. Authentication is the foundation of groupware security. Only if we can be sure of who we are talking to can any other security features be of any value. Authentication is the groupware mechanism most different from mechanisms used in monolithic systems, and the most complex part of the groupware layer. Hence a careful study of the authentication process in the EIS groupware is essential. The problem we need to solve is to guarantee that the EIS design and implementation satisfies following conditions:

- No activity by a rogue process (user/server/main server) should corrupt the database of any EIS server.
- No activity by a rogue process should plant a link in a valid database that when subsequently accessed allows the rogue to obtain rights or privileges on the requestor, or to corrupt data on the requestor.

A couple of popular solutions possible in the UNIX client/server domain are explained here. We shall later discuss the viable alternatives for authentication in EIS in section 4.2. In a network environment, it is difficult to determine the exact identity of a user on a remote host. In most client/server implementations of processes such as "rlogin" or "rsh", the client and server code on a host are owned by the root user of the host. The "rlogin" command logs a user into a specific remote host and attempts to connect the user initiating the rlogin request to the remote host. The "rsh" command attempts to execute the specified command on the remote host on behalf of the user initiating the request. Both "rlogin" and "rsh" are client/server implementations where the server runs as a daemon process waiting for client requests and the client is executed whenever a user issues the command. A user on a host can tell the rlogin and rsh daemons to allow some users log into their accounts directly without prompting them for a password while preventing all other users from

logging in without entering the correct password. The client code can be executed by any normal user. A user executing this code temporarily becomes “root”, and thus temporarily gains privileges allowing the execution of root operations like reading kernel files or binding the client to a reserved socket port. However it is important to note that it is the process executing this code that gains “root” privileges though it is executed by any normal user. Hence as long as the integrity of the executable allowing temporary root privileges is guaranteed, we can ensure that a normal user cannot do anything rogueish.

The authentication mechanism explained here pertains to the implementation used in standard Unix Network software such as “rlogin”. A detailed study of the implementation of Unix Network Programming is discussed in [8]. When a socket connection is established by the client with a remote server, the server can request the port and machine address of the live connection<sup>1</sup>. Since only root processes can bind to a reserved socket port, the server process can verify if the connection is established with a client process that has root privileges on a trusted host. A rogue can modify the client code and try to do something rogish. However if the rogue is not root, the rogue’s code will not have root privileges and hence will not be able to bind to a reserved socket port. In such a protocol, there is a potential problem since the server must trust the root on a remote site. This is quite acceptable since the client can do nothing to harm the server host as long as the client is not granted the unusual privileges by the server (in the rlogin example, the “.rhosts” file determines a remote client’s privileges).

The authentication mechanism explained in the above paragraph cannot be adopted into EIS for the following reasons. Since EIS is still in a developmental stage, it is arguably better not to install the EIS client and server code as root processes with full root privileges (most often, the servers are the entry-points for Internet rogues.).

---

<sup>1</sup>The Unix system call `getpeername()` returns the foreign machine’s IP address and the port number to which the foreign process is connected

Since the source code for EIS is to be freely distributed, trusting the client process could be disastrous. An alternative would be to use the information provided by the **ident** facility to identify the owner of each client process requesting server access. As per the RFC 931 protocol, **ident** provides a reasonably good way of authenticating a user. It requires that both machines engaged in the client/server connection must run the identity daemon **identd** process in the background. The server upon receiving requests from a client, then connects to the **identd** daemon on the host where the client process is running to verify the user's identity.

Unfortunately there are problems with **ident** which make it unacceptable for use in authentication in EIS. Firstly, the **identd** daemon has to be a root process, since it reads kernel files (`/dev/kmem`) to determine which user is connected to a given port. As such it should be started by the **inetd** daemon. Secondly, there is a significant amount of communication overhead (`end_server`  $\leftrightarrow$  **identd** daemon) needed just for authentication purposes. Also while **identd** works to authenticate local/remote requests, it does not help local host authentication in any way. Finally, use of **ident** is gaining popularity, but it is still not an accepted standard and hence not all systems support it yet. To simulate **identd** functionality within the EIS server process is also not possible since the process would need to read kernel files which are readable only by the root.

## 4.2 Authentication in EIS

As depicted in Figure 1, an EIS server interacts directly with either clients on the same host or another server on a remote host which issues requests on behalf of some client on that host. Given such a situation, the problem is to identify appropriate methods to verify the trustworthiness of a remote server and a local client so as to ensure the validity of the conditions described above.

### 4.2.1 Local Client Authentication

In EIS, a client directly connects to the server on its host for every request it needs processed. As long as we assume that the server on that host is trustworthy, this helps us in the authentication process. With every request generated by the client that needs authentication, the EIS client determines the user's identity and passes it to the local server, which verifies the validity of the user information by looking in the local process table. Since the client/server connection pertains to the same host, external "spoofing" can be ruled out. This authentication method of a local client process helps identify exactly the user issuing the client request and hence satisfies the conditions set forth above in this section.

### 4.2.2 Remote Server Authentication

Remote server authentication is more difficult to attain in the EIS system because of the following reasons:

1. Neither the SUN RPC nor the socket system calls provide any means of identifying the caller user's identity. There are provisions to determine the remote host's IP address and the port to which the caller is connected. However it is not possible to determine who the calling user is.
2. EIS 0.9 should not be run as a root process, and hence cannot take advantage authentication mechanisms that require root access.
3. The ultimate goal is to use a DCE-based implementation to provide a reliable security mechanism. EIS should not be based on elaborate ad hoc solutions in the interim.

Since our main aim is to ensure that no operation by any user of EIS 1.0 corrupts the database in any way, the following approach is suggested:

- All EIS administrators must be valid users on the *main\_server* of the cell. We restrict certain EIS admin operations, so that all admin-related operations (**create/remove a group, add/delete users from a group**) must be requested from the *main\_server*.
- Any user who is authorized with **read, extend/execute** privileges on a hierarchy can read the hierarchy. In order to *delete/modify* a subclass/instance/method, the user must not only have the right privileges, but must also execute the *delete/modify* commands from the same host on which the objects were created (i.e. the user has to be “local” to that host).
- No user from some host can effect a *delete/modify* operation on an object created on some other host, even though the user belongs to a group that is authorized with delete/modify privileges.
- For all other services requested by a remote user, EIS will normally trust the authentication information received from the remote host and cross-check only to verify the host IP address of the connecting client.

As explained in section 4.2.1, we can authenticate with certainty any user on the local host. However, when a request is made across the network, the system is more vulnerable to external spoofing and the different alternatives discussed earlier fail to ensure a sure-proof authentication mechanism. The above approach takes the firm stance of limiting the access privileges of outside users and simply denies outside users the right to delete/modify classes/instances/methods, because such operations can affect the local database. Yet, this does not mean that we are over-protective of the database information. For most services available in EIS, we normally trust<sup>2</sup> the authentication information provided by the remote server. Any user with the appropriate access can request and use “read & extend” services normally. The

---

<sup>2</sup>We still check the socket connection to verify the IP address of the host the server is connected to.



extra precautions in the case of modify/delete services are used to ensure that only the “right” persons modify local databases. This approach also avoids extensive performance penalty for authentication, since there are no repeated calls back and forth just for authentication purposes.

With this authentication scheme, we control “write” access to the database. However, a rogue user can still extend objects to a hierarchy or an entire hierarchy to a system that had no business being there. Under normal circumstances, we can assume that everything runs ok and that an EIS user has no malicious intent. But there are several issues we need to consider more carefully.

1. An EIS Administrator controls EIS information on a cell (a group of hosts). Suppose some rogue user creates a hierarchy  $H_1$  on some host B in the cell. The administrator sitting on the *main\_server*'s host M is an external user to any information on host B and hence has limited access to the hierarchy  $H_1$  on host B. We noted in section 3.4 that the *main\_server* maintains links to all the hierarchies (called hierarchy list) created within the cell it encapsulates. Any user who wishes to open/use a hierarchy has to first retrieve this hierarchy list and traverse the link to the appropriate hierarchy. As a solution to the above problem, the Administrator can choose to remove the link to the hierarchy  $H_1$  from the hierarchy list and control the user's access rights to the cell.

2. Another problem occurs when a rogue user extends objects to a hierarchy created by some other user. The owner of the hierarchy has the access rights to decide who can extend objects to the hierarchy and who cannot. But as in the previous case, if an object is already extended by a rogue user on some other host, the owner of the hierarchy cannot remove these objects physically from the database since now he is not a trusted user on the other host. Again, the solution in this case is similar to the one discussed for the previous problem. With every hierarchy, we maintain an object called *min\_info* which resides on the database of the host where

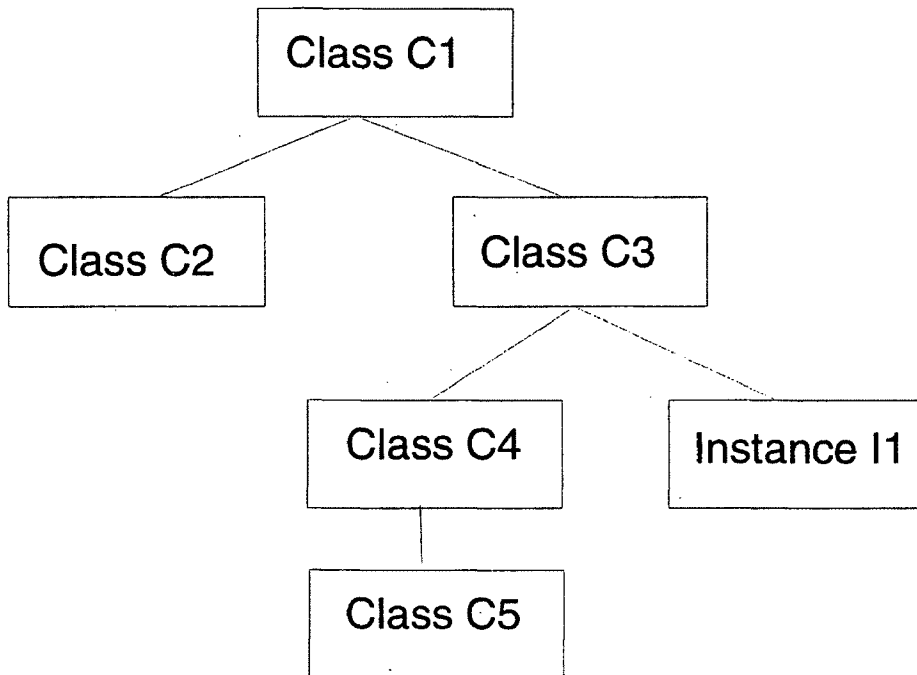
the hierarchy was created. The *min\_info* object maintains links to all the different classes/instances/methods extended to the hierarchy, along with some minimal information of how the classes/instances/methods relate to one another in the hierarchy. Everytime someone extends a class/instance/method to a hierarchy, the information the object carries is stored on the local database of the host where the object was created and a link to the object information is placed in the *min\_info* object. The owner of the hierarchy controls the *min\_info* object though he has no control to the physical information of the extended object. He could remove the link to the extended object. This could get complicated when a scenario given in Figure 10 develops. Classes C1, C2 and C3 are valid classes added to hierarchy  $H_2$ . Now a rogue user extends a class C4 and instance I1 to the class C3 as shown in the figure. If later some other user extends a class C5 to C4, then removing the link to C4 becomes non-trivial since it has some valid sub-class(es) C5 which should still continue to exist. Here the owner of the hierarchy can choose to either hide just C4 and I1 so that they are not accessible by any user, or remove the sub-tree starting at node C4 so that links to all objects in the sub-tree are lost, or move the sub-class C5 to some safe parent class and then remove the links to C4 and I1.

### 4.3 Authorization

As we explained earlier when defining the groupware classes and objects, the object that encapsulates the authorization process in EIS is *group\_info*. Authorization is a process of determining the different groups the user belongs to. Once the authentication procedure is done correctly, to authorize a user, we have to make a connection to the *main\_server* of the cell<sup>3</sup>. The *main\_server* requests its *group\_info* object to return a list of groups to which the user is a member.

---

<sup>3</sup>If a copy of the cell's *group\_info* already exists on this site, then this process is much simpler



Class C4 and Instance I1 exhibit 'rogue-ish' behaviour.

Solutions:

1. Hide the sub-tree rooted at Class C4.
2. Delete the sub-tree rooted at Class C4. Implication of 1 & 2: Valid class C5 is not accessible anymore.
3. Make Class C3 the parent of C5 and delete the sub-tree now rooted at Class C4.

Figure 10: An example rogueish behaviour

#### 4.4 Access Control

Access Control is a per-hierarchy operation. A group can have different access privileges to different hierarchies within a cell. Each hierarchy encapsulates the object (*priv\_info*) that keeps of the permissions for different groups. After determining the different groups for a user, we look up the *priv\_info*<sup>4</sup> object to verify each group's permissions to the hierarchy. The user's access rights are a union of the access rights of all the groups to which he belongs.

---

<sup>4</sup>The *priv\_info* object physically exists on the host where the hierarchy was created.

## 5. Scenarios

We shall use a scenario-based examination to see how the groupware layer fits into the EIS system. A scenario is a description of how objects interact with each other while performing a specific activity. A scenario can be represented in many different ways, but one of the most convenient representations is as a diagram showing objects and the exchange of operation calls/value returns. Such a diagram represents the client/server relationship in this particular scenario.

### 5.1 Notations

We shall adopt the following notational convention in describing the scenarios:

<i>Client</i> $C_{I1}$ :	The EIS client process 1 on host I
<i>Server</i> $S_I$ :	The EIS server process (one per host) on host I
$S_M$ :	The designated <i>main_server</i> for the concerned cell
$S_R$ :	A remote server in the cell from which a hierarchy is being retrieved.
$L$ :	The local host from where the user is operating.

The arrows in the scenario-based diagrams indicate that the object pointed to is being requested for a resource/action by the object from where the arrow is originating. We also denote the sequence of events by marking the arrows with numbers. For example, an event marked 3 occurs earlier than one marked 5. The exact action that occurs on the event is shown separately at the bottom of the diagram, referenced by the event number. The sections below discuss the finer points of the scenarios depicted in the figures.

## 5.2 Basic Design

In all the scenarios presented in this section, every request involving remote data access generated by a client on some host  $L$  is sent first to the server  $S_L$  on the same host, then subsequently from  $S_L$  to the remote server. There are a few advantages of passing requests through the local server  $S_L$  rather than having the client connect directly to the remote server. The first advantage is that the server on the same machine can verify if the client caller is really the user that he claims to be. This helps 'strengthen' the authentication process against users trying to spoof the system. Secondly, the local server can be designed to maintain a cache of remote *group\_info* and *priv\_info* objects that are frequently accessed. *Group\_info* and *priv\_info* are the objects most frequently accessed within the EIS system and such a cache could dramatically reduce network loading and on remote servers. Caching this information at the local server whenever necessary helps speed further accesses to these objects while at the same time not compromising on security by preventing the client direct access to this information.

Unfortunately, although caching *group\_info* and *priv\_info* at different servers where they are being used greatly helps in reducing network traffic, it also introduces some problems. The first and most important of these relates to authentication. By caching groupware information on a remote host, we are trusting a remote server to provide a valid authentication. This works as long as the remote server is trustworthy. But the remote server could also be a rogue that provides false information. The authentication scheme presented in section 4.2 tells not to trust a remote user or server when any service with *write* or *modify* privileges is requested. The authentication scheme restricts *write* and *modify* services to only privileged users on the local host. Thus we are not forced (or allowed) to trust remote servers.

Another problem relates to consistency and concurrency issues. Appropriate steps need to be taken in order to maintain consistency between the different copies of a

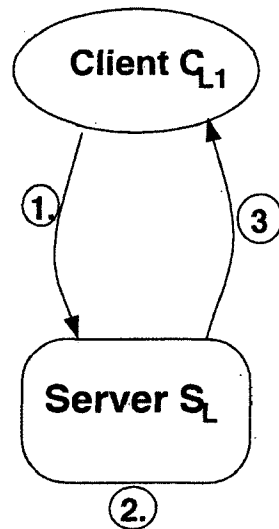
replicated object whenever the object is updated at any of the sites. The scenarios that follow deal with these issues and propose a method of dealing with the consistency issues.

### 5.2.1 Create a Group

A scenario for group creation is shown in Figure 11. We assume that the domain of this operation is restricted to the current cell under consideration. In order to perform this operation, the user has to be the EIS Administrator for the cell. As we discussed earlier in section 4.2, an EIS Administrator has to be a valid login account on the *main\_server* of the cell. This way, the authentication is fool-proof. The scenario indicates the order in which the authentication/authorization routines are performed. The EIS client  $C_{L1}$  identifies the user performing the action and sends the request to the server on its machine  $S_L$ . The server  $S_L$  verifies that the requestor is really the person he claims to be. This is the authentication process. An outline of the algorithm used to verify the user's identity is shown in Figure 1 in appendix A. The server checks to see if server  $S_L$  is the *main\_server* itself. Since the authentication verifies if the client  $C_{L1}$  was running on the same host L, this check helps determine if the client  $C_{L1}$  is running on the *main\_server*. The *main\_server* then looks up the *group\_info* object in its database to see if the user is an administrator. This is the authorization process. If the authorization process returns a positive acknowledgement, only then is the newly created group is effected on the *group\_info* object in the database.

### 5.2.2 Open a Hierarchy

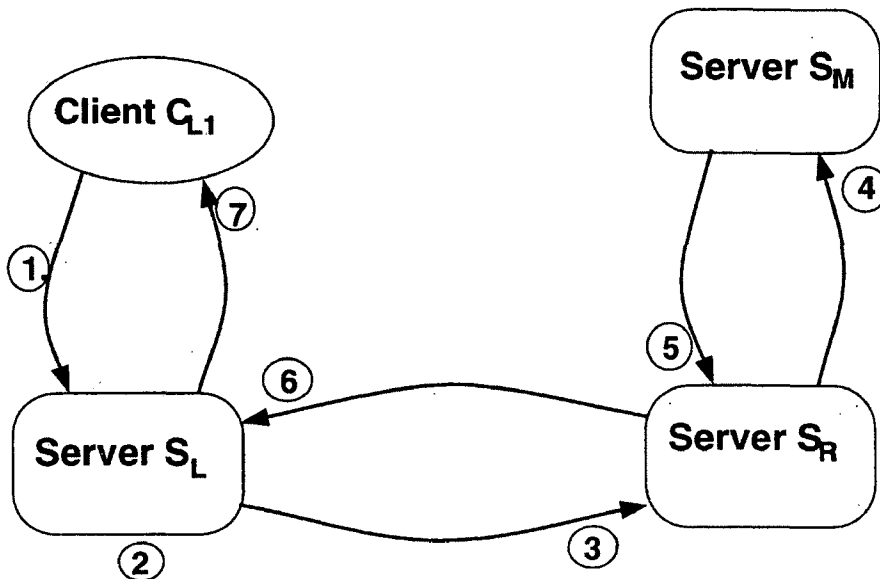
Figure 12 illustrates execution of the "Get.Hierarchy" operation.  $S_R$  is the server on the site where the hierarchy is stored. Server  $S_M$  is the *main\_server* of the cell including host R. The OME object in the client  $C_{L1}$  has knowledge of where the requested hierarchy exists on the network.  $C_{L1}$  sends a request to its local server  $S_L$



1. Create\_Group(myid, A)
2. if authenticate(myid) then /\* verify if the userid myid is local and valid \*/
  - if  $S_L = S_M$  then
    - if myid = admin and group A does not exist already then
      - begin
        - create\_group(A);
        - ack = OK;
        - return(ack);
      - end
        - ack = PERMISSION\_DENIED;
        - return(ack);
3. return(ack);

Figure 11: Scenario: Create a Group





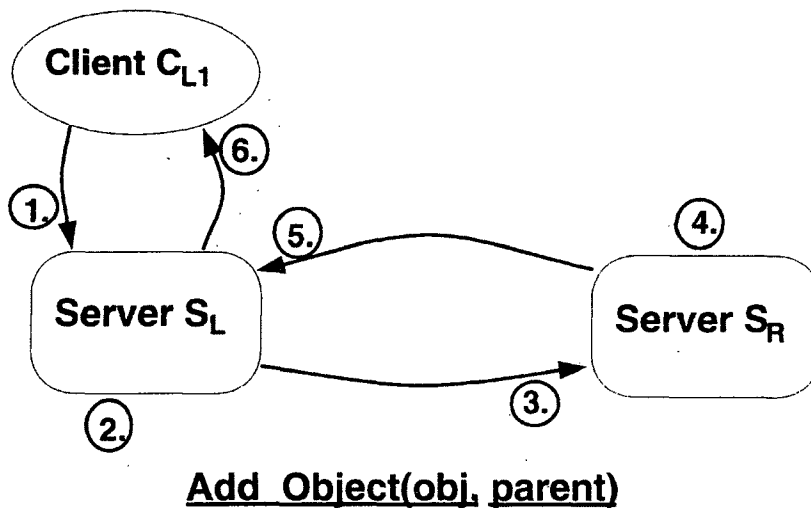
1. Get\_Hierarchy(uid, HIER, R)
2. if authenticate(uid) = OK then /\* verify if the userid myid is local and valid \*/
  - if group\_info and HIER.priv\_info copies exist in the cache then
    - if authorize(uid, READ) = OK then
      - /\* forward request to host R to retrieve hierarchy's min\_info \*/
 Get\_Hierarchy(uid, HIER, R);
      - else return PERMISSION\_DENIED;
    - else Get\_Hierarchy(uid, HIER, R);
3. if verify\_client\_host(uid.host\_address) = OK then
  - group\_info = get\_group\_info(M);
4. /\* Server S<sub>M</sub> notes that server S<sub>L</sub> will keep a copy of group\_info \*/
  - group\_info.site\_list.add(L);
  - return(group\_info);
5. if authorize(uid, READ) = OK then
  - begin
    - /\* Server S<sub>R</sub> notes that server S<sub>L</sub> will keep a copy of HIER.priv\_info \*/
 HIER.priv\_info.site\_list.add(L);
    - return(min\_info, group\_info, HIER.priv\_info);
  - end
    - else return PERMISSION\_DENIED;
6. HIER.priv\_info.site = R; /\* Server S<sub>L</sub> notes the site of the original copies \*/
  - group\_info.site = M;
  - copy group\_info and HIER.priv\_info on local database;
7. return(min\_info);

Figure 12: Scenario: Open a hierarchy 'HIER'

passing as arguments the requesting user's identification, the name of the hierarchy requested and the site from where to receive the hierarchy. The local server first executes the authentication algorithm to verify the user's identity. The local server then tries to determine if the groupware information for the hierarchy already exists in the cache. If a reference is found, the client is authorized and his access control verified at the *local\_server*. If the authorization/access\_control operations fail to validate the user, then a negative acknowledgement is sent back to the user immediately and the service request ignored. If however, the user passes the authorization test on the *local\_server* or there is no groupware information stored in the cache, the request is forwarded to the remote server  $S_R$ .  $S_R$  requests its *main\_server* to authorize the user (determine all the groups to which the user belongs).  $S_R$  then checks if any of these groups are privileged with *read* permissions on the hierarchy. If so, the appropriate information is returned to the user (in this case, the *min\_info* for the hierarchy). Even if the client passes the authentication and access control test on its local server from the groupware information stored in the cache, a cross-verification is done by the server on the hosts where the actual database exists. Also notice that the *group\_info* that originally existed on server  $S_M$  and a copy of *priv\_info* for the hierarchy HIER whose original copy exists on server  $S_R$  are copied on server  $S_L$ . The local server  $S_L$  keeps track of the respective sites from where the copies were obtained. Also, the sites that store the original copies note down that server  $S_L$  now has a copy of their data.

### 5.2.3 Add an Object to the Hierarchy

This scenario continues the scenario shown in Figure 12. Having opened a hierarchy, the user now tries to add an object to the hierarchy. The user is first checked for *extend* privileges to the hierarchy. The scenario in Figure 13 explains how caching the protection-related information helps in improving system performance. The request for authentication is sent to the local server for authentication and authorization. If



1. `HIER.Add_Object(uid, Obj, ParentObj);`
2. `/* Authentication performed on group_info and HIER.priv_info in the cache */`  
`if authenticate(uid) != OK then`  
`return(PERMISSION_DENIED);`  
`store_max_info(Obj); /* Add Obj's max_info to local database */`
3. `HIER.add_link(uid, Obj, ParentObj); /* Request SR to place a link to the new object */`
4. `if authorize(uid) != OK then`  
`return(PERMISSION_DENIED);`  
`HIER.add_link(Obj, ParentObj);`  
`return(OK);`
5. & 6. `return(ACK);`

Figure 13: Scenario: Add object 'Obj' to object 'ParentObj'

the user is denied the access rights to extend to the hierarchy, control is immediately returned to the calling process. It is in situations like these that a cache greatly helps in reducing network traffic. However, caching of information could leave holes in the authentication process. Hence a more careful approach is adopted. Suppose in this case, the authentication performed based on the information in the cache goes through successfully, then the newly extended object is added to the local database. This does not lead to any penalty on the remote database<sup>5</sup> encapsulated by  $S_R$  in terms of disk space. However, for every object in the hierarchy, a link has to be placed in the hierarchy's *min\_info* on server  $S_R$ . In order to cross-verify that the server  $S_L$  was not spoofing, when a request to place a link to the newly extended object is sent to the server  $S_R$ , the authentication process is carried out again before placing the link. Though this authentication means extra CPU cycles on host R in most cases, it helps to make sure that the some remote rogue process is not placing a link in the hierarchy that does not need to be there.

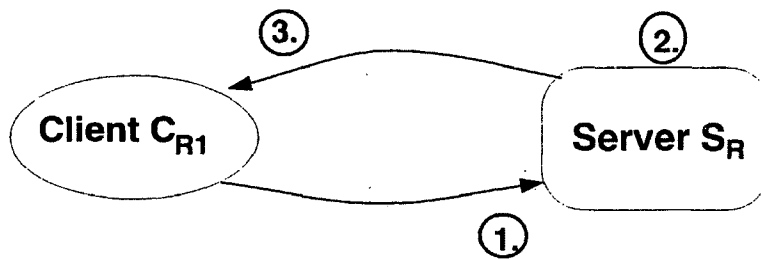
#### 5.2.4 Delete an Object from the Hierarchy

This scenario is slightly different from the “Add Object” scenario because of the way authentication is done differently for *delete* and *extend* services. Suppose an object “A” is created on some host R. i.e. the *max\_info* for the object “A” exists on the database on host R. The authentication mechanism restricts the delete privileges to only those users on the same host R that have *delete* privileges to the hierarchy. A user on some other host, say L, **cannot** delete (or modify) the object “A” even though they could belong to a group that has the required privileges to the hierarchy.

For example, let a group  $G_1$  contain users “X@R” and “Y@L” where X and Y are valid user accounts on hosts R and L respectively. If group  $G_1$  has *delete* privileges to the hierarchy, then the object “A” can be deleted by user “X@R” because he is

---

<sup>5</sup>where the actual hierarchy exists



### HIER.Delete Object(uid, A)

1. HIER.Delete\_Object(uid, A); /\* send request to local server \*/
2. **if** authenticate(uid) = OK **then**  
     /\* check if the request is coming from a client on the same host \*/  
     **if** C<sub>R1</sub>.host\_address = S<sub>R</sub>.host\_address **then**  
         **begin**  
             HIER.delete\_object(A);  
             **return**(OK);  
         **end**  
     **return**(PERMISSION\_DENIED);
3. **return**(ACK);

Figure 14: Scenario: Delete Object "A" from hierarchy "HIER"

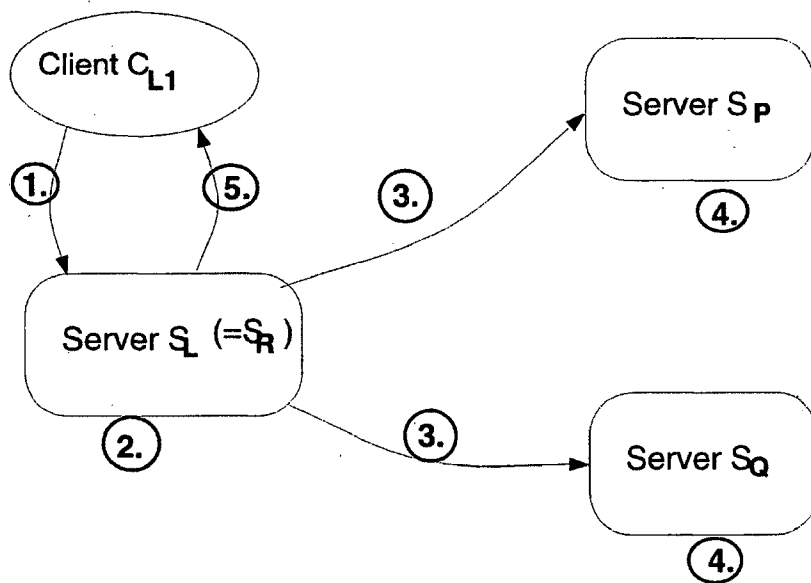
a privileged user on the same host R where the *max\_info* of the object exists. User “Y@L” cannot delete the object “A” even though he is a privileged user because he is a remote user with respect to the object “A”.

### 5.2.5 Add a group to a hierarchy

Groups for a hierarchy are encapsulated in a one-per-hierarchy object called *priv\_info*. A group can be added to or deleted from a hierarchy only by the owner of the hierarchy. Every group for a hierarchy has a set of privileges to the hierarchy. Refer to Figure 7 for an example *priv\_info* object for a hierarchy. In order to add a group to a hierarchy, the group must already be defined in the cell domain. The example *priv\_info* in Figure 7 is for a hierarchy “habitat\_type” that is encapsulated within the cell “A”. The groups defined on the cell “A” as given in Figure 6 are “admin”, “dasl” and “eis”. If the owner of the hierarchy “habitat\_type” tries to add a group “wsal” to the hierarchy, then the operation should fail since the group “wsal” is undefined.

Assume that the scenario in Figure 15 occurs when some user on host P and some user on host Q have obtained a copy of the hierarchy “HIER” under consideration. Then a copy of the *group\_info* and *priv\_info* for “HIER” reside on the databases encapsulated by  $S_P$  and  $S_Q$ . So whenever an update operation is done on any of these copies (including the original copy), the update operation has to be effected on all other copies of *group\_info* and *priv\_info* for “HIER” in order to maintain consistency.

In this scenario, the client  $C_{L1}$  directly requests the local server  $S_L$  for the concerned service. In chapter 3, we saw that the *min\_info* for a hierarchy resides on the database of the host where it was created. So the owner of the hierarchy should be a valid login account on the host where the *min\_info* for the hierarchy is stored. Following this logic, the server  $S_L$  makes a check to see if the server  $S_L$  is the same as  $S_R$ , the site of the database for the hierarchy. If not, the user is denied access to the



1. Add\_Group\_to\_Hier(uid,  $G_1$ , HIER); /\* client request to local server \*/
2. if authenticate(uid) = OK then
  - /\* check if request is from a local client and if the server  $S_L$  is the site of HIER \*/
  - if  $C_{L1}.host\_address = S_L.host\_address$  and  $S_L = S_R$  then
    - /\* add group to HIER.priv\_info in the database with the appropriate permissions \*/
    - HIER.add\_group\_to\_hier( $G_1$ ,  $G_1.perms$ );
    - else return(PERMISSION\_DENIED);
    - else return(PERMISSION\_DENIED);
3. send update\_priv\_info(HIER, new\_priv\_info) request to servers  $S_P$  and  $S_Q$
4. HIER.priv\_info.update(new\_priv\_info); /\* update cache copies of HIER.priv\_info \*/
5. return(ACK);

Figure 15: Scenario: Add group " $G_1$ " to hierarchy "HIER"

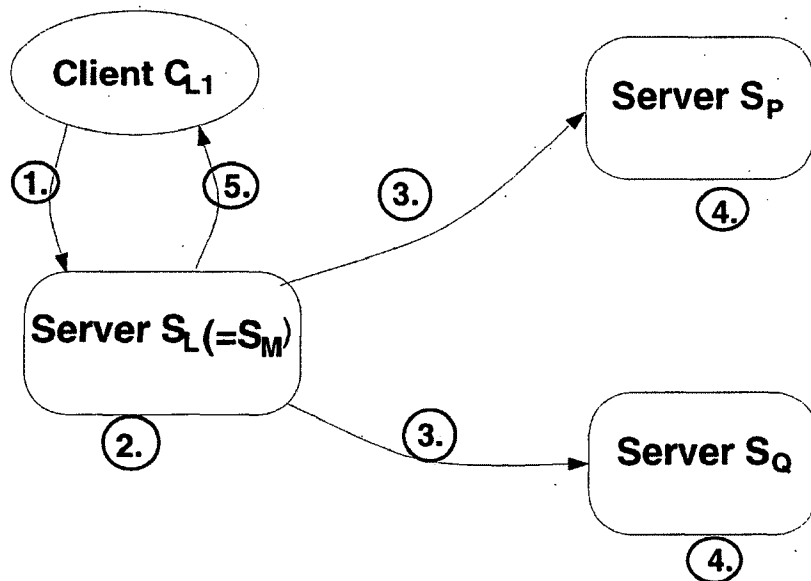
“add\_group\_to\_hier” service to the hierarchy. It is possible that a rogue process on some host directly connect to  $S_R$  rather than to the local host. A Unix daemon process does not treat requests from local processes differently from those from remote processes. The *getpeername* system call however helps identify the calling process’s host address. By making sure that the calling process has the same host address as the server process, we can thus ensure that the calling process is local to the host.

On successful authentication, the group  $G_1$  is added to the hierarchy’s *priv\_info* on server  $S_L$  (In this case, the local server  $S_L$  is the server  $S_R$  on which the hierarchy’s *min\_info* exists). Then an update request is sent to each server ( $S_P$  and  $S_Q$ ) that keep a copy of the *priv\_info* object. It is important to note that the copies of *group\_info* and *priv\_info* are *writable* only by the server that owns the original copy and not by any other. This is because only the owner of a hierarchy has write access to the *priv\_info* object for the hierarchy and only the EIS administrator has write access to the *group\_info* object for a cell. The local server that caches this information has only read access to the copy it owns. Any write operation on these objects performed on a copy does not affect the original copy.

### 5.2.6 Add a user to a group

This scenario is conceptually similar to the “add\_group\_to\_hier” scenario described in section 5.2.5. The main difference between these two scenarios is that this scenario is a *group\_info* related operation while the previous scenario was a *priv\_info* related scenario. And since the *group\_info* resides on the *main\_server*, and this operation requests the *write* privilege to the *group\_info* object, the requests to this operation has to originate on the *main\_server* and the EIS Administrator for the cell is the only authorized user to this operation. The approach followed in the scenario is otherwise similar to what has been described earlier. Note again that the cache copies of *group\_info* that exist on different hosts are *writable* only via a request from the *main\_server* of the cell. All other processes (the local client processes), use this





1. Add\_User\_to\_Group(uid,  $U_1$ ,  $G_1$ ); /\* client request to local server \*/
2. if authenticate(uid) = OK then
  - /\* check if request is from a local client and if the local server  $S_L$  is also the *main\_server*  $S_M$  \*/
  - if  $C_{L1}.host\_address = S_L.host\_address$  and  $S_L = S_M$  then
    - /\* add user  $U_1$  to group  $G_1$  in the *group\_info* object of the cell \*/
    - group\_info.add\_user\_to\_group( $U_1$ ,  $G_1$ );
    - else return(PERMISSION\_DENIED);
    - else return(PERMISSION\_DENIED);
3. send update\_group\_info(new\_group\_info) request to servers  $S_P$  and  $S_Q$ ;
4. group\_info.update(new\_group\_info); /\* update cache copies of group\_info \*/
5. return(ACK);

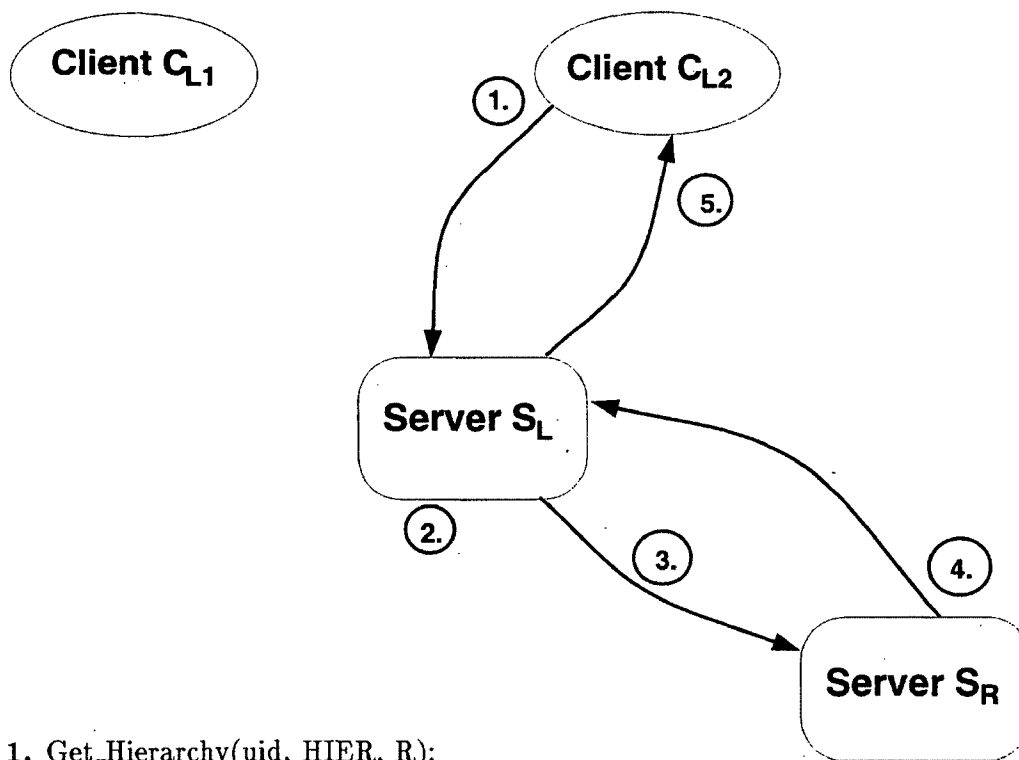
Figure 16: Scenario: Add user “a” to group A

cache copy for read purposes only.

It is not difficult for some rogue process to gain write privileges to cache copies. After all, cache copies of *group\_info* and *priv\_info* on a host are owned by the server process on the host. By manipulating the server code, one could try to gain write privileges on these cache copies. This however should not affect the integrity of the original copy nor should it weaken the authentication/authorization process. The *group\_info* and *priv\_info* objects in the cache are used to speed up authentication and authorization. If a user is validated by the information in the cache, before the actual operation is effected, a cross-check is done by validating the user against the original copies of the *group\_info* and *priv\_info* objects. This works fine and does not add to network congestion since most operations have to make remote requests to effect a service. Authentication/authorization can be done during this connection with the remote server.

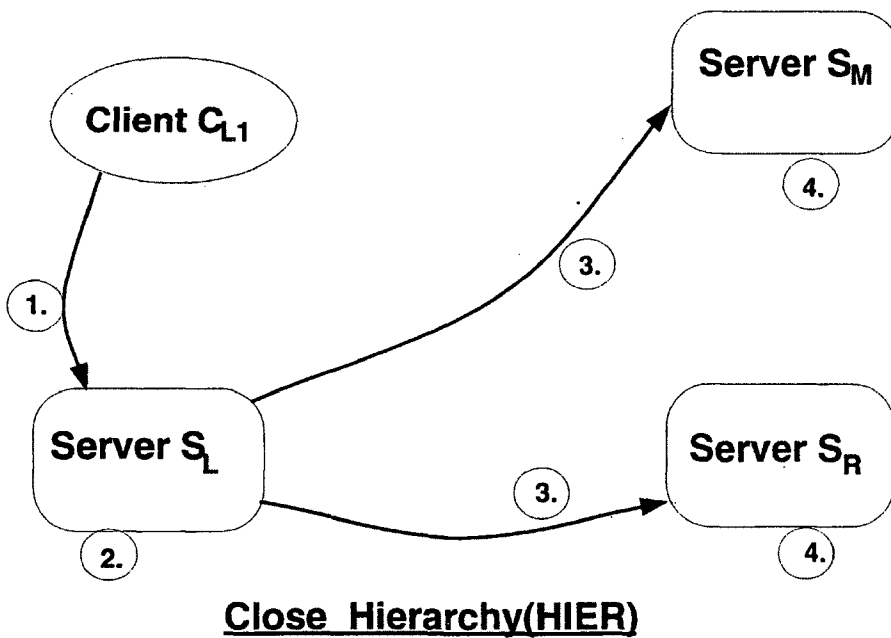
### 5.2.7 Multiple Clients on the same Host

Consider the case where a hierarchy is opened and being used by a user on some host L and another user on the same host now issues a request to open the same hierarchy. Here, we can take advantage of the existence of the protection related information for the hierarchy on the local cache, rather than having to connect to the remote hierarchy for each new caller. As soon as a request to open a hierarchy reaches the local server, it checks to see if the protection information already exists in the cache. If it does, then the authentication/authorization need be done only on this host. On the event of a successful authorization, the local server has to connect to the remote database site to retrieve the hierarchy information. Note that we do not cache the hierarchy's *min\_info* since there is no significant performance improvement in doing so. However, every time any EIS service is requested, we go through a process of authenticating/authorizing the user. Since *group\_info* and *priv\_info* are the objects encapsulating this process, we find it beneficial to cache these objects.



1. `Get_Hierarchy(uid, HIER, R);`
2. `if authenticate(uid) = OK then`  
*/\* Note down user-id and process id of all clients using HIER. \*/*  
`HIER.ref_list.add(uid, pid);`
3. `Get_min_info(HIER);`
4. `if authorize(uid) = OK then`  
`return(min_info);`
5. `return(min_info);`

Figure 17: Scenario: Multiple Clients on the same Host



1. Close\_Hierarchy(HIER);
2. delete HIER.priv\_info;  
delete group\_info;
3. delete\_site\_from\_list(L); /\* asynchronous request : no reply \*/
4.  $S_M$ : group\_info.site\_list.delete(L);  
 $S_R$ : HIER.priv\_info.site\_list.delete(L);

Figure 18: Scenario: Close Hierarchy

### 5.2.8 Close Hierarchy

It is necessary and important to keep data in the cache only for as long as it is required. As long as an object is cached, every update operation on that object will have to be passed on to all the copies that exist out there on the network so as to maintain consistency among the copies. So when a user chooses to close an open hierarchy, we have to do a cache cleanup operation. The “close\_hierarchy” request is forwarded<sup>6</sup> to the local server  $S_L$ . The local server deletes its copy of *group\_info* and *priv\_info* for the hierarchy and issues asynchronous requests to servers  $S_M$  and  $S_R$  telling them that the copies of *group\_info* and *priv\_info* respectively are deleted. Servers  $S_M$  and  $S_R$  then remove the host entry L from the site list that keeps track of the hosts maintaining a copy of the concerned information. However, in the event of multiple clients using the same hierarchy and hence the same data in the cache, it becomes necessary that we keep some sort of reference count to check the number of active clients referencing the cache information. In order to deal with this, the server is made to maintain a reference list to an opened hierarchy. The reference list keeps track of the process-ids of the processes that have opened the hierarchy along with a latest-use time-stamp. The time-stamp helps identify the last time the process made a service request for the hierarchy. The time-stamp also helps recover from situations where the client process dies abnormally.

### 5.3 Consistency

Consistency is one of the key issues in a system where replication of data is being done. In EIS, the replicated information in the cache is writeable only by one process. The *group\_info* cache object is writeable only by the *main\_server* process in the cell domain and the *priv\_info* cache object for a hierarchy is writeable only by the server

---

<sup>6</sup>The client need not wait for the return value of this request. It just closes the hierarchy window and issues an asynchronous one-way request to  $S_L$

where the hierarchy's *min\_info* is stored. All clients on the local host can use these cache objects in read-only mode. These factors make consistency among these objects easier to implement. In this section, we shall verify consistency of a replicated object and that of a requested service by looking at some scenarios with the read/read, read/write issues into consideration. The write/write situation never occurs since the replicated objects are writeable only by the server that owns the original copy of the object.

In the read/read situation, we can think of two processes reading from its local copy of the object. This does not affect the integrity of the replicated objects and hence nothing need be done to ensure consistency.

Let us consider a read/write situation. This is a situation when an update operation is effected by some process on an object A and another process on some other host has already read its local copy of the object,  $A^L$ , to perform some service before the *update object* request reaches object  $A^L$ . In section 5.2.5 and 5.2.6, we discussed some example scenarios where an update operation needs to be propagated to all those servers where a cache copy of the concerned object exists. Let us look at another scenario, "delete\_user\_from\_group", whose object diagram is exactly similar to that in the scenario in section 5.2.6. Let's suppose that the user X@L, who belongs to group  $G_1$ , has the all the privileges that reside with  $G_1$ . The group  $G_1$  is assumed to have *delete* rights to the hierarchy HIER. In this scenario, the "delete\_user\_from\_group" request can be issued only by the EIS Administrator and hence the *update group\_info* request originates on the *main\_server* of the cell. If the user X@L now requests the "delete\_object" service to an object in HIER before the "update group\_info" request reaches the replicated object  $group\_info^L$ , then we have a consistency problem in the requested service. The user X@L passes the authentication/authorization test on the local copy of group\_info,  $group\_info^L$ , even though he actually no longer is a user to the group  $G_1$  and hence cannot *delete* an object in the hierarchy. However, if we look back at the authentication mechanism used in EIS, we find that in the

event of successful authentication of a user against a local copy, a service requested is granted to the user only after authenticating him against the original copy of the groupware information. Hence in this scenario, the “delete\_object” request has to be propagated to the *main\_server* for authentication before the service is granted, thus ensuring consistency of the service.

We argued that a write/write situation never occurs. While this is true under normal circumstances, it is not impossible for a rogue to manipulate the replicated object on his host. In such a scenario, even though the consistency between the different copies of an object cannot be guaranteed, the validity of the original object and the services provided by another server are not compromised.

#### 5.4 Client crashes

As noted in the “Close Hierarchy” scenario, the life of data in a cache is determined by the existence of clients using that information. The reference-list that maintains the process ids of all the client processes using a cache object helps determine the processes still using a cache object. As soon as a client closes an open hierarchy, the process entry is removed from the reference-list. The data in the cache is physically removed when the last process entry is removed from the reference list. However, if a client process dies before closing an open hierarchy, then there is a reference to the process in the reference-list that will never be removed and the cache object would continue to live forever leading to unnecessary network communication overhead whenever the cache needs to be updated. To solve this problem, the server that keeps a local copy is made to periodically look up the process table to find if every client process in the reference-list is still alive. If any client process was found to be no longer in the process table, then its entry is deleted from the reference list. This ensures that the copy exists in the cache only as long as it needs to be there.

## 6. Summary and Conclusion

In Chapter 1, we discussed the need for a security mechanism for EIS that would provide a reliable authentication scheme and guarantee the coherence of the database against rogueish behaviour. Chapter 3 presents the design of the groupware layer for EIS. The groupware layer was designed using the object-oriented methodology. The design of a groupware layer for EIS was considerably different from that for a monolithic system. A detailed study of the organization of the different objects relevant to the groupware layer and how they fit into EIS were discussed in Chapter 3. When designing any network security mechanism, the most important concern is the way authentication is done. Only if we are able to identify correctly to whom we are talking, can we determine what access privileges that user can have. Chapter 4 deals with a careful study of various popular authentication mechanisms such as that used in “rlogin” and “rsh”, ident and kerberos and presents a viable authentication mechanism for EIS. Chapter 4 also discusses the authorization and access control aspects of the groupware layer. Based on the design of the groupware layer presented in Chapters 3 and 4, a series of scenarios were presented in Chapter 5. The scenarios were a means of viewing different design issues and helped greatly in designing the right approach for implementation in EIS.

### 6.1 Implementation Status

EIS 1.0 supports the groupware layer and is fully functional. The implementation follows the design discussed in this thesis. EIS is currently supported on IBM/RS6000 workstations and runs on AIX 3.2. The code is written in C++ and the client/server communication system is implemented using the SUN RPC 4.0. The code should be portable to a wide range of Unix workstations within the portability bounds of different C++ compilers, X/Motif library implementations and the SUN RPC. EIS is presently populated with enough database information to illustrate both its use and its value to potential users.



The *group\_info* object encapsulating the authentication information for users in a cell exists on the *main\_server* of the cell. Since this is by far the most frequently used object in EIS, we use a shared memory implementation of the object thereby keeping it in shared memory on the *main\_server* at all times. This helps speedier access to authentication and also eases coding the operations performed on the *group\_info* object. Also since there is only one *group\_info* object per cell, there are not too many shared memory segments kept floating in the machines.

## 6.2 Directions for Further Research

The implementation does not include the caching concept discussed in the scenarios presented in Chapter 5. The goal of the groupware design was to first design a security mechanism that was reliable and helped provide controlled access to the EIS database and then consider implementing features that improved the system performance. The present implementation provides all the functional features necessary in a security system. Migrating from the present system to one including the caching concept should not affect the end-user in any way and the process is upward compatible. EIS is still in a rudimentary stage with a limited user community. As the number of users using EIS increases, the design details presented here should form the basis for a consideration of the finer design issues. The future designer can pay more attention to the caching concept and come up with a reliable coherence scheme for implementation in EIS. Another feature that could be very helpful is to design appropriate replication algorithms to replicate the EIS databases on multiple systems. As the user community grows across the network, making far and remote requests for database resources could greatly reduce the performance. Replication algorithms help the end-user to connect to the nearest database server and hence speed up access times.

The present authentication mechanism does not use the more recent popular authentication schemes such as kerberos and public key authentication. The ultimate

goal is to be able to use a DCE-based implementation of EIS to provide a reliable and robust security mechanism. As DCE gains popularity and becomes available on multiple platforms more easily, a port of EIS to a DCE-based implementation is most desirable. Newer versions of SUN RPC also claim to provide a more robust security mechanism. Also they should be freely available on many platforms. An upgrade of the present system to one using the newer version of SUN RPC might also help improving the authentication in EIS.

EIS 1.0 does not support any kind of recovery from server crashes. If a server servicing requests to an EIS database crashes, then that database remains inaccessible until the server is restarted and starts functioning normally again. The future researcher could explore possibilities of implementing more graceful server recovery from crashes. One possible approach could be to keep a log of the current state of the server. Upon a server crash, the server could be initialized to the previous state as noted in the log and proceed from there. Another, more reliable approach, could be to consider replication algorithms to replicate the EIS database so that there are more than just one server providing services to a database. Such a system not only speeds up access times by allowing the user to connect to the nearest database server, but also provides fault-tolerance.

## Appendix A

### Pseudocode for Authenticating a User

```
authenticate(user X) /* begin authentication for create group */
begin
    getpeername(socket_descriptor, &remote_host_address, &remote_port)
    if (X.host_address != remote_host_address) then
        return(FALSE);
    proc_info_list = getprocinfo();
    while (proc_info != NULL)
    begin
        if proc_info.program_name = "eis" then
            if proc_info.user_id = X.user_id then
                return(TRUE);
            proc_info = proc_info_list.next;
        end
    return(FALSE);
end
```

Figure 19: Authenticate User X

# Appendix B

## Installation Guide for EIS

### Product Description

The Ecosystem Information System (EIS) is an object-oriented distributed system containing various types of information of interest to ecosystem modelers and managers. EIS 1.0 is presently supported on the IBM RS6000s. Successful installation of EIS 1.0 on your system will require:

AIX XL C++ 1.1 or AIX C++ Set 2.0

X/Motif

ONC RPC 4.0 library

EIS comes in two parts: EIS client and EIS Server. EIS client can be used by the user to gain access to the EIS repository. EIS server runs as a daemon process on each active EIS host.

EIS can be downloaded in two ways:

1. If your system is an IBM RS6000 and your machine does not support any one or more of the above mentioned libraries, then you can download the executables in binary mode. The three files to download are

eis	(The client executable)
eis_svc	(The server executable)
eis_install	(The server installation script)

2. If your system supports the above libraries, then download "eis.tar.Z" into the directory where you want to install this software and execute the following commands.

If you were in `SOME_DIR` directory when you executed the above commands, there will be an “eis” directory created under it. This will be your EIS base directory (henceforth named as `EISDIR`) i.e. `SOME_DIR/eis` is the same as `EISDIR`. The directory structure within `EISDIR` is shown later in this section.

After the `EISDIR` have been created, the next step is to build all the executables required for the EIS system. In order to do this, go to the `EISDIR` directory, and execute the command:

```
% make all
```

This will build the following executables

<code>EISDIR/d.clnt/eis</code>	(The client executable)
<code>EISDIR/d.server/eis_svc</code>	(The server executable)
<code>EISDIR/d.server/eis_install</code>	(The server installation script)

In order to use EIS on your host, you will have to first install the server as a daemon process. “eis\_install” is an installation script to install the server on your machine. The server on your machine has to know:

- 1) The path of the directory where the EIS database on this host exists. For example, you can specify `EISDIR/d.data` as the database path.
- 2) The EIS Main Server for this host. The Main Server is one that maintains a list of database sites and database hierarchies. An existing main server is: `meggars.cs.umt.edu`. You could choose to name your host as your main server in which case the database will have nothing to start with. However you can connect to any of the other sites through the client interface.
- 3) The EIS administrator for this site. [ This is necessary only if you choose your site as the main server]. The person installing EIS is automatically chosen as the EIS administrator.

Once you have made sure that the EIS Main Server and your local EIS Server are running, any user can access the database by running `EISDIR/d.clnt/eis`. It is advisable to include `EISDIR/d.clnt` in your `PATH` environment variable so that `eis` can be started just by typing “`eis`”.

## References

- [1] R.Ford, R.Righter, T.Duce, V.Hemige, D.Thompson, "A Network-based Object-Oriented Ecosystem Information System," *Decision Support 2001 - Resource Tech. '94 Symposium*, Toronto, Canada, Sept. 94.
- [2] R.Ford, R.Righter, T.Duce, V.Hemige, D.Thompson, "EIS: A Network-Accessible Repository for Ecosystem Modelers and Managers", *10th Annual ACM Symposium on Applied Computing*, Feb. 1995.
- [3] R.Ford, R.Righter, T.Duce, V.Hemige, D.Thompson, "A Network-Accessible Repository for GIS and Natural Resource Data", *GIS Symposium on Natural Resources*, Vancouver, Canada, Mar. 95.
- [4] Grady Booch, *Object Oriented Design with Applications*, The Benjamin/Cummings Publishing Company, Inc., 1992
- [5] John Bloomer, *Power Programming with RPC*, O'Reilly & Associates, Inc., 1992
- [6] Harold Lokhart, *OSF DCE: Guide to Developing Distributed Applications*, J.Ranade Workstation Series, 1994.
- [7] Andrew S. Tannenbaum, *Distributed Operating Systems*. Prentice Hall, 1995.
- [8] Richard Stevens, *Unix Network Programming*, Prentice Hall, 1993.
- [9] Peter Triantafillou, "The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems", *IEEE transactions on Software Engineering*, Vol. 21, No. 1, Jan. 1995, pp 1-17.
- [10] Robert Netzer, Jian Xu, "Necessary and Sufficient Conditions for Consistent Global Snapshots", *IEEE transactions on Parallel and Distributed Systems*, Vol. 6, No. 2, Feb. 1995, pp 165-169.