1-1-1996

An approach to building a secure and persistent distributed object management system

Yu Kin Ho
University of Nevada, Las Vegas

An Approach to Building a Secure and Persistent Distributed Object Management System

by

Yu Kin Ho

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

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May 1996

The Thesis of Yu Kin Ho for the degree of Master in Computer Science is approved.

Kia Makki April 10, 1996

Chairperson, Kia Makki, Ph.D.

John T. Minor

Examining Committee Member, John Minor, Ph.D.

April 10, 96

Examining Committee Member, Evangelos Yfantis, Ph.D.

Bahram Nassersharif April 10, 96

Graduate Faculty Representative, Bahram Nassersharif, Ph.D.

6-6-96

Dean of the Graduate College, Ronald Smith, Ph.D.

University of Nevada, Las Vegas

May 1996

ABSTRACT

The Common Object Request Broker Architecture (CORBA) proposed by the Object Management Group (OMG) is a widely accepted standard to provide a system level framework in design and implementation of distributed objects. The core of the Object Management Architecture (OMA) is an Object Request Broker (ORB), which provides transparency of object location, activation, and communications. However, the specification provided by the OMG is not sufficient. For instance, there is no security specifications when handling object requests through the ORBs. The lack of such a security service prevents the use of CORBA from handling sensitive data such as personal and corporate financial information.

In view of the above, this thesis identifies, explores, and provides an approach to handling secure objects in a distributed environment along with a persistent object service using the CORBA specification. The research specifically involves the design and implementation of a secured distributed object service. This object service requires a persistent service and object storage for storing and retrieving security specific information. To provide a secure distributed object environment, a secure object service using the specifications provided by the OMG has been designed and implemented. In addition, to preserve the persistence of secure information, an object service has been implemented to provide a persistent data store.

The secure object service can provide a framework for handling distributed object in applications requiring security clearance such as distributed banking, online stock tradings, internet shopping, geographic and medical information systems.

TABLE OF CONTENTS

List of Figures

ACKNOWLEDGEMENTS

It gives me great pleasure to acknowledge and thank those who have helped me overcome various obstacles in the course of my graduate studies.

If I were to name an ideal teacher and advisor it would be Dr. Kia Makki. He was always there and ready to help as my teacher and advisor. Despite his busy schedule, Dr. Kia Makki recommended the topic and has always made time to supply and discuss many ideas and to make many technical contributions. His excellent suggestions and criticisms were enormously helpful in the classification and development of this work. My discussions with him have been extremely rewarding because of his insight knowledge and quick grasp of the essentials of the problem. The ideas included in this thesis are the result of our intense discussions; it has always been a challenge to propose an idea and justify it to him. I also thank him for his immense patience in carefully reading, correcting and constructively criticizing with his thoughtful insights all the drafts of my thesis and providing enlighting comments. This thesis would not have been in its present form without his continuing support, suggestions, corrections, comments, and technical contributions. I am deeply indebted to him for personal, moral, and professional support and guidance in research and ethics.

I would also like to express my deepest gratitude and appreciation to Dr. Niki Pissinou from The Center for Advanced Computer Studies at the University of Southwestern Louisiana, for her early suggestions and valuable advice during the course of my thesis work and for her inspiring guidance and encouragement. Despite her busy schedule she has spent time to edit several drafts of this thesis. Her comments have been very helpful in making the technical presentation of this thesis clearer. Her suggestions

and criticisms on the technical aspects of the thesis have been invaluable. Her suggestions have had a great impact on this document. I am also very grateful to Dr. Pissinou's database research group, for their previous work in this area and providing me with a research context for the development of this thesis.

I also would like to thank other members of my thesis committee Dr. Bahram Nassersharif, Dr. John Minor and Dr. Evangelos Yfantis for their comments and serving on my thesis committee. I also like to thank the staffs in the National Supercomputing Center for Energy and Environment for their support and providing computer resources. I specially thank Ms. Justine Clarkin for proof reading the early draft of my thesis.

CHAPTER 1

INTRODUCTION

A distributed database is a collection of data that belongs logically to the same system but is physically spread over the sites of a computer network [17]. Design of distributed databases by this definition can increase the performance of database applications in two aspects: by reducing the amount of irrelevant data accessed by the applications and by reducing the amount of data transferred in processing the applications [22]. The object-oriented concept for representing real-world entities, on the other hand, can be added onto the distributed database management system to provide encapsulation, type and class hierarchies, inheritance, and operator polymorphism.

An object is one of the main forms of abstraction used in object-oriented programming methodologies. An object includes a "state" and "behavior". An instance of data is called a state. Operations that can be performed on the data are called behavior. An object includes both the state and the behavior. A class or type, is a specification of the object's state and behavior [3]. Classes form a hiearchy. A class below another class in the hiearchy inherits all the properties of the above class. This mechanism is called inheritance. These set of operations form the interface to that object. Ideally, the state or data held in an object should be accessed or modified using only the object's interface.

This property is called encapsulation.

Since 1987, a number of object-oriented database systems have emerged in the market. However, most of them have been in evaluation and preliminary prototype application development. None of them have been seriously used for mission-critical applications [23]. The current state of object technology is itself contributing to the problem via the diversity of object models. This is one of the aims of the Object Management Group (OMG) [40] to develop command models, a common interface for the development, and use of large-scale distributed applications using object-oriented technology [53]. The Common Object Request Broker Architecture (CORBA) developed by OMG is designed to allow integration of a wide variety of object systems. It defines the high-level framework in order to develop an object-oriented database system in a heterogeneous distributed environment.

Currently, database systems are designed to accommodate many users and very large sets of data; hence performance, security and authorization, transaction management, concurrency control, recovery, persistent storage, and dynamic schema changes become important issues. These issues have been stressed in the CORBA model. In particular, we focus more on the areas of persistence, and security and authorization. CORBA can be extended to incorporate a number of common object services which support basic functions for using and implementing objects. Each of the issues mentioned above can be solved by adding more object services. The Common Object Services specification released by OMG in March 1995 includes the persistent object service specification, but few CORBA based systems have incorporated the persistent storage of objects. However, the latest CORBA specification (2.0) still has not covered

how to handle the security and authorization. Research is continuing in an effort to incorporate security to the CORBA specification.

1.1 THESIS OBJECTIVES

The objective of this thesis is to design and implement a secure distributed object-oriented database management system along with a persistent object service using the CORBA specification. The object service should provide a mechanism to protect sensitive object data from being released to unauthorized personnel. A persistent object service is required to provide methods to preserve security information in a object-oriented data store.

In brief, we have five major objectives:

1. Evaluation of existing CORBA based implementations.
2. Evaluation of existing Object-Oriented Database Systems.
3. Evaluation of various security models and requirements.
4. Addition of persistence to a CORBA based implementation.
5. Design and implementation a security model to a CORBA based implementation.

For the first three objectives, we will examine some of the CORBA based implementation and Object-Oriented Database Storage systems on the market that were developed by industry and educational institutions. Then, we will implement the persistence object service based on the persistence object service specification released by OMG. Finally, we will design and implement our security model on a CORBA based implementation.

1.2 Overview of the Approach

As mentioned above, this research requires design and implementation of a secure object service. This thesis will provide a security model suitable to handle distributed objects. The model is based on a role-based environment. Each role is associated with a set of system privileges. A role contains authorization information to allow or to deny access to users. The model preserves the inheritance property of objects such that a role can inherit properties from other roles. An approach to operate and to maintain a role graph will be designed and implemented as an object service using CORBA specifications.

It is necessary to preserve the security information for the role-base model. In doing so, a persistent object service is required to store and retrieve role graph information in and out from an object storage. The persistent service hides the communication interfaces necessary to connect to an object-oriented data store.

The overall design consists of three main software components. They are: a role-based authorization interface, a persistent object service to an object store, and an event handler. The role-based authorization interface provides functions to perform security check on users and to maintain the role graph security information. The persistent object service allows permanent and secure storage for the security information. The event handler provides mechanism to allow the role-based authorization interface to perform security checks for object requests through the Object Request Brokers (ORBs). The ORB is the core of the object management system to identify, locate and handle object requests.

This thesis includes a detailed description of the design and implementation of

secure object service. It also includes an implementation of persistent object service in use with an object-oriented database system.

1.3 TARGET APPLICATION ENVIRONMENT

Secure distributed object management has become a hot issue with the increasing demands on internet services and distributed applications. Financial institutions such as banks and stock brokers start to provide on-line services to customers. The information requested can be complicated objects such as monthly financial reports with spreadsheets and graphical charts. The role-based model can provide security control for user authentication. This model is flexible and can be used to handle multimedia objects.

Another application is in handling distributed data of a corporation. Large corporations are trying to handle complex objects distributed among their various locations. The complex objects can incorporate financial data, employee information and business plans. The data are classified and security clearance is necessary for information request. The data can also be distributed into different database systems at remote locations. CORBA with the role-based security service introduced in this thesis can provide a standard framework with authentication control to implement distributed applications on these data.

1.4 SCOPE

Distributed object management covers a broad area in computer science. It covers computer network issues, distributed computing issues, object management issues and so on. This thesis addresses issues of providing a secure distributed object-oriented database

management system to handle sensitive data transfer using CORBA specification. The implementation will use ORBeline by Post Modern Computing as the CORBA implementation. ORBeline is a commercial CORBA implementation. It provides a general framework for developing CORBA compliant applications on distributed objects.

This work does not address all current issues related to distributed object-oriented database management. Rather, it addresses the critical problems existing in the widely accepted standard CORBA. The prototype introduced in this thesis is not sufficient to cover all the limitations of CORBA. Data encryption on transferring objects is another research topic. Other critical issues related to distributed object-oriented database management system include locking mechanisms, optimization methodologies and interoperability issues in ORBs.

1.5 Organization of the Thesis

The remaining chapters of this thesis are organized as follows: Chapter 2 is the research context and related works. This will include discussion on some CORBA based distributed objects management systems and some object-oriented database systems currently available on the market. The model of persistent storage will also be discussed in this chapter. In addition, we will evaluate several security models which could be used on our object-oriented database system. Chapter 3 will focus in detail on the design and implementation of a persistent object service in use with an object-oriented database system. Three protocols will be discussed for handling persistent objects. The three are the Direct Access Protocol, the Dynamic Data Object Protocol and the ODMG'93 Protocol [12]. The implementation of the persistent service is provided with description

of the CORBA Interface Definition Language (IDL). Implementation to persistent service for storing and retrieving objects in and out from an object-oriented database system will also be explained. Chapter 4 is the design and implementation of the role-based security model. Our abstract design model will be given in this chapter. Also, codes and explanation will be provided to implement the security service and to integrate it to CORBA. Chapter 5 will conclude our research work in this area and suggest some future extensions to our model.

CHAPTER 2

RESEARCH CONTEXT

Object-oriented technology emerged in late 1960s with the introduction of the SIMULA language. The object paradigm has been used in various areas since then. The most powerful capabilities from the object-oriented technology are class inheritance and encapsulation of knowledge. Today's database systems have been used not only in medium-size database systems, but also in very large database systems. Data handled by database systems no longer reside on single machines. They may be distributed over high-speed networks. Types of data stored in a database range from simple text data to multimedia data involving image, audio and video. The idea of data hiding resulted from the object relation can hide the complexity of today's database systems to the database developers, end-users, database administrators. Overloading, overriding, and late binding capabilities from object-oriented concepts can reduce significant amounts of time in application development [45]. However, the advancement from this technology on the area of distributed database development has been slowed down by the fact that no standardized framework exists for the developers. The Object Management Group (OMG) [40] was formed in 1989 with the purpose of creating standards allowing for the interoperability and portability of distributed object-oriented applications. A large group

of companies including the major players in the commercial distributed object-oriented computing arena have joined OMG for creation of a standard specification for a framework in distributed object-oriented database environments. The Common Object Request Broker Architecture released by OMG is a concrete description of the interfaces and services provided by the compliant core elements in the architecture [58]. But still, there are other issues that have not yet been handled by the CORBA specification such as security, one of the key elements in distributed object-oriented database system. The enforced security policies are directly affected by the based architecture.

This chapter will first briefly describe CORBA, then will evaluate some of the existing CORBA based systems on the market. Next, we will move our attention to some existing object-oriented database systems. Finally, several security models will be discussed.

2.1 DISTRIBUTED OBJECT TECHNOLOGIES

This section will discuss the current technologies available to handle distributed object. To begin with, the Common Object Request Broker Architecture (CORBA) by the Object Management Group (OMG) [40] will be presented. A number of CORBA based systems have become available on the market since the release of CORBA specification version 1.1 by OMG. There are also some CORBA like systems such as Xeros PARC's ILU [20] and Stratus/ISIS Reliable Distributed Objects (RDO). CORBA based systems are IBM's DSOM [54], Iona's Orbix [37], DEC's Object Broker [36], Expersoft's PowerBroker, ILOG's ILOG Broker, HP's ORBplus, PostModern Computing's ORBeline [38], Prism Technologies' OpenBase, and Sun Microsystems'

NEO™[55]. Of these CORBA based systems, three will be discussed; namely, IBM's SOM [54], Sun Microsystems' NEO [55] and PostModern Computing's ORBeline [38]. Their basic features, functionalities, and their compliance to CORBA specification will be evaluated.

2.1.1 CORBA

The Common Object Request Broker Architecture (CORBA) is designed to allow integration of a wide variety of object systems. It comprises of various key interfaces, services, and the core element, the Object Request Broker (ORB). The ORB is like a postal system and is basically responsible for routing client requests to objects. The objects can reside on a local computer system or a remote one. The client is unaware of the location of the recipient object and the recipient object does not know the location of the client. All of the communication between client and object is performed by the ORB and such activity happens transparently as if all the participants were on a single computer system.

2.1.1.1 Structure of an Object Request Broker

Figure 1 shows a request being sent by a client to an object implementation. The client wishes to perform an operation on the object and the object implementation is the code and data that actually implements the object. The ORB is responsible for all of the mechanisms required to find the object implementation for the request, to prepare the object implementation to receive the request, and to communicate the data making up the

request. The client will not know where the object is located or what programming language implements the object implementation [40].

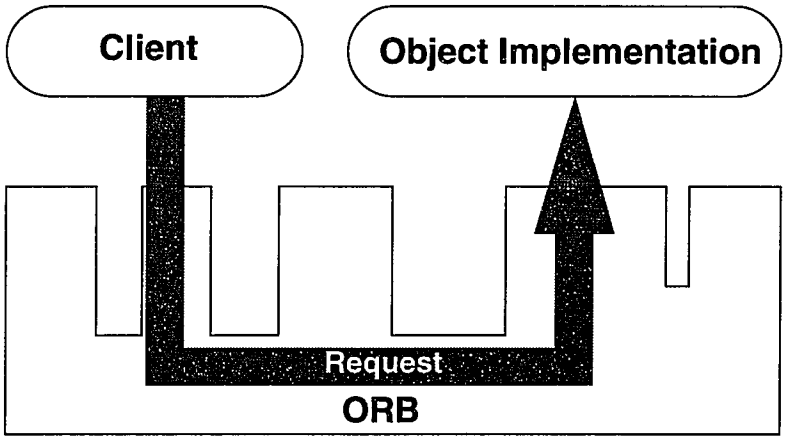

Figure 1: Client sending a request through the Object Request Broker

Figure 2 shows the structure of the Object Request Broker (ORB). In order to make a request, the client can use the Dynamic Invocation interface which is the same interface independent of the target object's interface. Or, the client can use the IDL stubs which are specific to the interface of the object. It is also possible for the client to call other ORB interfaces.

There are two ways for the object implementation to receive the request from the client, either through the IDL generated skeleton or through a dynamic skeleton. The object implementation can call the object adapter during the process of the request.

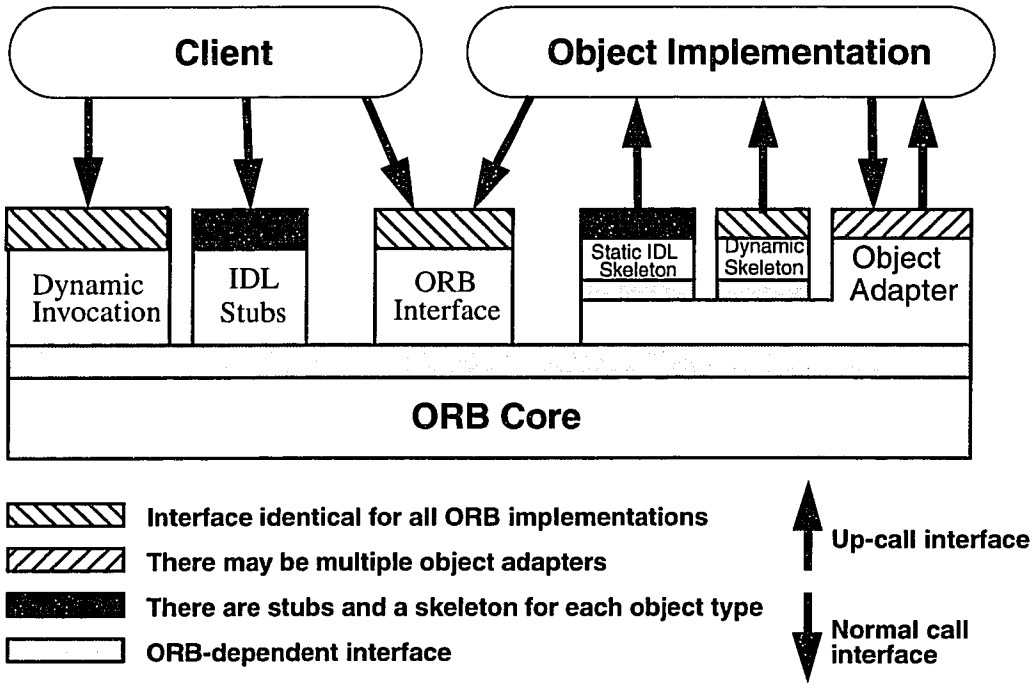

Figure 2: The Structure of Object Request Broker Interfaces [40]

The primary way to invoke services provided by the ORB is through the Object Adapter. It can provide services such as generation and interpretation of object references, object implementation activation and deactivation, mapping objects references to implementations, and registration of object implementations.

Interfaces can be defined in an Interface Definition Language (IDL). The language defines the types and structure of objects and the operation and the parameters to those operations. During a runtime, interfaces can be added to or referred from an Interface Repository Service. The Interface Repository Service provides persistent objects that represent the interface defined in the IDL available at a runtime.

This is just an overview of the functionalities of ORB. For more detail description, please refer to the CORBA specification 2.0 released by OMG in July 1995 [40].

CORBA provides a general framework for a distributed object-oriented system. Any Object Request Broker (ORB) with CORBA compliance can be interoperable under this framework. For example, the ORB of IBM's DSOM can communicate with the ORB of NEO from Sun Microsystems to locate and perform distributed object requests. In the following section, a few CORBA based systems will be discussed.

2.1.2 IBM's DSOM

IBM's SOM (System Object Model) [54] is fully based on CORBA standards and supports operating systems such as OS/2, AIX/6000 and MS Windows. It also provides OMG Common Object Services such as Persistent service that allows objects to be stored or restored to and from a repository that can be a file system, database or object database. The Persistent service will be discussed in the next chapter. SOM also provides frameworks such as replication and emitter frameworks. The Replication framework enables an object to be replicated in the address spaces of several processes distributed about a network. Each replicated object can be updated; the framework will guarantee that the updates are serialized. The Emitter framework aims to reduce the complication in software development. SOM's Interface Definition Language (SOM IDL) compiles with CORBA's standard for IDL; it also adds constructs specific to SOM. There are two IDL compilers that include SOM, SOM IDL and OIDL which is the OMG CORBA IDL compiler. Programming languages supported by SOM are C and C++. In order to address

the cross-platform interoperability problem, SOM relies on its distributed object framework, sometimes called DSOM (Distributed SOM). DSOM is a set of SOM classes that transparently extends the method dispatch mechanisms embodied in the SOM runtime engine to allow methods to be invoked, in a programmer transparent way, on objects in a different address space or on a different machine from the caller. DSOM is fully CORBA compatible, supporting all CORBA data types, functionality and programming interfaces. Currently, fully interoperable versions of the DSOM framework are available for SOM on AIX, OS/2 and Windows [54].

2.1.3 SUN MICROSYSTEMS' NEO™

NEO™ formerly called Distributed Objects Everywhere (DOE) is a more powerful system than DSOM. It not only provides a full CORBA Specification 2.0 facility but also numerous Common Object Services to support the CORBA framework such as Naming, Persistent Storage Manager, Associations, Properties, Events, and Life Cycle Services [55]. Life Cycle Services, for instance, define services and conventions for creating, deleting, copying and moving objects. In distributed environment, life cycle services create a set of services and conventions for clients to perform life cycle operations on objects in different locations. Similar to DSOM, NEO™ provides language bindings for C/C++ [41]. NEO's Object Request Brokers (ORBs) can also interoperate with other CORBA compatible ORBs such as Iona' Orbix for MS Windows on PC based platforms. With the recent development of Java™, an object-oriented programming language developed by SunSoft, it can provide Internet access to NEO environment through Java scripts.

2.1.4 POSTMODERN COMPUTING'S ORBELINE

ORBeline is a complete implementation of the CORBA specification 1.1. It provides all the features specified in the specification. ORBeline's Smart Agent and Dynamic Directory Service provides an easy way for migration and replication of objects. Special features are fault-tolerance, failure recovery and event handling. ORBeline can interoperate with other major ORB products [38]. On the other hand, ORBeline only provides language bindings for C++. Other Common Object Services such as Persistent Object Services are not supported in the current version. The implementation of persistent services and security (to be discussed in the next chapter) will be built on top of ORBeline to provide a complete CORBA framework for Object-Oriented Database Management System in a distributed environment. Advantages of ORBeline includes its availability and completeness to CORBA specification.

2.2 PERSISTENT OBJECT STORES

Object-Oriented Database Systems (OODSs) became available on the market in late 1980s. Many object-oriented database systems are currently available from both commercial vendors and educational institutions. Earlier OODSs are O2 [2][15][27][28], Orion [24], and Iris [18]. Object Store from Object Design, Inc. [25] is one of the latest generation of commercial OODSs. Many OODSs are developed by research laboratories and educational institutions such as Exodus from University of Wisconsin, Madison [9][10], ODE and EOS from AT&T Bell Labs [5], OBST from Forschungszentrum Informatik, Germany [11], and Texas Persistent Store from University of Texas, Austin

[52]. Before we move on to the next section, two terms, persistent object and transient object will be defined. The term "persistent object" can be defined as an object which is valid beyond the life-time of the process that created it [23]. On the other hand, transient object's lifetime ends when the process that created it terminates. In the following, we provide a brief discussion of a database architecture which is based on Texas Persistent Store, Exodus and EOS.

2.2.1 TEXAS PERSISTENT STORE

Texas Persistent Store, developed at the University of Texas at Austin, is a persistent storage system written in C++. One of the key feature of the design is the use of pointer swizzling at page fault time. This technique makes use of the virtual memory management scheme available in most current systems. In Texas Persistent Store model, persistent objects are treated similarly as the transient objects. The only difference is persistent objects are loaded into virtual memory transparently from disk. Transient objects are resident in the memory throughout their life. Therefore, it is not necessary to distinguish between persistent objects and transient objects in the client code.

Objects can be retrieved by name in Texas Persistent Store model. When a rooted object is requested by name, a page of virtual memory is reserved and protected. The actual object will not be loaded into the reserved page until the object is actually referenced. If the rooted object, has been page faulted into virtual memory, then a page will be reserved and protected for each successor object from the root. This process will be continued in this manner. Hence, pages of virtual memory are reserved one pointer ahead for each object being page faulted into virtual memory [52].

The technique of pointer swizzling at page fault time has also been used in the commercial product, ObjectStore [25]. The main advantage of this technique is providing an efficient method for a high performance object storage system.

2.2.2 EXODUS

EXODUS is one of the extensible DBMS projects developed in late 80s and early 90s at the University of Wisconsin, Madison [7]. The goal of the project is to develop a set of primitive DBMS facilities for constructing application specific DBMS. One of the major facilities of EXODUS is the storage manager which provides the basic procedures for file operations such as file creation, deletion and scans. It also provides procedures for storing and destroying objects from a file. Small objects and large objects are treated differently in EXODUS. Objects that can be stored in one page are treated as small objects. The storage manager will automatically convert an object to a large object if it exceeds the size limit of a page. The object identifier (OID) of a small object points to the disk location of the object. For the large object, its OID points to the object header instead. A large object is an uninterpreted byte sequences. It is represented as a B+ tree index on byte position within the object plus a collection of leaf blocks [9].

EXODUS provides another way to support persistence with the introduction of E language. The E language is an extension of C++. It defines the DB (database) attributes and a couple of predefined classes for manipulation of persistent objects. Generic classes for unknown types are also added to enhance the development of database implementation. Generic classes can be used with one or more unknown types within the class definition. Type information can be provided when the class is actually declared.

The idea of generic class is very similar to the template class of C++ which has been introduced in the later version of C++.

The successor of EXODUS is Scalable Heterogeneous Object REpository (SHORE) which was developed at the University of Wisconsin at Madison [10]. The objective of SHORE is to provide heterogeneous object management for object-oriented database and file systems. A parallel version of SHORE has also been developed which is built on top of PVM[1] for interprocess communication [10].

2.2.3 EOS

EOS is a storage manager developed at AT&T Bell Labs [6] for high performance DBMSs. The EOS is serving as the storage manager for the OODBMS, ODE which is also developed at AT&T Bell Labs. EOS provides extensive support for large objects. Objects can be named and retrieved by name efficiently. EOS also follows the ODMG-93 standards proposed by the Object Database Management Group [12] to provide persistent reference to an object type.

Objects are stored in the database or storage areas which can be UNIX files or raw disk partitions. Storage areas can be shared or private. The EOS server will control the access for multi-users for the shared areas. Users can create private areas on local machine for faster performance. In EOS, there are two kinds of object representations, file objects and ordinary objects. File objects are objects which are related. This approach provides a mechanism for sequential scanning of objects within a file. Ordinary

---

[1] PVM is a set of message passing libraries written in C or Fortran for implementation of parallel programs in a heterogeneous environment [20].

objects and file objects have object headers attached to them. The object header contains information of the object such as the object length, whether the object is named or unamed and so on. The header only occupies two bytes of space. A handle is a object reference from which we use it to perform object operation. An object handle must be obtained first before one can operate the object. The handle contains the address of the object in the EOS buffer pool. The page where the object is located will then be locked. When the object handle is released, the page will be unlocked.

The way EOS defined large and small objects is similar to EXODUS. A small object is an object which can fit into one page, the object is large otherwise. Users will see no differences in accessing small or large objects. In addition, EOS provides access to portions of an object. It is useful for handling a very large object [6].

EOS is a rather complete implementation of a persistent object store. It has all the basic features for a data storage system such as concurrency control, logging, transaction commit and abort, checkpoint, and recovery from system crash. It can be extended by associating actions with certain primitive events. Primitive events are low-level events such as page fault, object fault, transaction commit and so on. In our complete implementation of a distributed OODBMS based on CORBA, EOS has been used to serve as a persistent object store. EOS has both C and C++ bindings. There are several advantages for choosing EOS as the persistent store over the others. First of all, EOS has a C++ binding and actually EOS is implemented in C++. Secondly, EOS is a more complete database system which provides reliable functionalities. Moreover, EOS provides persistent reference ability which conforms to the ODMG-93 standards [12]. Lastly, EOS is very easy to use and to install when compared to EXODUS and Texas

Persistent Store. Even though EOS provides a client-server architecture, one can still use the CORBA structure to handle all requests and operations to the persistent store. Clients can send requests to the server of the persistent storage only through the Object Request Broker (ORB) rather than the default communication routines provided by EOS.

So far, we have gone through some basic concepts of the Common Object Request Broker Architecture (CORBA), three commercial implementation of CORBA, and three persistent storage systems. In the next section, we will discuss some of the security models for our environment. We will focus more on the security models for the object-oriented database management system (OODBMS).

2.3 Security Models

Distributed object-oriented databases provide real-life relationships to the stored objects. Maintenance of the complex object relations and distribution of processing over a network pose some challenges for the database security which addresses the issues of confidentiality, integrity and denial of service [35]. Various models for security have been proposed for dealing with this complex system. Generally speaking, we can divide these models into two categories; namely, mandatory access control and discretionary access control. Mandatory security mechanisms are used to enforce multilevel security by classifying the data and users into various security levels and then implementing the appropriate security policies of the organization [17]. Such policies are necessary to fulfill the requirements set by some organizations such as the Department of Defense (DoD) [14]. On the other hand, discretionary security mechanisms are identity-based access control policies. They usually grant privileges to users including capability to

read, write, or update specific data.

In the following sections, we will discuss, in detail, multilevel security including the concept of polyinstantiation. Next, we will move our focus on some discretionary security models such as negative and positive authorizations, strong and weak authorizations, and finally roles model.

2.3.1 MULTILEVEL SECURITY MECHANISMS

The concept of multilevel security comes with the classification of users in computer systems. A user who attempts to access specific classification of data requires clearance for such classification or higher security privileges. Thus, there is a hierarchical sensitivity level in a security classification system such as Top-secret, Secret, Confidential and Unclassified. In addition, there is also a set of nonhierarchical categories in association with the hierarchical sensitivity level [30]. For example, a user having clearance for the secret level can access information classified in that level or lower such as confidential and unclassified. However, this user can not access information on the top-secret level. On the other hand, users who want to access information on a specific security level and in a specific category are required to have clearance for both.

Most commonly used models for multilevel security are defined in the Bell and LaPadula security model [4]. This model classifies each subject such as user, account or program, and object such as relation, tuple, column or operation into one of the security classifications, Top-secret, Secret, Confidential or Unclassified. For simplicity, we only use these four security classifications to describe the model. There are two properties or

restrictions for the subject and object classifications:

1. Simple security property: A subject S is not allowed to have "read" access to an object O unless $Classification(S) \geq Classification(O)$.
2. *-property: A subject S is not allowed to have "write" access to an object O unless $classification(S) \leq classification(O)$.

It is obvious that the simple security property guarantees that no subject can read an object whose security classification is higher than the classification of the subject. The *-property prohibits a subject from writing into an object that has lower security classification than that of the subject. This property prevents information flow from higher to lower security level.

The multilevel classification system leads to the concept of polyinstantiation. Polyinstantiation arises where several tuples can have the same primary key but have different attribute values for users in different classification level. The situation arises because of the simple security property which allows users with higher classification to read attributes of objects with equal or lower classification level. Therefore, users with higher security level will be able to read more information from the same primary key of the search than users with lower security level [46].

Even though most of the commercial DBMSs use discretionary access control mechanisms, multilevel security is still required in government, military and corporate applications. Operating systems such as UNICOS for Cray Supercomputers incorporate an option for using multilevel security to handle the user access on the system.

2.3.2 DISCRETIONARY SECURITY MECHANISMS

Discretionary access control models usually involves granting and revoking privileges. Many relational DBMSs, such as Oracle, use this technique for database access. ORION, one of the early development of OODBMSs, has developed a formal model of discretionary security for object-oriented databases [47][48]. The model is comprised of positive and negative authorization. The notion of strong and weak authorization was also introduced in that model [31].

The Department of Defense (DoD) has provided metrics for secure systems. The metrics divide security level into 7 levels, A1, B3, B2, B1, C2, C1 and D, with A1 the most secure and D the least. Secure systems at level C1 and C2 must provide discretionary access control. For level B1 and up, systems must provide both discretionary and mandatory access control. In the following sections, three of the discretionary security models, positive and negative authorizations, strong and weak authorizations, and roles will be discussed.

2.3.2.1 POSITIVE AND NEGATIVE AUTHORIZATIONS

Positive and negative authorizations are sufficient to satisfy the requirement at Class B3 for the criteria set by the DoD. Positive authorizations explicitly specify users' right to access information while negative authorizations explicitly deny users from accessing the information.

Implementation of such authorized schemes requires an object to carry two lists, one for positive authorization and the other for negative authorization. A user must

belong to an object's positive authorization list in order to have access permission on the object. On the other hand, a user must not belong to the negative authorization list in order to access the object. A combination of the positive and negative authorizations provides an easy way to achieve the goal of discretionary access control. However, as pointed out by Rabitti et al. [47][48], there is a situation where the positive authorizations may conflict with the negative authorizations [47][48]. For example, if a user has been put in both the positive and negative authorization lists, it is not clear whether the system should allow or deny the user from accessing the object. This situation may not happen at the same object. However, since child objects inherit the properties from their parent objects, the situation might occur in which a user's name is in the positive authorization list of an object and the same user's name is in the negative authorization list of one of the descendents of the object. To solve this problem, the concept of strong and weak authorizations is introduced.

2.3.2.2 Strong and Weak Authorizations

As in ORION, a strong authorization cannot be overridden by other authorization while a weak authorization can be overridden by a strong authorization. Therefore, the problem would be solved in the example described in the previous section if we assign weak positive authorization in one place and strong negative authorization in the other for the same user. It is still possible to have conflicts such as weak positive authorization and weak negative authorization assigned for the same user at an object. However, this conflict can be detected by a simple tool which browses through the object hierarchy and finds the conflicts.

Another approach has been suggested by Lunt [29] to solve the problem in positive and negative authorizations. Such an approach takes denials as higher precedence. This approach can also be interpreted as all negative authorizations to be strong and all positive authorizations to be weak. One advantage of having precedence on denials is the assurance that specific users and groups cannot obtain authorization to an object without explicit authorization [31].

2.3.2.3 ROLE-BASED SECURITY

User roles have been used in several applications to provide discretionary access control. In UNIX, for example, there are certain built-in roles such as superuser, operator, system administrator, and so on. Each role have been granted certain privileges by the system. A system administrator can create a role for a particular purpose, delete a role or modify a role's privilege. Named protection domains (NPDS) proposed by Baldwin [1] is a facility for specifying user roles. Other research has been done to employ user-role model for object-oriented database security such as the models suggested by Ting et al. [57] and Nyanchama et al. [32][34].

Role-based security provides a flexible way to manage a large object hierarchy. Each role is provided with sufficient privileges to carry its function only and therefore role-based implementation is based on the principle of least privilege [56]. Roles can have overlapping privileges. The object-oriented hierarchy can apply to the role-based model. A role can be the child of another role. In this case, the child role will inherit all predefined privileges set from its parent role. The child role then can have other privileges not in its ancestor roles. Therefore, the role hierarchy can always be built from

a basic role or root role which have minimal privileges. Child roles can be generated from the root role and other roles can be created in this manner.

Figure 3 shows a privilege graph suggested by Baldwin [1]. Each subject or user can be assigned one or more roles. Each role, on the other hand, is either assigned some privileges or inherits from other role(s). For example, subject A and B are both assigned to role of Account Supervisor. The role Account Supervisor is a child of Account Clerk that implies Account Supervisor will have all the privileges from the role of Account Clerk. Therefore Account Supervisor will have the privilege to Compile Accounts. In additional to this privilege, Account Supervisor also has privilege to Audit Accounts which Account Clerk does not have. It is also possible for a single subject to have more than one role. Subject G, for example, has been assigned to role of Shipping Manager and System Administrator. Thus, Subject G will have the privileges of creating Shipping Orders and Creating Computer Accounts.

The hierarchy graph of roles should preserve the acyclic property in order to provide decretionary access control over the privilege set. Nyanchama, et al. [34] presented a formal graph model for the acyclic role organization.

So far, we have discussed some of the existing models suitable for secure object-oriented database systems. Multilevel security is too restrictive in a way that may not be applicable to most commercial applications. Positive and negative authorizations are inherited with ambiguity which can be solved by an association with strong and weak authorizations. A role-based model is flexible and extendable such that we can extend this model to cover some of the features from multilevel security. Our design of security mechanisms on Common Object Request Broker Architecture (CORBA) will be based

on the role-based model which will be discussed in detail in the next chapter.

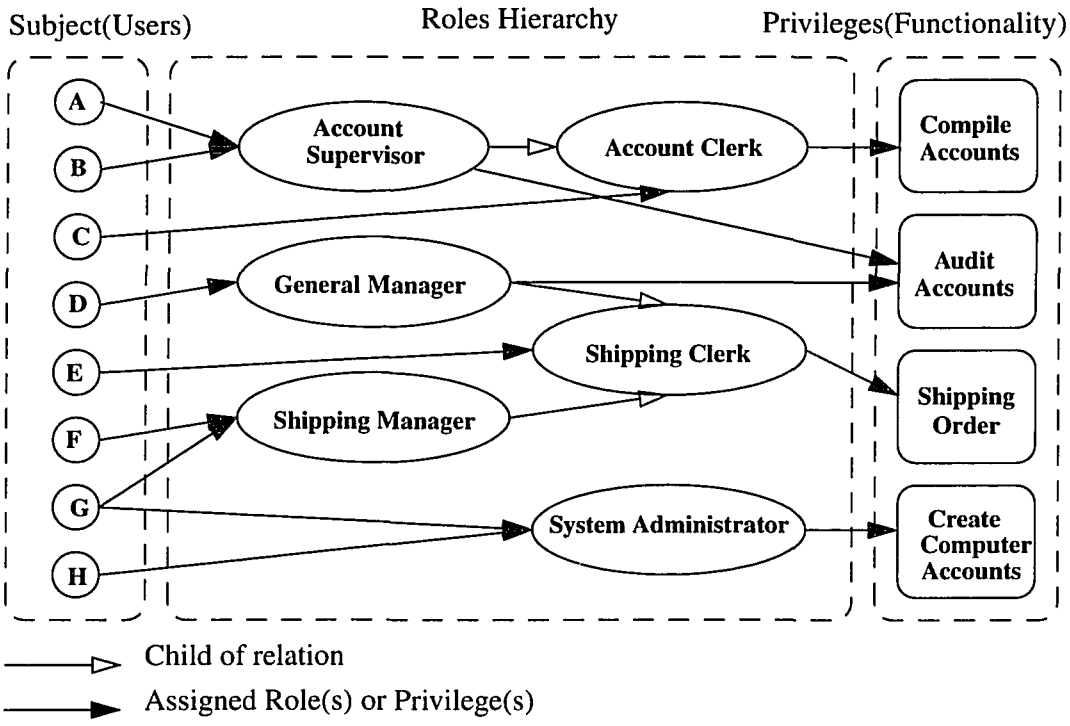

Figure 3: Baldwin's Privilege Graph

2.4 THESIS CONTRIBUTIONS

The multi-level security requirements provide guidelines for designing a security model. The guidelines are necessary to assert the validity of the security model. The role-based security model provides a fundamental model for user authentication. However, the object relationships of roles can cause conflicts in resolving the user privileges. The addition of positive and negative, and strong and weak authorizations can help to resolve these conflicts. These issues are going to be discussed in chapter 4.

The persistent object store provides a physical storage for storing persistent

security information for the role-based security design. EOS by the AT&T Bell Laboratories is the persistent object store in our implementation. EOS is also ODMG'93-compliant which simplify the implementation of the persistent object service for CORBA.

ORBeline by the PostModern Computing provides a basic CORBA implementation. It provides the Interface Definition Language compiler for preprocessing interface definitions. The security and persistent object services are implemented on top of ORBeline.

2.5 SUMMARY

Evaluation of existing distributed object technologies is one of the objective of this thesis. In section 2.1, the general framework of CORBA is explained. Three commercial CORBA implementations, IBM's DSOM, Sun Microsystem's NEO and PostModern Computing's ORBeline are evaluated. IBM's DSOM includes more common object services which provide additional functionalities for distributed application. NEO by Sun Microsystem has recently incorporated with Java as its front end interface. The widely adopted internet programming language Java may lead NEO to be a leading CORBA implementation for internet users.

Two of the persistent object stores, Texas Persistent Store and EXODUS are developed in academic institutions while EOS is developed by AT&T Bell Labs. The major differences among them is the ease of use and installation. EOS is best of the three. It provides minimal set of functions to manage persistent objects. It also provides object clustering for creating file object to allow efficient mechanism for object

management. EOS is designed to handle both large and small objects.

Finally, the discretionary security models provide primary concepts for the design of our security service. Our design can also be extended to a multilevel security model.

The next chapter discusses three persistent protocols for designing the persistent object service. The design and implementation of our persistent object service is explained with the use of EOS as the persistent object store.

CHAPTER 3

DESIGN AND IMPLEMENTATION OF PERSISTENT OBJECT SERVICE USING CORBA

3.1 INTRODUCTION TO PERSISTENCE

An object whose lifetime is transient is allocated memory that is managed by the programming language run-time system. Sometimes a transient object is declared in the heading of a procedure and is allocated memory from the stack frame created by the programming language run-time system when the procedure is invoked. That memory is released when the procedure returns. Other transient objects are scoped by a process rather than a procedure activation and are typically allocated to either static memory or the heap by the programming language system. When the process terminates, the memory is deallocated. An object whose lifetime is persistent is allocated memory and storage managed by the ODBMS run-time system. The objects continue to exist after the procedure or process that creates them terminates [12]. The goal of creating a persistent object service using CORBA is to provide common interfaces for Object Request Brokers (ORBs) to restore and to manage the persistent state of objects [41]. The persistent object service will be used in converting a dynamic/transient object to a

persistent object. Dynamic object is volatile and is typically resided in memory. System failures, for instances, will end the lifetime of a dynamic object. To preserve the persistent state of an object, means should be provided to transfer contents of objects from memory to physical storage such as disk. Persistent object service therefore serves this purpose for maintaining the object state persistence between virtual and physical storage. As shown in Figure 4, Persistent Object service will convert an object from its dynamic state to a persistent state and vice versa. In order for all ORBs to locate the desired object, a persistent handle or reference should be created for each persistent object. This handle identifies the location of an persistent object. The persistent handle must be unique for different objects managed by the ORBs.

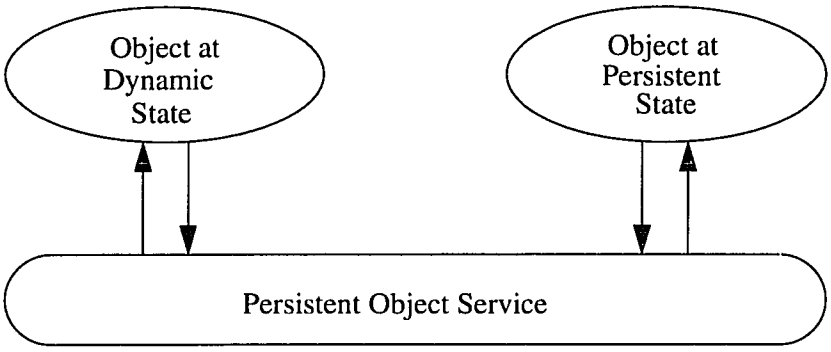

Figure 4: Function of Persistent Object Service

In the next few sections we shall describe several protocols from the specification of Common Object Services (COS) published by the Object Management Group (OMG) [41]. Then we shall discuss the ODMG'93 protocol on persistence. The design of our persistent model is similar to the ODMG'93 protocol and is also part of the COS specification. We now move on to our design and implementation of the persistent store

using EOS developed by AT&T Bell Laboratory.

3.2 OMG PERSISTENT OBJECT SERVICES PROTOCOLS

Common Object Services Specification describes three basic protocols including the ODMG'93 protocol to support persistent object handling. CORBA already provides a persistent reference handling interface that is the object_to_string, string_to_object, release, and so on [41]. These operations allow conversion of object to type string and vice versa and should be sufficient for most of object manipulation by clients. Three protocols will be discussed in this section; namely, the Direct Access Protocol (PDS_DA which stands for Persistent Data Store with Direct Access), the Dynamic Data Object (DDO) protocol and the ODMG'93 protocol.

3.2.1 THE DIRECT ACCESS (PDS_DA) PROTOCOL

Direct Access Protocol represents a persistent object as one or more interconnected data objects. The persistent data of an object is described as a single data object which might be a root of tree containing the object data. It is similar to represent a persistent object as a heap in some persistent object store. In the case of multiple data object (an object consists of several instances of data objects), it requires object traversal from the root object followed by the stored object references.

It is necessary to define the type of each data object within an object. Fortunately, the Interface Definition Language (IDL) provided by CORBA includes the Data Definition Language (DDL) which can be used in describing object types during the interface definition.

As shown in Figure 5, PDS_DA is an object reference to multiple data objects such as instances of object A and object B. Each persistent object is represented by a PDS object reference handle (PDS_DA) in the figure. PDS can locate the data object references within a persistent object from the PDS_DA handle and thus it can perform operations on data object references to get and to modify the attributes of data objects. Attributes can be normal data types defined by DDL or can also be another object instance.

Direct Access Protocol is a simple and direct method to achieve persistence. Implementation of the protocol requires only interface preprocessing facility such as the Data Definition Language (DDL) or Interface Definition Language (IDL).

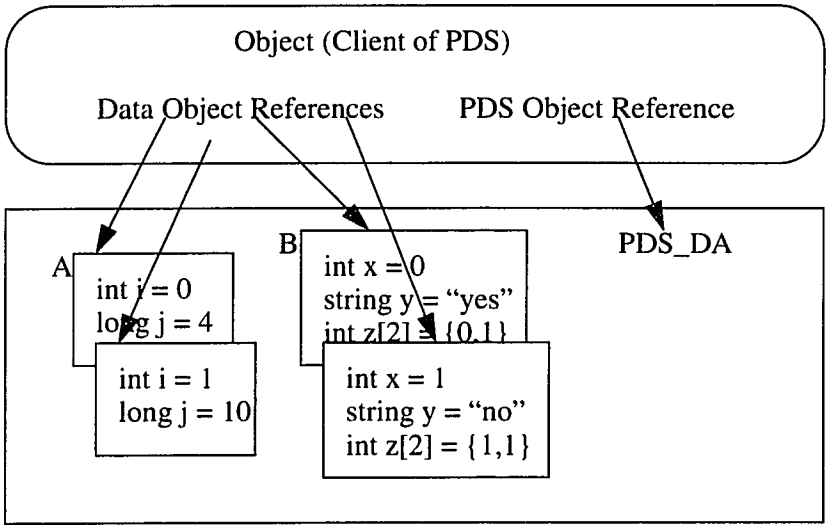

Figure 5: Direct Access Protocol Interfaces

3.2.2 THE DYNAMIC DATA OBJECT (DDO) PROTOCOL

A Dynamic Data Object (DDO) is a Datastore-neutral representation of an

object's persistent data [41]. Datastore-neutral representation is the simplest form to describe an object data independent with datastore type. A DDO is an object containing all data of a single object. Data contained in a dynamic data can be divided into three categories: description of a DDO, data item and data property. Data item comprises of data property and data item information. Figure 6 shows an example of a DDO structure. The data item can store both data types and data methods of an object.

DDO can be optimized in use with specialized types of data store. It provides a fast and simple storage and retrieval mechanism for different types of data store.

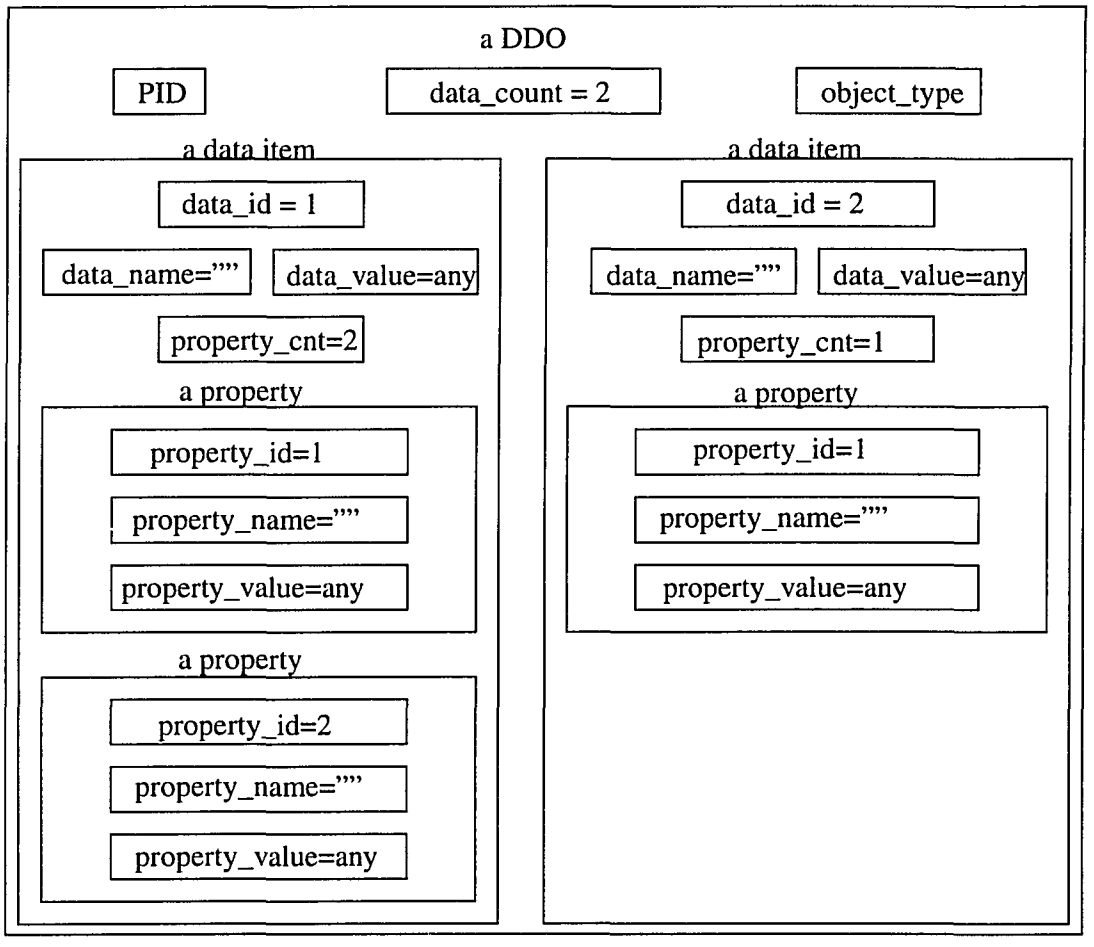

Figure 6: Structure of a DDO [41]

3.2.3 THE ODMG'93 PROTOCOL

The ODMG'93 protocol proposed by the Object Database Management Group is similar to the Direct Access Protocol we discussed before. The only difference between them are that the ODMG'93 protocol uses an Object Definition Language (ODL) to define an object interface instead of using the Data Definition Language (DDL) and ODMG'93 protocol uses programming language mapping defined for data object specified in ODMG'93, rather than the CORBA IDL attribute operations [41]. The IDL attribute operations define the data types in interface definition.

After reviewing three protocols with two different approaches, our implementation of the persistent object service will use the Direct Access Protocol (PDS_DA). One reason of choosing this over the ODMG'93 protocol is that CORBA IDL already defines the DDL for writing the object interface, even though the ODMG'93 protocol has been widely used by most database vendors. We have to rewrite a preprocessor compiler if we choose to use the ODMG'93 protocol. Conversion from the PDS_DA protocol to the ODMG'93 protocol is straightforward since the ODL can be easily interpreted as DDL. In the following section, we are going to discuss our persistent object service in use with the EOS storage manager developed by AT&T Bell Laboratories.

3.3 DESIGN OF THE PERSISTENT OBJECT SERVICE

Our aim is to provide persistent object services for one or more datastore interconnected through the CORBA communication layer. To achieve this goal, our

design should be able to adapt different object data stores. Throughout this section, we will provide a high level overview of our design.

We use a Persistent Object Manager (POM) to identify the object type, data store and the corresponding persistent data store for an object. POM contains a table of object type, datastore type and persistent data store. In our implementation, each persistent object has a persistent ID which is a combination of the type, location and name of an object. When the POM receive a request from client to set or get an object, the client will pass the PID of the specified object. The POM can then locate the object information from the PDS registry and identify the protocol and data store to store or to restore the object.

Figure 7 shows the role of POM to handle requests from different clients and to establish connections between clients and persistent data stores. We plan to use EOS as a persistent object store for our Role security model. Client 1 will first send request to the POM for retrieving information of the Role graph object with pid1 which is the persistent ID for the object. The POM will then look up the object location at the persistent datastore (pds1) from the registry with the information submitted by the client. Afterward, a connection will be established between the client 1 and the EOS data store (pds1). Client will not know in which protocol or datastore the object is located. All the transparency is handled by the POM and the Object Request Brokers (ORBs). Similar approach can be used if we would like to add on capabilities to handle different object stores such as ObjectStore by Object Design, Inc. [25] and Versant. There are no strict rules for using protocols with different object stores. Figure 7 just shows an example for various combination of protocols and data stores.

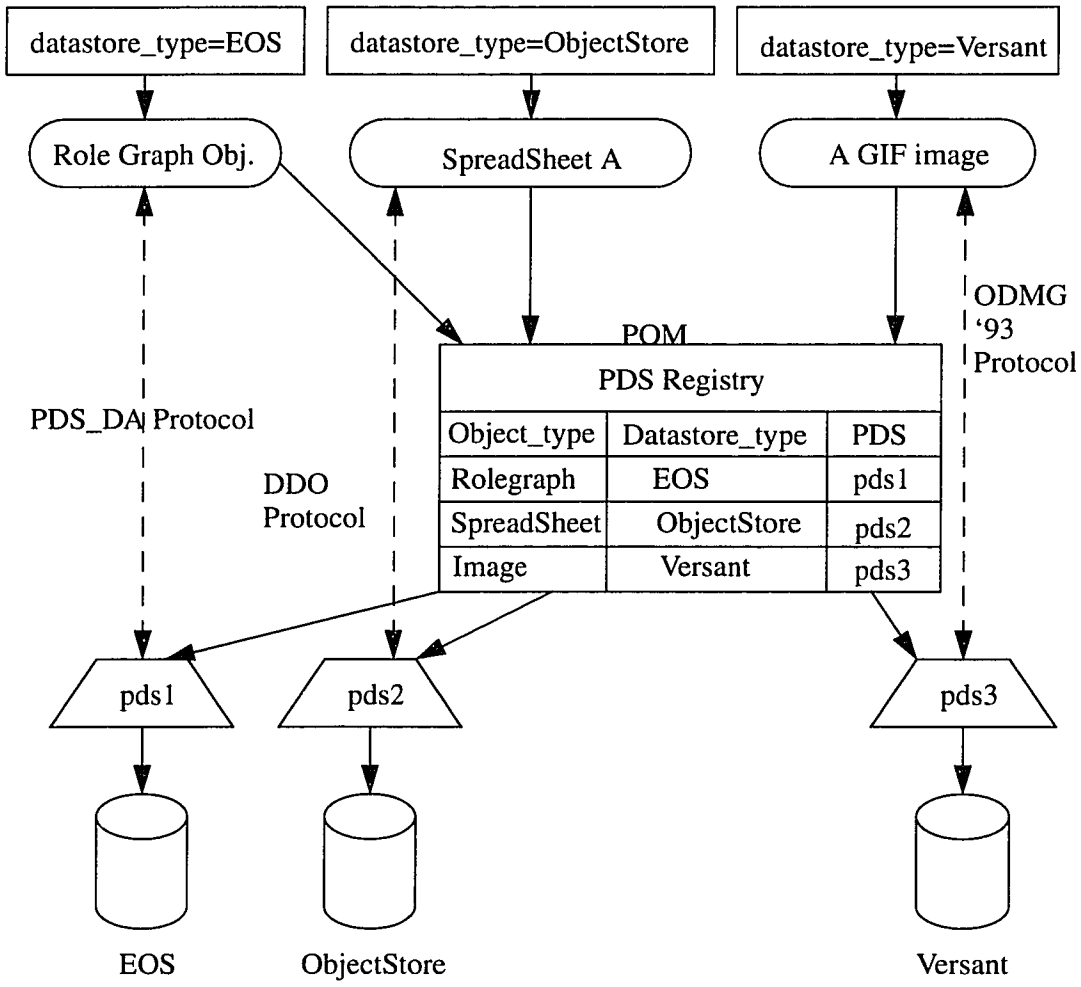

Figure 7: The role of POM

There are dependency issues regarding different methods for object management with different data stores. Therefore, we need to provide interfaces for different data stores. Figure 7 has shown another key component other than the POM that is the pds's. For example, pds1 and pds2 are required for EOS and ObjectStore respectively. A PDS, persistent data store, is a data store and protocol dependent interface. To be specific,

there are two basic functionalities of PDS.

1. Interacting with the object to retrieve and to store data in and out using a specific protocol which has been discussed earlier in this chapter.
2. Interacting with the data store to retrieve and to store an object. In our case, we will provide an interface for handling objects in and out from EOS storage manager.

In the following section, we will focus on our implementation of the persistent object service.

3.4 Implementation of Persistent Object Service

Throughout this section, we will explain our implementation of the persistent object service in details by providing both CORBA IDL interfaces and our C++ class definitions. IDL interfaces, in general, provide a high level description of object classes and will directly map to C++ classes by the IDL compiler. Classes generated by IDL interfaces involving virtual base classes which should be implemented by implementors.

3.4.1 Persistent Identifier (PID)

In order for the Persistent Object Manager (POM) to locate a persistent object, each persistent object is associated with an unique ID. In our implementation, a PID consists of a datastore type of an object, a object ID to identify the object in a specific data store, and a hostname or an IP address of the datastore. The IDL interface is very simple and is shown in Figure 8. It contains an object PID with a member datastore_type and a member function get_PIDString() to get the PID in CORBA::String format.

```
module CosPersPID {
   interface PID {
       attribute string datastore_type;
       string get_PIDString();
   };
};
```

Figure 8: IDL for the Persistent ID interface

To implement the actual function, we need to create a derived class from the CosPersPID module. The definition of the derived class is shown in Figure 9. We are showing both the whole IDL interface and its class implementation only at this simple introduction. We will only give the important part of the codes for other interfaces.

```
class PosPID: public CosPersPID_impl::PID_impl
{
private:
  CORBA::String _datastore_type;
  CORBA::String _id;
  CORBA::String _ip;
  CORBA::String *_pid;
public:
  PosPID(const char *datastore, const char *id, const char *ip)
    : _datastore_type(datastore), _id(id), _ip(ip), _pid(NULL)
      {}
  ~PosPID() {}
  CORBA::String* datastore_type(); // function to return
datastore type
  void datastore_type(const CORBA::String& val); // function
to set datastore type
  CORBA::String* get_PIDString(); // function to get PID as a
```

Figure 9: Persistent ID derived class

3.4.2 PERSISTENT DATA STORE USING EOS

EOS is an efficient object database developed by AT&T Bell Labs. In EOS, data objects are stored in one or more EOS storage areas which are preformated by an EOS area format routine. Each storage area consists of a bundle of pages. The size of a page can be preset to a certain extent prior to formatting the storage area. A data object can be

stored at one or more page depending on its size. Object stored in EOS are identified by a unique name or an object ID. EOS provides a set of routines to retrieve an object handle from which our client can reach the object and perform operations.

We have implemented a set of basic routines to allow the Direct Access Protocol to connect to and to disconnect from the EOS data store, plus operations to store, to restore and to remove objects. The IDL for the Persistent Data Store module is shown in Figure 10.

```
module CosPersPDS {
  interface Object {};
  interface PDS {
    PDS connect (in Object theobj, in CosPersPID::PID p);
    void disconnect (in Object theobj, in CosPersPID::PID p);
    void store (in Object theobj, in CosPersPID::PID p);
    Object restore (in Object theobj, in CosPersPID::PID p);
    void remove (in Object theobj, in CosPersPID::PID p);
  };
};
```

Figure 10: IDL declaration of the Persistent Data Store (PDS)

Subsequently, we define a derived class from CosPersPDS module for the client to connect to the EOS data store. Figure 11 shows the class definition for the persistent data store. An persistent object is represented by an instance of class PosPDS where the object handle is _obj_data and can be obtained through a series of database operations.

Typically, the Object Request Broker needs to "connect" to a EOS server first. Then, we can perform various database operations such as to "store" and to "remove" an object. Member functions such as "trans", "createfile", "openfile", "closefile" and "commit" are private operations necessary to establish connection to the EOS storage manager and are only used by the member functions themselves. EOS handles objects by the object handle. An object can be identified by its Object ID (OID) or its object name.

3.5 SUMMARY

We have designed and implemented a persistent data service with EOS as the data object storage. This is an add-on persistent facility for the CORBA implementation of ORBeline by Post Modern Computing. In the next chapter, we shall describe how to add security to CORBA. We also explain in detail its design and implementation.

```
class PosPDS: public CosPersPDS_impl::PDS_impl
{
private:
        CosPersPDS::Object *_obj_data;
  ....
  int trans();
  int createfile();
  int openfile();
  int closefile();
  int commit();
  long getobjsize();
public:
....
  CosPersPDS::PDS* connect(CosPersPDS::Object* obj,
CosPersPID::PID* p);
  void disconnect(CosPersPDS::Object* obj, CosPersPID::PID* p);
  void store(CosPersPDS::Object* obj, CosPersPID::PID* p);
  CosPersPDS::Object* restore(CosPersPDS::Object* obj,
CosPersPID::PID* p);
  void remove(CosPersPDS::Object* obj, CosPersPID::PID* p);
....
};
```

Figure 11: Class definition of the Persistent Data Store

CHAPTER 4

DESIGN AND IMPLEMENTATION OF A SECURE OBJECT SERVICE

Security is still a missing component from CORBA specification 2.0 released in Fall 1995 [40]. The Object Management Group is currently requesting proposals for adding a security service to CORBA. All existing CORBA implementations rely on the system level security as the underlying security mechanism. System level security such as the UNIX system security mechanism is not designed for management of object entities. It is designed to handle file systems and control of processes. In this chapter, we propose our security model which can not only handle object entities but also be extended to meet the multilevel security requirement. This chapter is divided into two main parts. First section will provide a high level design of the role security model. We will discuss several authorization schemes to handle specific needs for the role model. Second section will focus on the actual implementation of the role model. We will present our role graph management algorithms and implementation of the security service on CORBA.

4.1 DESIGN OF THE ROLE SECURITY MODEL

We have discussed different types of security mechanisms including multilevel, discretionary, authorizations and role-based security models in Chapter 2. Our task is to design and implement a security service add-on to a CORBA implementation. The security service should be extendable to different levels of security requirements set by the DoD. In the following sections, we will introduce our basic design of a role security model. Then, we will add on various authorization schemes to cover the limitations of the role model. In addition, an abstract model of the design described in a Booch diagram [7] will be discussed at the end.

4.1.1 THE ROLE MODEL

A role model consists of a collection of privileges and a set of users who can access the role. Privilege is an access right for system information. A role can inherit the privilege from other roles. Before presenting our design model, we will use the following definition throughout this section.

Definition 3: Φ is a universal set of roles.

Definition 4: Privilege set: $P(\chi)$ is the privilege set of role χ.

Definition 5: Senior Role: Let $\Re$ be a set of roles. Role χ is a senior of role $\Re$ if and only if $\forall\psi((\psi \in \Re) \rightarrow (P(\psi) \subseteq P(\chi)))$

Definition 6: Junior Role: A role ψ is a junior role of role χ if and only if $(P(\psi) \subseteq P(\chi))$

We also define a super role which is the senior role of all other roles within the

role graph. The duty of the super role is to maintain the role graph. A User that belongs to the super role can add a new role to and delete a role from a role graph. Role graph modification can be only performed by users that belong to the super role. In a distributed database system, it may be impossible to have a super role which can manage all databases distributed in a wide area network. In our design, the super role exists only in local area network.

We design our role model as a graph containing all roles for a local area network. The role graph is an acyclic directed graph. Graph traversal is unidirectional. Senior roles can access their junior roles, but not the reverse. This prevents data from being accessed by users without necessary security clearance. Senior roles also inherit the privileges of their junior roles. Figure 12 shows a graphical model of a role graph. It is represented as a multi-level role tree. The role tree is a hiearchical structure in which the superior roles have rights to access more system privileges. The role graph shown in Figure 12contains redundant role paths. For example, role B can reach role L from path BL or from paths BE and EL. Paths BE and EL already implies role L is a junior role of both role B and role E. Therefore, we can remove the path BL without altering the privilege set of role B and role E.

Figure 13 shows the simplified version of the previous role graph with minimum redundancy. Each role still preserves its system privileges as the previous graph. The graph still have more than one path to a particular role, but we can not eliminate the paths since the system privileges among those roles will be changed. Two more properties will be added on top of the role-based security model and they will be discussed in the following two sections.

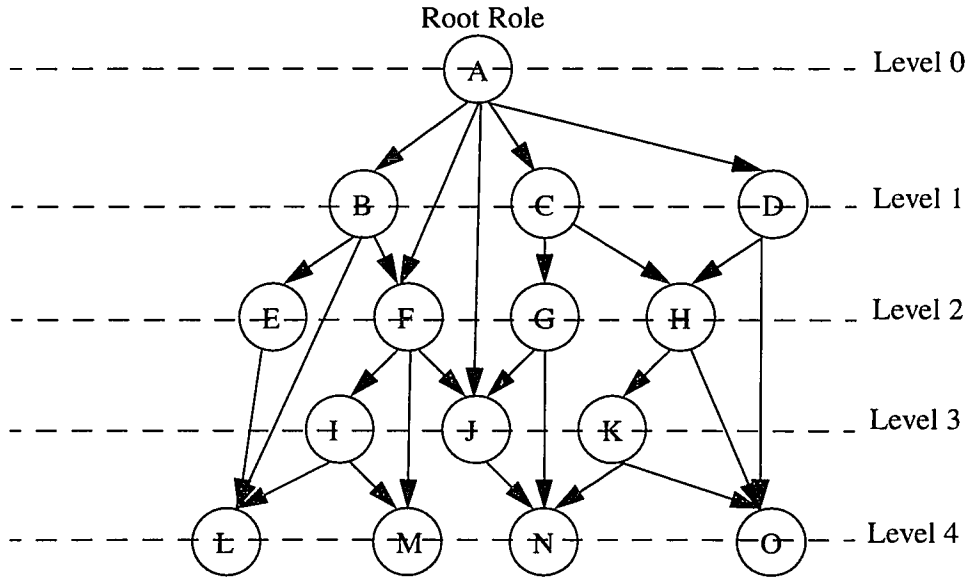

Figure 12: A typical role graph

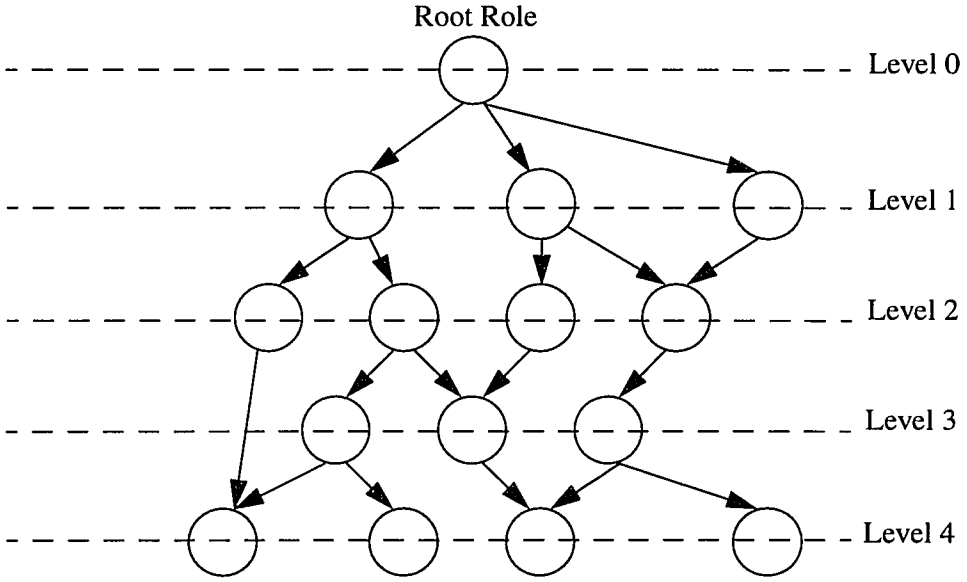

Figure 13: Simplified role graph with minimum redundancy

4.1.2 Implicit and Explicit Privileges

In our role graph design, a role can have two different kinds of privileges. One is explicit privilege which belongs to the privilege set associated with the role. The other is implicit privilege which belongs to the privilege sets associated with the junior roles. In order to obtain both the implicit and explicit privileges of a role, we need to perform a role graph traversal along the role paths. In general, we will perform a Breadth First Search (BFS) to collect both the implicit and explicit privileges for a role.

4.1.3 Authorization Properties

4.1.3.1 Positive and Negative Access Control

It is necessary to have positive and negative authorization in a role graph implementation. Positive authorization is used to explicitly grant access to a set of users. Negative authorization is used to deny access rights to a set of users. Thus, each role will incorporate a list of users who have rights to access the current role and a list of users whose access will be denied to the current role. This explicit access and denial properties is mandatory to fulfill the DoD multilevel security requirements [14].

4.1.3.2 Weak and Strong Access Control

Positive and negative authorization may cause conflict under certain situations. Figure 14 shows an example of a conflict resulting from improper role graph management. User Sam is in both the access and denial lists of Role A. Should we allow

or deny the access to Sam? To preserve minimum access rights for users, we assign the denial list to have stronger authorization than access list. Therefore, in this example, Sam does not have access rights to role A.

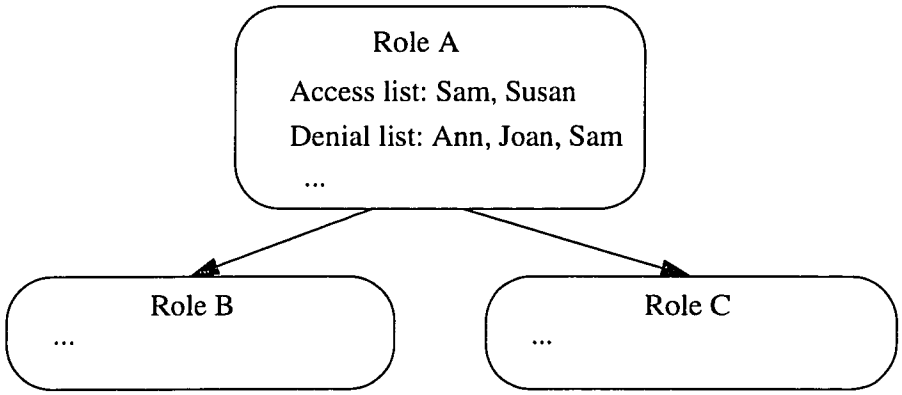

Figure 14: Example of conflicts between access and denial list

4.1.4 ROLE GRAPH MAINTENANCE PROPERTIES

4.1.4.1 ROLE ADDITION AND DELETION

Any operation performed on a role graph must preserve the consistency of the security infrastructure. After adding a new role to a role graph, the acyclic property must be preserved. When a new role is being added, new paths are generated from its immediate senior and junior roles. Redundant edges are removed to minimize the edges in the graph. Figure 15 shows the result of removing redundant edges. After adding new role X, edge BC and edge AC can be removed since they can be replaced by edge BX and XC, and AX and XC respectively.

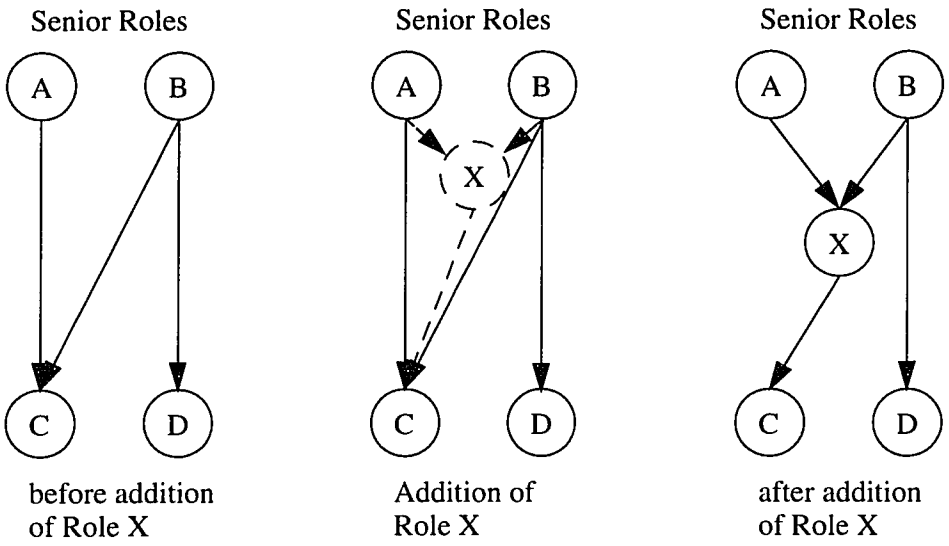

Figure 15: Addition of new role

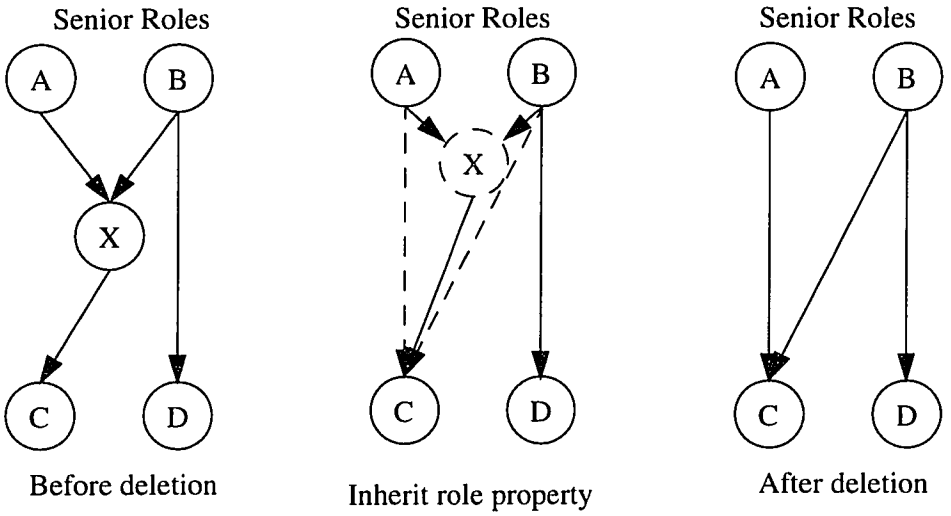

Figure 16: Deletion of role

On the other hand, when a role is deleted, edges associated with the deleted role

are removed at the same time. New edges are added to preserve the role hierarchy of the role graph. Figure 16 shows the deletion of role X. When role X is deleted from the graph, without proper modification, role A and role B will no longer be the senior role of role C. In that case, the privilege sets of role A and role B are reduced. In our design, we try to keep the roles inheritance to be the same after the role deletion so that role A and role B are still the senior roles of role C after the deletion. Therefore, we add new edges AC and BC to eliminate privilege reduction from deleting role X.

4.1.5 High Level Design for the Role Model

During our design process, we employ the Booch Method [7] for designing object-oriented applications. It provides models for representing complex object relationships, class inheritance, access control among classes, and so on. Figure 17 shows our design in a Booch Diagram. We summarize the basic components in the design model in the diagram.

Each icon with hyphenated line boundary in the diagram represents a class. We have two abstract classes CosRID and CosRoles. CosRID represents the common object service for manipulation on role identifier while CosRoles represents the common object service for operation on Roles. The little triangle with a letter 'A' indicates that the class is abstract. These abstract classes should be defined in the interface using the Interface Definition Language (IDL) in order to allow the Object Request Broker to locate the objects. Class CosRID is a basic class to handle the Role identifier (RID) which is the key to identify a role. Class RID is derived from class CosRID and therefore class RID inherits all the methods of class CosRID. RID consists of three attributes; a role ID, a

hostname/IP address and a role name. Two member operations are necessary to retrieve the role ID and the role name.

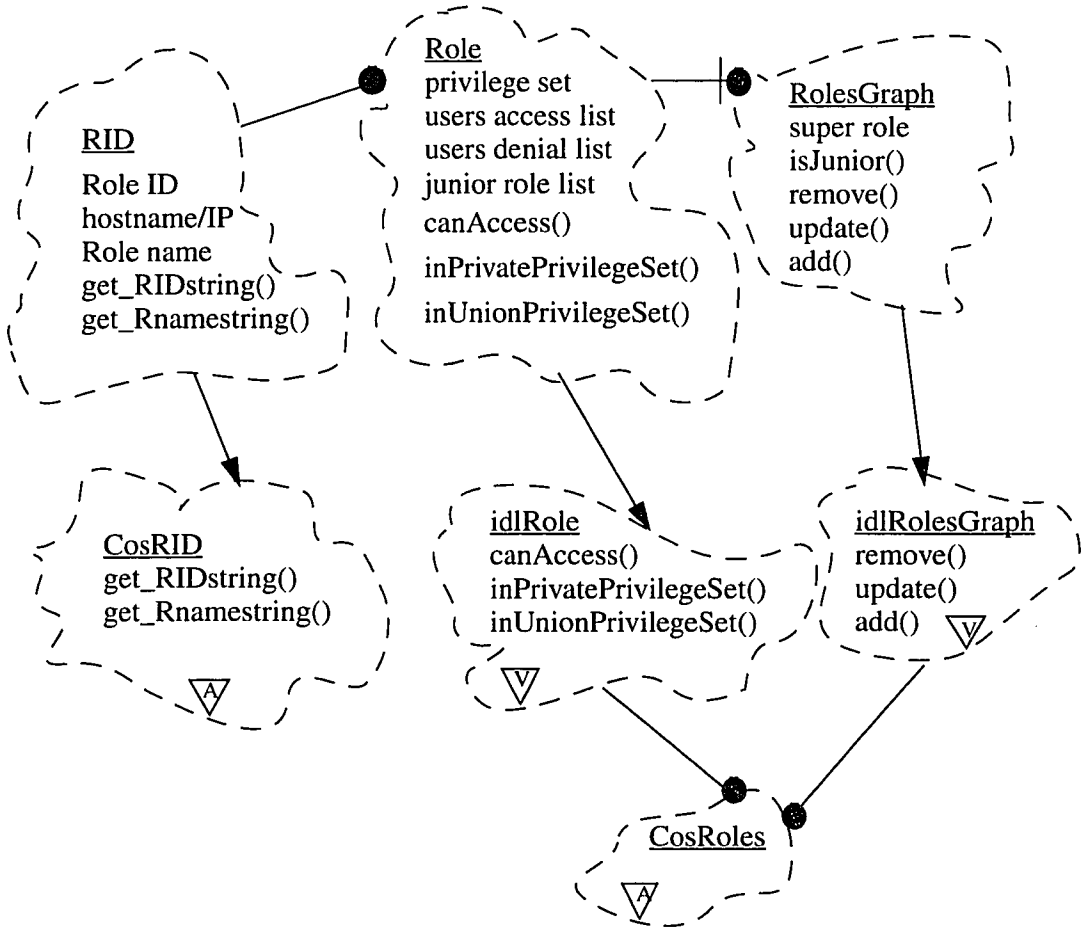

Figure 17: Booch diagram of our Role graph design

Class idlRole is a virtual class which contains three virtual functions. Virtual functions are functions that will be defined in a derived class. Function "canAccess" checks if a user can access the current role. Function "inPrivatePrivilegeSet" checks if a

user has explicit rights to access a specified privilege. Function "inUnionPrivilegeSet" checks if a user has implicit rights to access a specified privilege. The actual implementation of these functions are done on the derived class Role. Class Role also includes four attributes; a privilege set containing the explicit privileges of the role, a users' access list containing a set of users who have rights to access the role, a users' denial list containing a set of users whose access to the current role will be denied, and a set of junior roles.

Class idlRolesGraph contains three virtual functions which perform basic role graph maintainence such as adding a new role, updating an existing role and removing a role from the role graph. These operations require special properties which will be discussed in the next section. The actual implementation of class idlRolesGraph is a class RolesGraph which has additional attributes and member functions. Function "isJunior" will check if a role x is a junior role of role y. The word junior role here can be implicit junior which simply means role x is a descendent of role y. The diagram also shows that the RolesGraph has access rights to the protected members of class Role. It is represented by the special strip and filled circle at class RolesGraph in the diagram.

We have described our design as an abstract model. The next section will present our implementation interfaces and class definition in details.

4.2 Implementation of the Security Model

In this section, we first present our implementation of roles. Then we show how the role implementation can be integrated as a CORBA service.

4.2.1 ROLE IMPLEMENTATION

A role graph is the key of our security model. During the implementation, the feature of data hiding from C++ will help us to protect data and member functions from being accessed without access rights. Figure 18 shows the IDL of role module describing the two interfaces: the interface of idlRole and the interface of idlRolesGraph. The interface idlRole performs a stand alone operation for a role such as checking access right for a user. The interface idlRolesGraph performs basic graph maintainence such as adding, deleting and updating a role in a role graph. In general, to check access rights of a user for a system privilege, we first find out a role X associated with the user and then check if the privilege is in the role's union privilege set which is the union of the privilege set of the role and its junior roles. It can be done by a simple traversal of the nodes in the subtree headed by the role X.

Before a new role is added, we first check to see if the acyclic property is preserved after addition. If so, we add the new role and minimize the redundant paths within the role graph. Deletion requires new paths creation from the senior roles to the junior roles of the deleted role as explained in previous section.

```
#include "CosRID.idl"
module CosRoles {
  struct auth_para {
    string login;
    string ip;
  };
  typedef sequence<string> seqId;
  typedef sequence<auth_para> seqAuth;
  typedef sequence<CosRID::idlRID> seqRid;
  interface idlRole {
    boolean canAccess(in string login, in string ip);
    boolean inPrivatePrivset(in string privId);
    boolean inUnionPrivset(in string login, in string ip, in
                           string privID);
  };
  interface idlRolesGraph {
    idlRolesGraph remove (in CosRID::idlRID rid);
    idlRolesGraph add (in seqRid seniorset, in idlRole role);
    idlRolesGraph update (in seqRid seniorset, in idlRole
                          new_role, in CosRID::idlRID rid);
  };
};
```

Figure 18: IDL of Role Module

4.2.2 INTEGRATION OF SECURITY SERVICES ON CORBA

CORBA supports a collection of services that provide basic functions for using and implementing objects. Object services are necessary to construct in any distributed application and are always independent of application domains. Examples of object services are naming service to bind or to resolve a name to an object relative to a naming context, event service to deliver asynchronous events, and life cycle service to define conventions for creating, deleting, copying and moving objects. Object services allow an add-on facilities capability for CORBA. We implemented our security service as a common object service on CORBA.

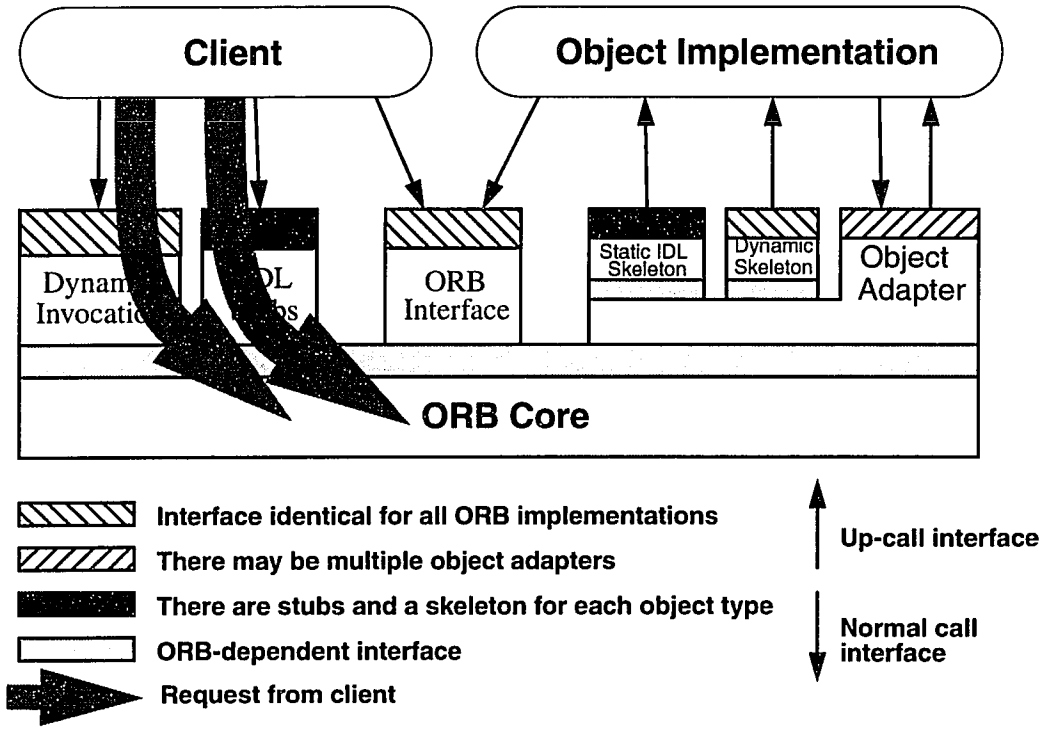

Figure 19: Client requests mechanism on CORBA

Figure 19 shows how clients issue request for object implementation through the Object Request Broker (ORB). Client requests can come either from Dynamic Invocation Interface or from IDL stubs. When ORB receives a request from the client, it will try to locate the requested object implementation through communication with ORBs which are distributed in the network. After the ORB locates the object implementation, the request will pass to the object implementation through Object Adapters where all the Common Object Services are located. We have implemented our Security Service as a common object service registered to the object adapters. It is shown in Figure 20. Therefore, all requests are validated by the security service for their access rights.

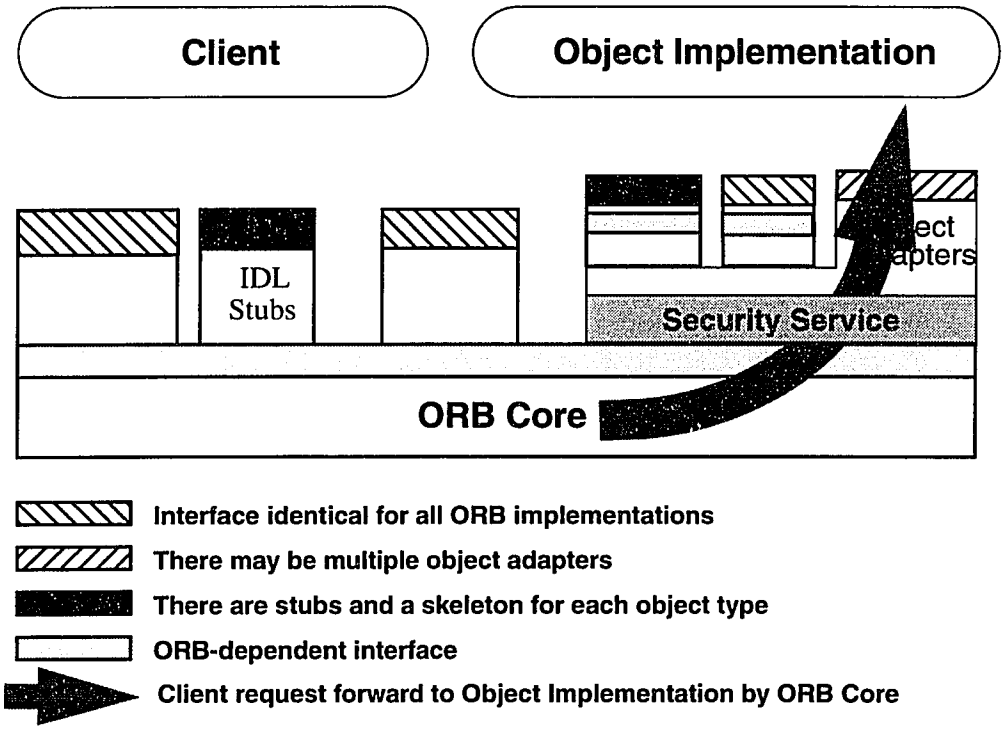

Figure 20: Object implementation receiving request through the Object Adapters

There are different kinds of object adapters for various uses. The Basic Object Adapter (BOA), for example, can be used for most ORB objects with conventional implementations. Our implementation uses the Basic Object Adapter to define our security services. Figure 21 shows the IDL implementation of the security service. In this example, we create a sample server which will be called by clients. The sample server will register itself during the start-up so that The ORB will first locate the host where the sample server is located and then the security services will check the access right of the client for the sample server. Exceptions will be raised if the client permission is denied.

```
struct SecurityAssoc
{
            string      host_name;
            SampleServerserver;
};
interface SecurityMonitor
{

            exception PermissionDenied {};

            SampleServer get_server() raises ( PermissionDenied,);

};
```

Figure 21: IDL Implementation of Security Monitor Service

Figure 22 shows the class definition for the Security Monitor Service. We implemented the SecurityEventHandler as a derived class from the BOA event handler class. SecurityEventHandler redefines the pre_method of the BOA event handler. Pre-method of the event handler will be invoked every time a client performs a method invocation on any method of the object implementation from server. The principal of the client, which includes a user ID and a hostname, the object's interface and object names and environment are passed to this method. Once the SecurityEventHandler is registered

to the BOA, any attempt to access server object implementation without proper security clearance will result in a permission denied exception being raised. Routines for access authorization have been discussed in the previous section.

```
class SecurityEventHandler: public CORBA::BOA::IMPLEventHandler
{
  protected:
    void pre_method (const CORBA::Principal& princ,
                     const char *interface, const
                     CORBA::Object& obj,
                     CORBA::ULong methodid,
                     CORBA::Environment& env);
};
class SecurityMonitorServer: public SecurityMonitor_impl
{
  friend SecurityEventHandler;
  public:
      SecurityMonitorServer(const char *name) :
                  SecurityMonitor_impl(name) {}
      ~SecurityMonitorServer() {}
       SampleServer *get_server(CORBA::Environment& env);
};
```

Figure 22: Class Definition for the Security Monitor Service

The next section is an example showing how to use the security services we designed and implemented to provide authorization controls over distributed object implementations.

4.3 AN APPLICATION EXAMPLE

This is an example to show the usage of our security services to perform access rights authentication. We have created a computational server which will perform a simple binary operation to multiply two real numbers. Clients will send request to obtain the result from this operation through the Object Request Broker (ORB). The ORB will locate the object implementation of the computational server from the network. Our security monitor services described in the previous section provide access rights authorization at the beginning of any request for object implementations.

There are two kinds of access rights authorizations in this example. The first authorization is performed prior to registration of the computational server to the ORB. This authorization provides security control for creating object implementations at the server side. This kind of security control can prevent users from creating insecure object implementations to release system information to unauthorized clients. The second authorization is performed when the client is requesting the computational server. The ORB will pass the client's information to the security services which will then check the inherited role of the client. If the client belongs to a role which has access to the computational server, the ORB will pass the request to the computational server to perform the multiplication. Result will be returned to the client via the ORB. Otherwise, the ORB will inform the client that his/her permission to the computational server is denied. Both authorization tests are handled by the Security Monitor Server.

```
int main(int argc, char **argv)
{
  signal(SIGINT, sighandler);
  MonitorServer server("SecurityMonitor");
  int c;
  CORBA::BOA *boa = CORBA::BOA::instance();
// Create Role graph
  create_rolesgraph();
  CORBA::Environment env;
  boa->event_handler(&server, env,
                     new SecurityEventHandler);
  if ( env.check_exception() ) {
     cout << "Error registering event handler." << endl;
     cout << env;
  }
  CORBA::BOA::impl_is_ready();
  return 0;
}
```

Figure 23: Main function of Security Monitor Server

Figure 23 shows the main function of the Security Monitor Server (SMS) which is implemented using the Basic Object Adapter (BOA). At the beginning, we calls function create_rolesgraph to create a role graph and store it as a persistent object in EOS object store for later use. We then register the SMS to the BOA event handler and create an new security event handler for SMS. The purpose of the SMS is to monitor

various computational server while the security event handler will perform pre-method authorization tests on any client requests. SMS will raise an exception if the authorization test of the security event handler is failed. Otherwise, the ORB will continue to perform object request for the computational server.

```
void SecurityEventHandler::pre_method(const
          CORBA::Principal& princ, ...) {
     CORBA::String name;
     CORBA::String hostname;
     RolesGraph *persistent_rolesgraph;
     CORBA::String priv("Access to comp. server");
     name = princ.userName();
     hostname = princ.hostName();
     cout << "Checking validity for user " << name <<
             " at host " << hostname << endl;
     persistent_rolesgraph = get_rolesgraph();
     if (persistent_rolesgraph->checkSecurity(name,
         hostname, priv))
         cout << "Security test passed!" << endl;
     else
         env.exception_value(new
                     CORBA::StExcep::NO_PERMISSION);
}
```

Figure 24: Class Implementation of the Security Event Handler

Figure 24 shows the class implementation for the security event handler. The

function will obtain the "Prinicipal" of the client information from the ORB. The "Principal" contains username, host name and other information of the client. Security event handler will then call the function "get_rolesgraph" to restore the persistent role graph from the EOS object store. The role graph has been stored in EOS during the initialization of the Security Monitor Server as shown in Figure 23. Finally, it calls the checkSecurity member function of RolesGraph to perform security check on client.

The function checkSecurity is shown in Figure 25. This function will first find out the associated role of the user which is represented by "user_role". It returns error if the user does not associate with any role. Subsequently, we perform security check for the user to see if the user has access rights to create or to use the computational server. The function inUnionPrivset as shown in Figure 25 performs a Breadth First Search (BFS) to all the junior roles of the "user_role" to check if the privilege to create or to use the computational server is belongs to one of its junior roles. If the client pass this final test, its request for the computational server will be proceeded by the ORB. Otherwise, an exception will be raised for denial permission.

This example shows the use of our security services to provide access rights authorization. The Monitor Security Server and the Security Event Handler used in this application can be reused for other distributed object applications using CORBA. The full implementation for role graph management, persistent object service, secure object services and the application example are provided in Appendix.

```
CORBA::Boolean RolesGraph::checkSecurity(CORBA::String& login,
                        CORBA::String& ip, CORBA::String& privID){
    int i; long rid=0; Role *user_role;
    for (i = 0; i < _userinfo.length(); i++) {
        if ((_userinfo[i].login() == login) &&
            (_userinfo[i].ip() == ip)) {
            cout << "Username " << login << " exists." << endl;
            rid = _userinfo[i].rid_long();
            break;
        }
    }
    if (!rid) {
        cout << "Can't find security information of user " <<
                login << endl;
        return FALSE;
    }
    user_role = _node_ptr[rid];
    if (!user_role->canAccess(login, ip) return FALSE;
    if (user_role->inUnionPrivset(login, ip, privID)) {
        cout << "User " << login <<
          " pass security clearance for " << privID << endl;
        return TRUE;
    }
    return FALSE;
}
```

Figure 25: Function to perform security check for a user

4.4 SUMMARY

We have presented our design of the role-based security model and its implementation. We have also integrated the Security Service to CORBA. CORBA provides a nice communication layer for distributed client and server application. We find it very easy to implement distributed applications. Next chapter is a conclusion for this thesis. We shall discuss our implementation which concerns CORBA and suggest possible extension on CORBA and our security model.

CHAPTER 5

CONCLUDING REMARKS

5.1 SUMMARY ON PERSISTENT AND SECURITY MODELS

Our design of persistent model is based on some existing models and can provide the need to make our role graph persistence. Our security model, on the other hand, is a new attempt to use an object model to handle security. The model is implemented in C++ classes, but can be easily implemented on other non object-oriented languages. Security is still a missing component from CORBA specification 2.0 [40]. The Object Management Group has been requesting for a proposal for adding a security service to CORBA. All existing CORBA implementations rely on the system level security as the underlying security mechanism. Object entities contain real life relationship which is difficult to handle by general security models. The security prototype we have designed and implemented can not only handle object entities but also be extended to meet the multilevel security requirement. We can also extend our model to provide data encryption during the message passing between the client and server through the ORBs and the Security Services.

There are limitations in our security model, but they can be easily fixed. For

example, the class of role and role graph make usage of several pointer references to role objects and its private members. It is possible for computer hackers to obtain references to these pointers and hence creating security holes.

5.2 Implementation Issues on CORBA

CORBA is going to be a standard system for building distributed application in the future. It is not difficult to build applications on top of CORBA. However, CORBA IDL provides limited data types which limits the functionality of the IDL interface. We have implemented our Security Service on a heterogeneous network on coupled Sun Sparc stations running Solaris 2.4, Solaris 2.5 and SunOS 4.1.4. The Object Request Brokers can communicate efficiently to locate the desired object method across the network.

5.3 Future Work

There are still some concerns about the role of CORBA within the Object-Oriented Database Management System (OODBMS). ODBMS nowadays is required to support millions of fine-grained objects. Fast and efficient access to the objects is almost required by all applications. The role of CORBA is to provide a more efficient mechanism to handle millions of distributed objects located across the internet. There are more issues to be considered such as how to provide locking transaction through CORBA between distributed data stores.

Other issues such as interoperability among various ORBs from different vendors are in the final testing process. IBM SOM, Expersoft, Orbix, Sunsoft NEO and other

CORBA implementors are going to release their interoperability versions in few months. Sunsoft, PostModern Computing and Iona have announced their Java front-end for their CORBA products which will allow internet access to object methods through CORBA. In the near future, CORBA will be the underlying layer for most of the distributed applications.

APPENDIX I

PERSISTENT IDENTIFIER PROGRAM

```
#ifndef _posPID_h
#define _posPID_h

class PosPID: public CosPersPID_impl::PID_impl
{
private:
  CORBA::String _datastore_type;
  CORBA::String _id;
  CORBA::String _ip;
  CORBA::String *_pid;

public:
  PosPID(const char *datastore, const char *id, const char *ip)
    : _datastore_type(datastore), _id(id), _ip(ip), _pid(NULL)
      {}

  ~PosPID() {}

  CORBA::String* datastore_type();
  void datastore_type(const CORBA::String& val);
  CORBA::String* get_PIDString();
};

#endif
```

```
#include <stdlib.h>
#include <string.h>
#include "CosPersPID_s.hh"
#include "posPID.h"

CORBA::String* PosPID::datastore_type()
{
  return &_datastore_type;
}

void PosPID::datastore_type(const CORBA::String& val)
{
  _datastore_type = val;
}

CORBA::String* PosPID::get_PIDString()
{
  int i;
  char newstring[255];

  for (i = 0; i < _ip.length(); i++)
    newstring[i] = _ip[i];
  newstring[i] = '\0';
  strcat (newstring, ":");
  strcat (newstring, _id);

  _pid = new CORBA::String(newstring);
  return _pid;
}
```

APPENDIX II

PERSISTENT OBJECT SERVICE PROGRAM

```
#ifndef _posPDS_h
#define _posPDS_h
#include "eos.h"

class PosPDS: public CosPersPDS_impl::PDS_impl
{
private:

  // Persistence related data

  CosPersPDS::Object *_obj_data;
  CosPersPID::PID *_pid;
  unsigned _refcount;
  CORBA::String _datastore_type;
  long _obj_size;

  // Datastore (EOS) related data

  eosdatabase *_eosdb;          // database descriptor
  eosobj *_eosoh;               // object handle
  eosfile *_eosfh;              // file handle
  eosoid _eosoid;               // object id

  CORBA::String _db_name;
  CORBA::String *_fname;

  int trans();
  int createfile();
  int openfile();
  int closefile();
  int commit();
  long getobjsize();

public:

  PosPDS(CosPersPDS::Object* obj, CosPersPID::PID* p)
    : _obj_data(obj), _pid(p), _refcount(0), _fname(p-
>get_PIDString()) {
```

```
      _db_name = "/tmp/eos_area/roledatabase";
      _datastore_type = "EOS.2.0.2";
    }
  ~PosPDS();

  CosPersPDS::PDS* connect(CosPersPDS::Object* obj, CosPersPID::PID* p);
  void disconnect(CosPersPDS::Object* obj, CosPersPID::PID* p);
  void store(CosPersPDS::Object* obj, CosPersPID::PID* p);
  CosPersPDS::Object* restore(CosPersPDS::Object* obj,
CosPersPID::PID* p);
  void remove(CosPersPDS::Object* obj, CosPersPID::PID* p);
  void setobjsize(long objsize);
  void persRef(CosPersPDS::Object* obj1, CosPersPID::PID* p1, CosPer-
sPDS::Object* obj2, CosPersPID::PID* p2);
};

#endif
```

```
#include <stdlib.h>
#include <string.h>
#include "CosPersPDS_s.hh"
#include "posPDS.h"

char *stringtochars(char *outchars, CORBA::String *str)
{
  int i;
  CORBA::String pidstring;

  pidstring = *str;
  outchars = (char *) malloc(sizeof(char)*(str->length() + 1));
  for (i = 0; i < str->length(); i++) {
    outchars[i] = pidstring[i];
  }
  outchars[i] = '\0';
  return outchars;
}

PosPDS::~PosPDS()
{
  _obj_data = NULL;
  delete [] _fname;
  delete [] _eosdb;
  delete [] _eosoh;
  delete [] _eosfh;
}

CosPersPDS::PDS* PosPDS::connect(CosPersPDS::Object* obj, CosPer-
sPID::PID* p)
{
  cout << "_datastore_type " << _datastore_type << "  p-
>datastore_type "
    << *p->datastore_type() << endl;
  if (strcmp(_datastore_type, *p->datastore_type()) != 0) {
    cerr << "Wrong datastore type!!" << endl;
    exit(-2);
  }

  if (_refcount >= 2) {
    disconnect(_obj_data, _pid);
  }

// connect the object to its persistent state
  if ((_eosdb = eosdatabase::open(_db_name, 0, 1, 0)) == NULL) {
    cerr << "Cannot create database " << _db_name << endl;
    return NULL;
  } else if (trans() == 0) {
    cout << "trans" << endl;
    return NULL;
  }
  _refcount++;
  return this;
}
```

```
int PosPDS::trans()
{
  if (eostrans::begin(0) != 0) {
    cerr << "Cannot start transaction" << endl;
    return 0;          // Failure
  } else
    return 1;          // Sucess
}

int PosPDS::createfile()
{
  if ((_eosfh = eosfile::create(_eosdb, *_fname)) == NULL) {
    cerr << "Cannot create file " << *_fname << endl;
    return 0;
  } else
    return 1;
}

int PosPDS::openfile()
{
  char *fnameChars;
  CORBA::String tmpstring;
  int i;

  tmpstring = *_fname;
  fnameChars = (char *) malloc(sizeof(char)*(_fname->length() + 1));
  for (i = 0; i < _fname->length(); i++)
    fnameChars[i] = tmpstring[i];
  fnameChars[i] = '\0';
  if ((_eosfh = eosfile::open(_eosdb, fnameChars)) == NULL) {
    cerr << "Cannot open file " << fnameChars << endl;
    return 0;
  } else
    return 1;
}

int PosPDS::closefile()
{
  if (_eosfh->close() != 0) {
    cerr << "Cannot close file" << endl;
    return 0;
  } else
    return 1;
}

int PosPDS::commit()
{
  if (eostrans::commit() != 0) {
    cerr << "Cannot commit transaction" << endl;
    return 0;
  } else
    return 1;
}
```

```
void PosPDS::disconnect(CosPersPDS::Object* obj, CosPersPID::PID* p)
{
  if (_refcount < 1) {
    cerr << "Cannot disconnect " << (char *) p->get_PIDString() << endl
      << "_refcount = " << _refcount << endl;

    exit (-2);
  }
  if (_eosdb->close() != 0) {
    cerr << "Cannot close database" << endl;
    exit (-2);
  } else {
    _eosdb = NULL;
    _eosoh = NULL;
    _eosfh = NULL;
    _pid = NULL;
    _refcount--;
  }
}

void PosPDS::store(CosPersPDS::Object* obj, CosPersPID::PID* p)
{
  // save the persistent state of an object

  eosobj *eosoh;                // object handle
  char *pidchars;
  long strlength;

  // Add routine to distinguish update and create later

  // Preliminary verion: have not yet taken into account of concurrency
  // control

  if ((eosoh = eosobj::create((int) getobjsize(), _eosdb, obj)) == NULL)
{
    cerr << "Cannot create persistent object" << endl;
    exit(-2);
  }

  _eosoid = eosoh->oid();
  pidchars = stringtochars(pidchars, p->get_PIDString());
  printf ("pidchars = %s\n", pidchars);
  eosoid tmpoid;
  eos_Ref_Any objRef;
  if ((tmpoid = _eosdb->oid_of(pidchars)) == eosoid::null)
    cout << pidchars << " not exist" << endl;
  if (eosoh->name_set((const char *) pidchars) != 0)
    cerr << "Cannot set object name" << endl;
  cout << "Object name is " << eosoh->name() << endl;
  if (eosoh->release() != 0) {
    cerr << "Cannot release object handle" << endl;
    exit(-2);
  }
```

```
  if (PosPDS::commit() == 0)
    cerr << "Cannot commit transaction" << endl;
}

CosPersPDS::Object* PosPDS::restore(CosPersPDS::Object* obj, CosPer-
sPID::PID* p)
{
  // loads the object's persistent state unless a store or other
  // mutating operation is performed on the persistent state

  eosfilescan *eosfs;          // filescan handle
  char *pidchars;

   pidchars = stringtochars(pidchars, p->get_PIDString());

  if ((_eosoh = eosobj::get(_eosdb, pidchars, eosobj::HDR_ONLY)) ==
NULL) {
    cerr << "Cannot get object handle" << endl;
    return NULL;
  }

  _obj_data = (CosPersPDS::Object *) _eosoh->mptr();
  if (_eosoh->release() != 0) {
    cerr << "Cannot release object" << endl;
    exit (-2);
  }
  if (commit() == 0)
    cerr << "Cannot commit transaction" << endl;
  return _obj_data;
}

void PosPDS::remove(CosPersPDS::Object* obj, CosPersPID::PID* p)
{
  // delete the object's persistent data from the datastore indicated
  // by the PID.

  // Preliminary verion: have not taken into account of concurrency
  // control

  eosobj *oh;
  char *pidchars;

  pidchars = stringtochars(pidchars, p->get_PIDString());
  if ((oh = eosobj::get(_eosdb, pidchars, eosobj::HDR_ONLY)) == NULL) {
    cerr << "Cannot get object handle" << endl;
    exit(-2);
  }
  if (oh->destroy() != 0) {
    cerr << "Cannot destroy object" << endl;
    exit(-2);
  }
  if (commit() == 0)
    cerr << "Cannot commit transaction" << endl;
}
```

```
void PosPDS::setobjsize(long objsize)
{
  _obj_size = objsize;
}

long PosPDS::getobjsize()
{
  return _obj_size;
}

void PosPDS::persRef(CosPersPDS::Object* obj1, CosPersPID::PID* p1,
                CosPersPDS::Object* obj2, CosPersPID::PID* p2)
{
  // save the persistent state of an object

  eosobj *eosoh;               // object handle
  char *pidchars;
  long strlength;
  eos_Ref_Any *persobj;
  eosoid tmpoid;

   pidchars = stringtochars(pidchars, p1->get_PIDString());
  *persobj = _eosdb->lookup_object(pidchars);

  if (PosPDS::commit() == 0)
    cerr << "Cannot commit transaction" << endl;
}
```

APPENDIX III

ROLE IDENTIFIER PROGRAM

```
#ifndef _RID_h
#define _RID_h

#include "misc.h"

class RID: public CosRID_impl::idlRID_impl
{

private:
  long           _id;
  CORBA::String _ip;
  CORBA::String _name;

public:
   RID(const char *name, const char *ip);

  ~RID() {}

  RID& operator=(RID &rid);
  CORBA::String* get_RIDstring();
  long get_RIDlong();
  CORBA::String* get_RnameString();
};

#endif
```

```
#include <stdlib.h>
#include <string.h>
#include "../persist_store/CosPersPID_s.hh"
#include "../persist_store/posPID.h"
#include "CosRID_s.hh"
#include "misc.h"
#include "RID.h"

RID::RID(const char *name, const char *ip)
{
  char ridname[255];
  int thisid;

  _ip = ip;
  _name = name;
  strcpy(ridname, name);
  _id = hash(ridname);
}

RID& RID::operator=(RID &rid)
{
    _ip = rid._ip;
    _name = rid._name;
    _id = rid._id;

    return *this;
}

CORBA::String* RID::get_RIDstring()
{
  CORBA::String *rid;
  char newstring[255];

  itoa(_id, newstring);

  rid = new CORBA::String(newstring);
  return rid;
}

long RID::get_RIDlong()
{
  return _id;
}

CORBA::String* RID::get_RnameString()
{
  CORBA::String *rid=new CORBA::String(_name);

  return rid;
}
```

APPENDIX IV

ROLE GRAPH PROGRAM

```
#ifndef _Roles_h
#define _Roles_h

#include <seqmac.h>
#include "CosRoles_c.hh"
#include "RID.h"

enum {FALSE, TRUE};
enum STATUS {ALLOWED, DENIALED, UNKNOWN};

class Role;

class Item {
    friend class List;
    friend class Role;
    friend class RolesGraph;
private:
    Role *val;
    Item *next;
    Item(Role *value, Item *item = 0)
    {
        val = value;
        next = item;
    }
};

class List
{
public:
    List ()
    {
        list = 0;
        at_end = 0;
        current = 0;
        _length = 0;
    }
    ~List () { remove(); }
    CORBA::Boolean append(Role* node); // TRUE if append is success
```

```
    // FALSE if node already exists
    Role *iterator(); // Return first item value on the list
    CORBA::Boolean remove(Role* node); // TRUE if node is deleted
    // FALSE if node doesn't exist
    // or list is empty
    Role *remove_first();              // remove first role
    List& operator=(List &l);
    void remove(); // Remove all items
    CORBA::Boolean is_present(Role *node);
    CORBA::Boolean is_empty();
    void reset_current(); // Reset current pointer
    long length();

    void display();
private:
    Item *list;
    Item *at_end;
    Item *current;
    long _length;
};

class Role: public CosRoles_impl::idlRole_impl
{

private:

    CosRID::idlRID *_rid;
    CosRoles::seqId _privset;
    CosRoles::seqId _union_privset;
    CosRoles::seqAuth _accessList;
    CosRoles::seqAuth _denialList;
    CosRoles::seqId *getPrivset();
    CosRoles::seqAuth *getAccessList();
    CosRoles::seqAuth *getDenialList();

public:

    Role(CosRID::idlRID *rid, CosRoles::seqId privset,
         CosRoles::seqAuth accessList,
         CosRoles::seqAuth denialList, List *juniorList)
    : _rid(rid), _privset(privset), _accessList(accessList),
      _denialList(denialList), _juniorList(juniorList)
    {};

    Role() {};

    ~Role();

    Role& operator=(Role &r);
    // Check if this login with this ip can access this role or not.
    CORBA::Boolean canAccess(const CORBA::String& login,
                             const CORBA::String& ip);
    CORBA::Boolean inPrivatePrivset(const CORBA::String& privID);
    CORBA::Boolean inUnionPrivset(const CORBA::String& login,
```

```
                                    const CORBA::String& ip,
                                    const CORBA::String& privID);
    CosRID::idlRID *getRIDp();

protected:

    friend class RolesGraph;

    List *_juniorList;
    long childCount() const;
    CORBA::Boolean addJunior(Role* node);
    CORBA::Boolean removeJunior(Role* node);
    CORBA::Boolean addPriv(const CORBA::String& privID);
    CORBA::Boolean isPrivJunior(CosRID::idlRID *rid);
};

class RolesGraph : public CosRoles_impl::idlRolesGraph_impl
{
public:

    RolesGraph();           // Default constructor: build an empty
role graph

    RolesGraph(Role *root); // build a graph with root

    ~RolesGraph();

    RolesGraph& operator=(RolesGraph& rg);
    Role* Root() const;     // return pointer to the root if authorized

    Role* Node (RID rid);    // return pointer to the node with the rid

    CORBA::Boolean isJunior(CosRID::idlRID *rid1, CosRID::idlRID
*rid2) const;
    // return TRUE if role with rid1 is a junior
    // role of the role with rid2

    CORBA::Boolean isCycle(List *seniorList, Role* role);
    // return TRUE if cycle exists after adding role
    // return FALSE otherwise

    RolesGraph* removeRedundantPaths(); // Optional optimization routine

    CosRoles::idlRolesGraph* remove(CosRID::idlRID* rid);

    RolesGraph* add(List *seniorList, Role* role);


    RolesGraph* update(List *seniorList, Role* new_role,
                       CosRID::idlRID* rid);

    CORBA::Boolean addUser(CORBA::String& login, CORBA::String& ip,
                           RID *rid);
```

```
    CORBA::Boolean delUser(CORBA::String& login, CORBA::String& ip);

    CORBA::Boolean checkSecurity(CORBA::String& login, CORBA::String&
ip,
                                 CORBA::String& privID);

private:

    displayGraph();

    Role* _node_ptr[MAXTABLE+1];

    Role* _root;
    CORBA::Boolean isCycle(Role* role); // Check if this role is in a
cycle.
    CosRoles::seqUserInfo _userinfo;
};

#endif
```

```
#include <stdlib.h>
#include <string.h>
#include "CosRoles_s.hh"
#include "RID.h"
#include "Roles.h"
#include "misc.h"

CORBA::Boolean List::is_empty()
{
  return list == 0 ? TRUE : FALSE;
}

CORBA::Boolean List::append(Role *val)
{
  Item *pt = new Item(val);
  Item *iter_list = list;
  if (list == 0) {
    list = new Item(val);
    list->next = 0;
    at_end = list;
  }
  else {
    while (iter_list) {
      if (iter_list->val == val)
      return FALSE;
      iter_list = iter_list->next;
    }
    at_end->next = pt;
    at_end = pt;
  }
  _length++;
  return TRUE;
}

void List::display()
{
  for (Item *pt = list; pt; pt = pt->next)
    cout << *(pt->val->getRIDp()->get_RIDstring()) << " ";
  cout << endl;
}

Role *List::remove_first()
{
  Role *tmprole;

  if (list == 0)
    return 0;
  else {
    tmprole = list->val;
    list = list->next;
    _length--;
  }
  return tmprole;
}
```

```
void List::remove()
{
  Item *pt = list;
  while (pt) {
    Item *tmp = pt;
    pt = pt->next;
  }
  list = at_end = current = 0;
  _length = 0;
}

CORBA::Boolean List::is_present(Role *item) {
  if (list == 0)
    return FALSE;
  if (list->val == item || at_end->val == item)
    return TRUE;
  Item *pt = list->next;
  for (; pt != at_end; pt = pt->next)
    if (pt->val == item)
      return TRUE;
  return FALSE;
}

CORBA::Boolean List::remove(Role *val)
{
  Item *pt = list;
  Role *del_role;

  if (pt && pt->val == val) {
    Item *tmp = pt->next;
    del_role = pt->val;
    list = tmp;
    _length--; // _length should be zero after
    return TRUE;
  }
  if (_length == 0) {
    return FALSE;
  }
  Item *prev = pt;
  pt = pt->next;
  while (pt) {
    if (pt->val == val) {
      prev->next = pt->next;
      if (at_end == pt)
      at_end = prev;
      del_role = pt->val;
      pt = prev->next;
      _length--;
      return TRUE;
   }
    else {
      prev = pt;
      pt = pt->next;
```

```
    }
  }
  return FALSE;
}

Role *List::iterator()
{
  Role* tmprole;

  if (is_empty()) {
    cout << "In iterator: list is empty." << endl;
    return 0;
  }
  else if (!current) {
    return 0;
  }
  else if (current == at_end || at_end == list) {
    tmprole = current->val;
    current = 0;
    return tmprole;
  }
  tmprole = current->val;
  current = current->next;
  return tmprole;
}

void List::reset_current()
{
  if (!is_empty())
    current = list;
}

List& List::operator=(List &l)
{
  Item *tmpitem;
  Item *listiterator;

  listiterator = l.list;
  remove();
  while(tmpitem = listiterator) {
    append(tmpitem->val);
    listiterator = listiterator->next;
  }
  return *this;
}

long List::length()
{
  return _length;
}

Role::~Role()
{
  CORBA::ULong i;
```

```
  for (i = 0; i < _accessList.length(); i++)
    _privset.remove(i);
  for (i = 0; i < _accessList.length(); i++)
    _accessList.remove(i);
  for (i = 0; i < _denialList.length(); i++)
    _denialList.remove(i);
}

Role& Role::operator=(Role &role)
{
    _rid = role._rid;
    _privset = role._privset;
    _union_privset = role._union_privset;
    _accessList = role._accessList;
    _denialList = role._denialList;

    return *this;
}

CosRoles::seqId *Role::getPrivset()
{
  return &_privset;
}

CosRoles::seqAuth *Role::getAccessList()
{
  return &_accessList;
}

CosRoles::seqAuth *Role::getDenialList()
{
  return &_denialList;
}

CosRID::idlRID *Role::getRIDp()
{
  return _rid;
}

CORBA::Boolean Role::inPrivatePrivset(const CORBA::String& privID)
{
  for (int i = 0; i < _privset.length(); i++)
    if (privID == _privset[i])
      return TRUE;
  return FALSE;
}

CORBA::Boolean Role::inUnionPrivset(const CORBA::String& login,
                            const CORBA::String& ip,
                            const CORBA::String& privID)
{
  long toprid;
  Role *cur_role, *junior_role;
```

```
  List *queue = new List;
  CosRoles::auth_para auth_parameters;
  STATUS visited[MAXTABLE];

  if(canAccess(login, ip)) {
    if (inPrivatePrivset(privID)) {
      return TRUE;
    }
  }
  else {
    cout << login << " at " << ip << " can not access this role." <<
endl;
    return FALSE;
  }
  cout << "pass current role test" << endl;
  toprid = getRIDp()->get_RIDlong();
  cout << "rid is " << toprid << endl;
  for (int i = 0; i < MAXTABLE; i++)
    visited[i] = UNKNOWN;
  cur_role = this;
  visited[toprid] = ALLOWED;
  queue->append(cur_role);
  while (!queue->is_empty()) {
    cur_role = queue->remove_first();
    cur_role->_juniorList->display();
    cur_role->_juniorList->reset_current();
    while (junior_role = cur_role->_juniorList->iterator()) {
      toprid = junior_role->getRIDp()->get_RIDlong();
      if (visited[toprid] == UNKNOWN) {
      queue->append(junior_role);

      // Routine to check authority and to gather whole privilege set

        for (int i = 0; i < junior_role->getDenialList()->length() ;
i++) {
        CosRoles::seqAuth *denial_list;
        denial_list = junior_role->getDenialList();
        auth_parameters = (*denial_list)[i];
        if ((auth_parameters.login() == login) &&
            (auth_parameters.ip() == ip)) {
          visited[toprid] = DENIALED;
        }
      }
      if (visited[toprid] == UNKNOWN) {
        for (int i = 0; i < junior_role->getAccessList()->length(); i++)
{
          CosRoles::seqAuth *access_list;
          access_list = junior_role->getAccessList();
          auth_parameters = (*access_list)[i];
          if ((auth_parameters.login() == login) &&
            (auth_parameters.ip() == ip)) {
            visited[toprid] = ALLOWED;
            if (junior_role->inPrivatePrivset(privID) == TRUE)
            return TRUE;
```

```
          }
        }
      }
      if (visited[toprid] == UNKNOWN) {
        if (junior_role->inPrivatePrivset(privID) == TRUE) {
          cout << "inherit access allowed" << endl;
          return TRUE;
        }
      }
      }
    }
    cur_role->_juniorList->reset_current();
  }
  delete [] visited;
  return FALSE;
}

CORBA::Boolean Role::canAccess(const CORBA::String& login,
                  const CORBA::String& ip)
{
  Role *junior_role;
  CosRoles::auth_para auth_parameters;
  int i;

  junior_role = this;
  for (i = 0; i < junior_role->getDenialList()->length() ; i++) {
    CosRoles::seqAuth *denial_list;
    denial_list = junior_role->getDenialList();
    auth_parameters = (*denial_list)[i];
    if ((auth_parameters.login() == login) &&
      (auth_parameters.ip() == ip)) {
      return FALSE;
    }
  }
  for (i = 0; i < junior_role->getAccessList()->length(); i++) {
      CosRoles::seqAuth *access_list;
      access_list = junior_role->getAccessList();
      auth_parameters = (*access_list)[i];
      if ((auth_parameters.login() == login) &&
          (auth_parameters.ip() == ip)) {
      return TRUE;
    }
  }
  return FALSE;
}

CORBA::Boolean Role::isPrivJunior(CosRID::idlRID *rid)
{
  Role *junior_role, *cur_role;

  cur_role = this;
  cur_role->_juniorList->reset_current();
  while (junior_role = cur_role->_juniorList->iterator()) {
    if (rid->get_RIDlong() == junior_role->getRIDp()->get_RIDlong()) {
```

```
      cur_role->_juniorList->reset_current();
      return TRUE;
    }
  }
  return FALSE;
}

long Role::childCount() const
{
  return _juniorList->length();
}

CORBA::Boolean Role::addJunior(Role* node)
{
  return (_juniorList->append(node));
}

CORBA::Boolean Role::removeJunior(Role* node)
{
  return (_juniorList->remove(node));
}

CORBA::Boolean Role::addPriv(const CORBA::String& privID)
{
  if (!inPrivatePrivset(privID)) {
    _privset.append(privID);
    return TRUE;
  }
  else
    return FALSE;
}

RolesGraph::RolesGraph()
{
  _root = 0;
  for (int i=0; i < MAXTABLE; i++)
    _node_ptr[i] = 0;
}

RolesGraph::RolesGraph(Role *root)
{
  long toprid;
  Role *cur_role, *junior_role;
  CORBA::Boolean visited[MAXTABLE];
  List *queue = new List;
  int i;

  for (i=0; i < MAXTABLE; i++)
    _node_ptr[i] = 0;

  _root = root;

  toprid = root->getRIDp()->get_RIDlong();
  _node_ptr[toprid] = root;
```

```
  for (i = 0; i < MAXTABLE; i++)
    visited[i] = FALSE;
  cur_role = root;
  visited[toprid] = TRUE;
  queue->append(cur_role);
  while (!queue->is_empty()) {
    cur_role = queue->remove_first();
    cur_role->_juniorList->reset_current();
    while (junior_role = cur_role->_juniorList->iterator()) {
      toprid = junior_role->getRIDp()->get_RIDlong();
      if (visited[toprid] == FALSE) {
      queue->append(junior_role);
      visited[toprid] = TRUE;
      _node_ptr[toprid] = junior_role;
      }
    }
    cur_role->_juniorList->reset_current();
  }
  delete [] visited;
}

RolesGraph::~RolesGraph()
{
  delete _root;
  delete [] _node_ptr;
}

RolesGraph& RolesGraph::operator=(RolesGraph &rg)
{
    int i;

    for (i = 0; i < MAXTABLE; i++)
        _node_ptr[i] = rg._node_ptr[i];
    _root = rg._root;

    return *this;
}

Role *RolesGraph::Root() const
{
  return _root;
}

Role *RolesGraph::Node(RID rid)
{
  if (_node_ptr[rid.get_RIDlong()]) {
    return _node_ptr[rid.get_RIDlong()];
  }
  else
    return 0;
}

CORBA::Boolean RolesGraph::isJunior(CosRID::idlRID *junior_rid,
```

```
                                CosRID::idlRID *senior_rid) const
{
  long toprid;
  Role *cur_role, *junior_role;
  List *queue = new List;

  CORBA::Boolean visited[MAXTABLE];

  toprid = senior_rid->get_RIDlong();
  if (_node_ptr[toprid] == 0) // Junior role doesn't exist
    return FALSE;

  for (int i = 0; i < MAXTABLE; i++)
    visited[i] = FALSE;
  cur_role = _node_ptr[toprid];
  visited[toprid] = TRUE;
  queue->append(cur_role);
  while (!queue->is_empty()) {
    cur_role = queue->remove_first();
    cur_role->_juniorList->reset_current();
    while (junior_role = cur_role->_juniorList->iterator()) {
      toprid = junior_role->getRIDp()->get_RIDlong();
      if (visited[toprid] == FALSE) {
      queue->append(junior_role);
      visited[toprid] = TRUE;
      if (junior_rid->get_RIDlong() == toprid) {
        cur_role->_juniorList->reset_current();
        delete [] visited;
        return TRUE;
      }
      }
    }
    cur_role->_juniorList->reset_current();
  }
  delete [] visited;
  return FALSE;
}

// Perform BFS starting from role to look for cycle.

CORBA::Boolean RolesGraph::isCycle(Role *role)
{
  long thisrid, toprid;
  Role *cur_role, *junior_role;
  CORBA::Boolean visited[MAXTABLE];
  List *queue = new List;

  thisrid = role->getRIDp()->get_RIDlong();
  for (int i = 0; i < MAXTABLE; i++)
    visited[i] = FALSE;
  cur_role = role;
  visited[thisrid] = FALSE; // Allow access twice
  queue->append(cur_role);
  while (!queue->is_empty()) {
```

```
    cur_role = queue->remove_first();
    cur_role->_juniorList->reset_current(); // double insured
    while (junior_role = cur_role->_juniorList->iterator()) {
      toprid = junior_role->getRIDp()->get_RIDlong();
      if (visited[toprid] == FALSE) {
      queue->append(junior_role);
      visited[toprid] = TRUE;
      if (toprid == thisrid)
        return TRUE;
      }
    }
    cur_role->_juniorList->reset_current();
  }
  delete [] visited;
  return FALSE;
}

// Perform BFS on every roles with the graph to check if there exists
cycle.

CORBA::Boolean RolesGraph::isCycle(List *seniorList, Role *role)
{
  List *added_seniorlist = new List;
  Role *cur_role;
  CORBA::Boolean visited[MAXTABLE];
  CORBA::Boolean role_added[MAXTABLE];
  CORBA::Boolean ret;
  int i;

  for (i = 0; i < MAXTABLE; i++) {
    visited[i] = FALSE;
    role_added[i] = FALSE;
  }
  seniorList->reset_current();
  while (cur_role = seniorList->iterator()) {
    if (added_seniorlist->append(cur_role) == FALSE) {
      added_seniorlist->remove(cur_role);
    }
    if (cur_role->addJunior(role) == TRUE) {
      role_added[cur_role->getRIDp()->get_RIDlong()] = TRUE;
    }
  }
  ret = FALSE;
  for (i = 0; i <= MAXTABLE; i++) {
    if (cur_role = _node_ptr[i]) {
      cout << "isCycle testing " << *(cur_role->getRIDp()-
>get_RnameString())
      << " with RID " << i << endl;
      if (isCycle(cur_role)) {
      cout << " forms cycle" << endl;
      ret = TRUE;
      break;
      }
      else
```

```
      cout << " doesn't form cycle" << endl;
    }
  }
  added_seniorlist->reset_current();
  while (cur_role = added_seniorlist->iterator()) {
    if (role_added[cur_role->getRIDp()->get_RIDlong()] == TRUE)
      cur_role->removeJunior(role);
  }
  delete [] visited;
  delete [] role_added;
  delete added_seniorlist;
  return ret;
}

CosRoles::idlRolesGraph* RolesGraph::remove(CosRID::idlRID* rid)
{
  long thisrid, toprid;
  Role *cur_role, *junior_role, *del_role, *add_role;
  CORBA::Boolean visited[MAXTABLE];
  List *queue = new List;
  CORBA::Boolean done;

  del_role = _node_ptr[rid->get_RIDlong()];
  thisrid = _root->getRIDp()->get_RIDlong();
  for (int i = 0; i < MAXTABLE; i++)
    visited[i] = FALSE;
  cur_role = _root;
  visited[thisrid] = TRUE;
  queue->append(cur_role);
  done = FALSE;
  while (!queue->is_empty()) {
    cur_role = queue->remove_first();
    if (cur_role->isPrivJunior(rid)) {
      cout << *(rid->get_RnameString()) << " is a private junior of "
      << *(cur_role->getRIDp()->get_RnameString()) << endl;
      cur_role->_juniorList->display();
      if (cur_role->removeJunior(del_role)) {
      del_role->_juniorList->reset_current();
      while (add_role = del_role->_juniorList->iterator())
        cur_role->addJunior(add_role);
      delete del_role;
      }
      cur_role->_juniorList->display();
      done = TRUE;
    }
    else
      done = FALSE;
    cur_role->_juniorList->reset_current();
    while (!done && (junior_role = cur_role->_juniorList->iterator())) {
      toprid = junior_role->getRIDp()->get_RIDlong();
      if (visited[toprid] == FALSE) {
        queue->append(junior_role);
        visited[toprid] = TRUE;
        if (junior_role->isPrivJunior(rid)) {
```

```
        if (junior_role->removeJunior(del_role)) {
          del_role->_juniorList->reset_current();
          while (add_role = del_role->_juniorList->iterator())
            junior_role->addJunior(add_role);
          delete del_role;
        }
        else {
          cout << "Error: removing junior role" << endl;
          return 0;
        }
      }
      }
    }
    cur_role->_juniorList->reset_current();
  }
  _node_ptr[rid->get_RIDlong()] = 0;
  delete [] visited;
  return this;
}

// Add new role to role graph.
// Remove redundant paths.

RolesGraph* RolesGraph::add(List *seniorList, Role* role)
{
  Role *senior_role, *add_role, *junior_role;
  CosRID::idlRID *add_rid;

  if (isCycle(seniorList, role))
    return 0;
  cout << "passed cycle test" << endl;
  if (_node_ptr[role->getRIDp()->get_RIDlong()]) { // Should not happen
    cout << "Error: RID already exists" << endl;
    return 0;
  }
  else
    _node_ptr[role->getRIDp()->get_RIDlong()] = role;
  role->_juniorList->reset_current();
  while (junior_role = role->_juniorList->iterator()) {
    if (_node_ptr[junior_role->getRIDp()->get_RIDlong()] == 0)
      _node_ptr[junior_role->getRIDp()->get_RIDlong()] = junior_role;
    else if (_node_ptr[junior_role->getRIDp()->get_RIDlong()] !=
junior_role) {
      cout << "Inconsistent node ptrs!" << endl;
      exit(0);
    }
  }
  seniorList->reset_current();
  while (senior_role = seniorList->iterator()) {
    role->_juniorList->reset_current();
    // Remove immediate redundant paths
    while (add_role = role->_juniorList->iterator()) {
      if (senior_role->isPrivJunior(add_role->getRIDp()))
      senior_role->_juniorList->remove(add_role);
```

```
    }
    senior_role->addJunior(role);
  }
  return this;
}

RolesGraph* RolesGraph::update(List *seniorList, Role* new_role,
               CosRID::idlRID* oldrid)
{
  if (oldrid->get_RIDlong() != new_role->getRIDp()->get_RIDlong())
    cout << "Warning: RIDs are different!" << endl;
  RolesGraph::remove(oldrid);
  RolesGraph::add(seniorList, new_role);
  return this;
}

// Display the whole role graph

RolesGraph::displayGraph()
{
  long thisrid;
  Role *cur_role, *junior_role;
  CORBA::Boolean visited[MAXTABLE];
  List *queue = new List;

  thisrid = _root->getRIDp()->get_RIDlong();
  cout << "In displayGraph: root is " << thisrid << endl;
  for (int i = 0; i < MAXTABLE; i++)
    visited[i] = FALSE;
  cur_role = _root;
  visited[thisrid] = TRUE; // Allow access twice
  queue->append(cur_role);
  cout << "Roles Graph:" << endl;
  while (!queue->is_empty()) {
    cur_role = queue->remove_first();
    cur_role->_juniorList->reset_current(); // double insured
    cout << *(cur_role->getRIDp()->get_RnameString()) << "-->"
       << endl;
    while (junior_role = cur_role->_juniorList->iterator()) {
      thisrid = junior_role->getRIDp()->get_RIDlong();
      cout << "                  "
         << *(junior_role->getRIDp()->get_RnameString()) << endl;
      if (visited[thisrid] == FALSE) {
      queue->append(junior_role);
      visited[thisrid] = TRUE;
      }
    }
    cur_role->_juniorList->reset_current();
  }
  delete [] visited;
  return FALSE;
}

CORBA::Boolean RolesGraph::addUser(CORBA::String& login,
```

```
                                    CORBA::String& ip,
                                    RID *rid)
{
    int i;
    CosRoles::user_info userinfo;

    for (i = 0; i < _userinfo.length(); i++) {
        if ((_userinfo[i].login() == login) &&
            (_userinfo[i].ip() == ip)) {
            cout << "Username " << login << " exists." << endl;
            return FALSE;
        }
    }
    userinfo.login() = login;
    userinfo.ip() = ip;
    userinfo.rid_long() = rid->get_RIDlong();
    _userinfo.append(userinfo);
    return TRUE;
}

CORBA::Boolean RolesGraph::delUser(CORBA::String& login,
                                    CORBA::String& ip)
{
    int i;

    for (i = 0; i < _userinfo.length(); i++)
        if (_userinfo[i].login() == login && _userinfo[i].ip() == ip) {
            _userinfo.remove(i);
            return TRUE;
        }
    return FALSE;
}

CORBA::Boolean RolesGraph::checkSecurity(CORBA::String& login,
                                         CORBA::String& ip,
                                         CORBA::String& privID)
{
    int i;
    long rid=0;
    Role *user_role;

    cout << "In checkSecurity : " << endl;
    for (i = 0; i < _userinfo.length(); i++) {
        if ((_userinfo[i].login() == login) &&
            (_userinfo[i].ip() == ip)) {
           cout << "Username " << login << " exists and belongs to role
" << endl;
            rid = _userinfo[i].rid_long();
            //          cout << *(_node_ptr[rid]->getRIDp()-
>get_RnameString()) << endl;
            break;
        }
    }
    cout << "here 2" << endl;
```

```
    if (!rid) {
        cout << "Can't find security information of user " << login
            << endl;
        return FALSE;
    }
    else
        cout << "rid is " << rid << endl;
    user_role = _node_ptr[rid];
    if (!user_role->canAccess(login, ip))
        return FALSE;
    if (user_role->inUnionPrivset(login, ip, privID)) {
        cout << "User " << login << " pass security clearance for "
            << privID << endl;
        return TRUE;
    }
    cout << "after check inUnionPrivset" << endl;
    return FALSE;
}
```

APPENDIX V

SECURITY MONITOR PROGRAM

```
#ifndef _monitor_h
#define _monitor_h

// Monitor server program. Caches compute servers

#include "comp_c.hh"// ComputeServer client
#include "CosRoles_s.hh"
#include "RID.h"

class SecurityEventHandler: public CORBA::BOA::IMPLEventHandler
{
  protected:
    void pre_method (const CORBA::Principal& princ,
                     const char *interface, const CORBA::Object& obj,
                     CORBA::ULong methodid, CORBA::Environment& env);
};

class MonitorServer: public CosRoles_impl::Monitor_impl
{
        friend SecurityEventHandler;
    private:
      CosRoles::ServerList_servers;
      CORBA::ULong_cur_index;
      CosRoles::AccessControlList_userList; // temporary move from pri-
vate

    protected:
      void  unbind(const CORBA::Principal& princ, const char *interface,
                        const CORBA::Object& obj);
    public:
      MonitorServer(const char *name) : Monitor_impl(name) {
_cur_index = 0;}
      ~MonitorServer() {}
      void        add_user(const char *name);
      void        register_server(ComputeServer *server);
      ComputeServer *get_server(CORBA::Environment& env);
};
#endif
```

```
#include <stdlib.h>
#include <signal.h>
#include <string.h>
#include "../persist_store/CosPersPID_s.hh"
#include "../persist_store/posPID.h"
#include "../persist_store/CosPersPDS_s.hh"
#include "../persist_store/posPDS.h"
#include "CosRID_s.hh"
#include "CosRoles_s.hh"
#include "RID.h"
#include "Roles.h"
#include "misc.h"
#include "monsrv.h"

extern RolesGraph *rolesgraph;
void create_rolesgraph(void);
RolesGraph *get_rolesgraph(void);

void SecurityEventHandler::pre_method(const CORBA::Principal& princ,
                                      const char *interface, const
CORBA::Object& obj,
                                      CORBA::ULong methodid,
CORBA::Environment& env)
{
        CORBA::String name;
        CORBA::String hostname;
        RolesGraph *persistent_rolesgraph;

        CORBA::String priv("Access to project files");

        name = princ.userName();
        hostname = princ.hostName();
      cout << "Checking validity for user " << princ.userName() << "
at host "
             << princ.hostName() << endl;

        persistent_rolesgraph = get_rolesgraph();
        if (persistent_rolesgraph->checkSecurity(name, hostname, priv))
            cout << "Security test passed!" << endl;
        else
          env.exception_value(new CORBA::StExcep::NO_PERMISSION);
}

void MonitorServer::unbind(const CORBA::Principal& princ, const char *,
                  const CORBA::Object& )
{
      CORBA::ULong num_servers = _servers.length();
      for (CORBA::ULong i = num_servers; i > 0; i--) {
            ServerAssoc& server = _servers[i-1];
            if ( server.host_name() == princ.hostName() &&
                        server.pid() == princ.pid() )
                  _servers.remove(i-1);
      }
}
```

```
void MonitorServer::add_user(const char *name)
{
      _userList.append(new CORBA::String(name));
}

void MonitorServer::register_server(ComputeServer *server)
{
      // Get client info
      const CORBA::Principal *princ = _principal();

      ServerAssoc *assoc = new ServerAssoc;
      assoc->host_name() = princ->hostName();
      assoc->pid() = princ->pid();
      assoc->server(server);

        cout << "host name is " << princ->hostName() << ", pid is " <<
princ->pid()
           << ", loginName is " << princ->loginName() << ", password is
"
           << princ->password() << endl;

      // NOTE: We do not need to _duplicate the server
      // (Increment the ref count) since append on sequences automati-
cally
      // increment the ref count. The ref count is decrmented when
      // the element is removed/ sequence is deleted.
      _servers.append(assoc);
}

// Uses Round Robin scheduling to give out ComputeServer Objects
ComputeServer *MonitorServer::get_server(CORBA::Environment& env)
{
    //if ( ! valid_user(_principal()) ) {
    //      env.exception_value(new PermissionDenied);
    //      return NULL;
    //}

      if ( _servers.length() == 0) {
            env.exception_value(new NoServers);
            return NULL;
      }
      if ( _cur_index >= _servers.length() )
            _cur_index = 0;
      ComputeServer *server = _servers[_cur_index++].server();
      server->_duplicate();// Return value is released by ORB
      return server;
}

void sighandler(int)
{
      exit(0);
}
```

```
int main(int argc, char **argv)
{
      signal(SIGINT, sighandler);
      MonitorServer server("TestMonitor");

       CORBA::BOA *boa = CORBA::BOA::instance();

       // Create Role graph

       create_rolesgraph();

      // Attach the event handler to implementation so that unbind
      // gets called when clients disconnect

      CORBA::Environment env;

       boa->event_handler(&server, env, new SecurityEventHandler);

      if ( env.check_exception() ) {
            cout << "Error registering event handler." << endl;
            cout << env;
      }

      CORBA::BOA::impl_is_ready();
      return 0;
}
```

APPENDIX VI

ROLE GRAPH INITIALIZATION PROGRAM

```
#include <stdlib.h>
#include <string.h>
#include "../persist_store/CosPersPID_s.hh"
#include "../persist_store/posPID.h"
#include "../persist_store/CosPersPDS_s.hh"
#include "../persist_store/posPDS.h"
#include "CosRID_s.hh"
#include "CosRoles_s.hh"
#include "RID.h"
#include "Roles.h"
#include "misc.h"

table_type symboltable[MAXTABLE+1];
RolesGraph *rolesgraph;

void create_rolesgraph()
{
RID *rid0 = new RID("Root", "flash.adb.com");
RID *rid1 = new RID("Project Manager", "flash.adb.com");
RID *rid2 = new RID("Office Manager", "flash.adb.com");
RID *rid3 = new RID("Reception", "flash.adb.com");
RID *rid4 = new RID("Engineer", "flash.adb.com");
RID *rid5 = new RID("Sr Engineer", "flash.adb.com");

CORBA::String *priv_a = new CORBA::String("Access to reception desk");
CORBA::String *priv_b = new CORBA::String("Personal files");
CORBA::String *priv_c = new CORBA::String("Check out development
files");
CORBA::String *priv_d = new CORBA::String("Incoming phone calls");
CORBA::String *priv_e = new CORBA::String("Create new computer
account");
CORBA::String *priv_f = new CORBA::String("Issue checks");
CORBA::String *priv_g = new CORBA::String("Access to project files");
CORBA::String *priv_h = new CORBA::String("Cancel projects");
CORBA::String *priv_i = new CORBA::String("Order office supplies");

CORBA::String *login0 = new CORBA::String("john");
CORBA::String *login1 = new CORBA::String("eugene");
```

```
CORBA::String *login2 = new CORBA::String("sharon");
CORBA::String *login3 = new CORBA::String("michael");
CORBA::String *login4 = new CORBA::String("michelle");
CORBA::String *login5 = new CORBA::String("william");
CORBA::String *login6 = new CORBA::String("tom");
CORBA::String *login7 = new CORBA::String("cheryl");
CORBA::String *login8 = new CORBA::String("jean");
CORBA::String *login9 = new CORBA::String("dave");
CORBA::String *ip0 = new CORBA::String("flash.adb.com");

CosRoles::seqId privset0, privset1, privset2, privset3, privset4,
    privset5;

CosRoles::seqAuth al_reception, al_off_man, al_engineer, al_sr_engineer,
    al_proj_man, al_root;
CosRoles::seqAuth dl_reception, dl_off_man, dl_engineer, dl_sr_engineer,
    dl_proj_man, dl_root;

CosRoles::auth_para auth_parameter0, auth_parameter1, auth_parameter2,
    auth_parameter3, auth_parameter4, auth_parameter5, auth_parameter6,
    auth_parameter7, auth_parameter8, auth_parameter9;

CORBA::String *priv_j = new CORBA::String("Audit accounts");
CosRoles::seqId privset_fc;
CosRoles::seqAuth al_fc, dl_fc;
CosRoles::auth_para auth_parameter10;
CORBA::String *login10 = new CORBA::String("tim");

for (int i = 0; i < MAXTABLE+1; i++)
      *(symboltable[i].name) = '\0';

  privset0.append(*priv_a);
  privset0.append(*priv_d);
  privset1.append(*priv_c);
  privset1.append(*priv_g);
  privset2.append(*priv_b);
  privset2.append(*priv_i);
  privset2.append(*priv_f);
  privset3.append(*priv_g);
  privset4.append(*priv_h);
  privset5.append(*priv_e);


  auth_parameter0.login(*login0);
  auth_parameter0.ip(*ip0);

  auth_parameter1.login(*login1);
  auth_parameter1.ip(*ip0);

  auth_parameter2.login(*login2);
  auth_parameter2.ip(*ip0);

  auth_parameter3.login(*login3);
  auth_parameter3.ip(*ip0);
```

```
auth_parameter4.login(*login4);
auth_parameter4.ip(*ip0);

auth_parameter5.login(*login5);
auth_parameter5.ip(*ip0);

auth_parameter6.login(*login6);
auth_parameter6.ip(*ip0);

auth_parameter7.login(*login7);
auth_parameter7.ip(*ip0);

auth_parameter8.login(*login8);
auth_parameter8.ip(*ip0);

auth_parameter9.login(*login9);
auth_parameter9.ip(*ip0);

al_reception.append(auth_parameter2);
al_reception.append(auth_parameter7);
dl_reception.append(auth_parameter3);
dl_reception.append(auth_parameter5);
al_engineer.append(auth_parameter3);
al_engineer.append(auth_parameter6);
al_engineer.append(auth_parameter4);
dl_engineer.append(auth_parameter2);
dl_engineer.append(auth_parameter7);
dl_engineer.append(auth_parameter5);
al_off_man.append(auth_parameter8);
dl_off_man.append(auth_parameter2);
dl_off_man.append(auth_parameter7);
dl_off_man.append(auth_parameter3);
dl_off_man.append(auth_parameter6);
al_sr_engineer.append(auth_parameter0);
dl_sr_engineer.append(auth_parameter8);
dl_sr_engineer.append(auth_parameter3);
al_proj_man.append(auth_parameter9);
dl_proj_man.append(auth_parameter0);
dl_proj_man.append(auth_parameter8);
al_root.append(auth_parameter1);
dl_root.append(auth_parameter9);

List *jl_reception = new List;
List *jl_off_man = new List;
List *jl_proj_man = new List;
List *jl_sr_engineer = new List;
List *jl_engineer = new List;
List *jl_root = new List;

Role *role_reception = new Role(rid3, privset0, al_reception,
                        dl_reception, jl_reception);
Role *role_engineer = new Role(rid4, privset1, al_reception,
                        dl_reception, jl_engineer);
```

```
  jl_off_man->append(role_reception);
  Role *role_off_man = new Role(rid2, privset2, al_off_man,
                        dl_off_man, jl_off_man);
  jl_sr_engineer->append(role_engineer);
  jl_sr_engineer->append(role_reception);
  Role *role_sr_engineer = new Role(rid5, privset3, al_sr_engineer,
                            dl_sr_engineer, jl_sr_engineer);

  Role *role_proj_man = new Role(rid1, privset4, al_proj_man,
                       dl_proj_man, jl_proj_man);
  Role *role_root = new Role(rid0, privset5, al_root,
                 dl_root, jl_root);

  rolesgraph = new RolesGraph(role_root);
  List *sl = new List;
  sl->append(role_root);
  rolesgraph->add(sl, role_proj_man);
  List *sl2 = new List;
  sl2->append(role_proj_man);
  rolesgraph->add(sl2, role_sr_engineer);
  rolesgraph->add(sl2, role_off_man);

  Role *role;
  role = rolesgraph->Root();
  cout << "Root role is " << *(role->getRIDp()->get_RnameString()) <<
endl;

  RID rid6("Financial Controller", "flash.adb.com");
  privset_fc.append(*priv_j);
  auth_parameter10.login(*login10);
  auth_parameter10.ip(*ip0);
  al_fc.append(auth_parameter10);
  dl_fc.append(auth_parameter2);
  dl_fc.append(auth_parameter9);
  List *jl_fc = new List;
  jl_fc->append(role_engineer);
  jl_fc->append(role_off_man);
  Role *role_fc = new Role(&rid6, privset_fc, al_fc, dl_fc, jl_fc);
  List *sl_fc = new List;
  sl_fc->append(role_root);

  rolesgraph->addUser(*login0, *ip0, rid5);
  rolesgraph->addUser(*login1, *ip0, rid0);
  rolesgraph->addUser(*login2, *ip0, rid3);
  rolesgraph->addUser(*login3, *ip0, rid4);
  rolesgraph->addUser(*login4, *ip0, rid4);
  rolesgraph->addUser(*login6, *ip0, rid4);
  rolesgraph->addUser(*login7, *ip0, rid3);
  rolesgraph->addUser(*login8, *ip0, rid2);
  rolesgraph->addUser(*login9, *ip0, rid1);

    // Creating persistent rolesgraph

  RolesGraph *newobj;
```

```
  newobj = rolesgraph;
  PosPID newpid("EOS.2.0.2", "Rolegraph", "flash.adb.com");
  CORBA::String *newstring;

  newstring = newpid.get_PIDString();
  cout << *newstring << endl;
  PosPDS *newpds = new PosPDS((CosPersPDS::Object *) newobj, &newpid);

  // Remove persistent role graph

  newpds->connect((CosPersPDS::Object *) newobj, &newpid);
  newpds->remove((CosPersPDS::Object *) newobj, &newpid);
  newpds->disconnect((CosPersPDS::Object *) newobj, &newpid);

  newpds->setobjsize((long) sizeof(RolesGraph));

  // Store persistent role graph

  newpds->connect((CosPersPDS::Object *) newobj, &newpid);
  newpds->store((CosPersPDS::Object *) newobj, &newpid);
  newpds->disconnect((CosPersPDS::Object *) newobj, &newpid);


}

RolesGraph *get_rolesgraph(void)
{
  /*
  cout << "Start testing isCycle......" << endl;
  if (rolesgraph->isCycle(sl_fc, role_fc))
    cout << "Cycle exists after adding role, "
       << *(role_fc->getRIDp()->get_RnameString()) << endl;
  else
    cout << "Cycle doesn't exist after adding role, "
       << *(role_fc->getRIDp()->get_RnameString()) << endl;

  if (rolesgraph->isJunior(rid4, rid1))
    cout << *(rid4.get_RnameString()) << " is a junior of " <<
      *(rid1.get_RnameString()) << endl;
  else
    cout << *(rid4.get_RnameString()) << " is not a junior of " <<
      *(rid1.get_RnameString()) << endl;
      */
    //  rolesgraph->displayGraph();

  PosPID newpid("EOS.2.0.2", "Rolegraph", "flash.adb.com");
  CORBA::String *newstring;
  RolesGraph *getobj = new RolesGraph;

  newstring = newpid.get_PIDString();
  cout << *newstring << endl;
  PosPDS *newpds = new PosPDS((CosPersPDS::Object *) getobj, &newpid);

  newpds->connect((CosPersPDS::Object *) getobj, &newpid);
```

```
  if ((getobj = (RolesGraph *) newpds->restore((CosPersPDS::Object *)
                                                getobj,
                                                &newpid)) != NULL) {
        cout << "After restore" << endl;
        cout << "Object ptr after restoring is " << getobj << endl;
        getobj->displayGraph();
    } else
        cout << "return null from restore" << endl;
    cout << "Before disconnect" << endl;
    newpds->disconnect((CosPersPDS::Object *) getobj, &newpid);
    cout << "After disconnect" << endl;

    return getobj;
}
```

Bibliography

[1] Baldwin, R. (1990). "Naming & Grouping Privileges to Simplify Security Management in Large Databases". Proceeding of 1990 IEEE Symposium on Research in Security and Privacy, pages 116-132. IEEE Computer Society Press, May 1990.

[2] Bancilhon, F. Delobel, C. and Kanellakis, P. editors, (1991). "Building an Object-Oriented Database System - The Story of O2". Morgan Kaufmann Publishers.

[3] Barton, J. and Nackman, L. (1995). Scientific and Engineering in C++. Addisen-Wesley, Menlo Park, California.

[4] Bell, D. and LaPadula, L., (1976). "Secure Computer Systems: Unified Exposition and Multics Interpretation". Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, Mass., March 1976.

[5] Bertino, E., Martino, L., (1993). Object-Oriented Database Systems, Concepts and Architectures, Addison-Wesley Publishing Company.

[6] Biliris, A., Panagos, E. EOS User's Guide Release 2.2. AT&T Bell Laboratories, Murray Hill, NJ.

[7] Booch, G.Object-Oriented Analysis and Design, Second edition, The Benjamin/ Cummings Publishing Company, Inc. 1994.

[8] Cahill, V., Baker, S., Tangney, B., Horn, C., Harris, N., (1992). "On Object Orientation as a Paradigm for General Purpose Distributed Operating Systems". Proceedings of the ACM SIGOPS European Workshop, December, 1992.

[9] Carey, M., DeWitt, D., Graefe, G., Haight, D., Richardson, J., Schuh, D., Shekita, E., and Vandenberg, S., (1990). "The EXODUS Extensible DBMS Project: An Overview". In S. Zdonik and D. Maier eds., Readings in Object-Oriented Databases, Morgan-Kaufman, 1990.

[10] Carey, M., DeWitt, D., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh, D., Solomon, M., Tan, C., Tsatalos, O., White, S., and Zwilling, M., (1994). "Shoring Up Persistent Applications". Proceeding of the 1994 ACM SIGMOD Conference, Minneapolis, MN, May 1994.

[11] Casais, E., Ranft, M., Schiefer, B., Theobald, D. and Zimmer, W. (1992). "OBST - An Overview", Technical Report FZI.039.1, Forschungszentrum Informatik (FZI).

[12] Cattell, R. Object Database Standard: ODMG-93. Contributions by T. Atwood, J. Dubl, G. Ferran, M. Loomis, and D. Wade. Morgan Kaufmann, San Mateo, California, 1993.

[13] Chakravarthy, S., Anwar, E., Maugis, L., (1993). "Design and Implementation of Active Capability for an Object-Oriented Database". Tech. Report UF-CIS-TR-93-001, University of Florida, January 1993.

[14] Department of Defense. (1985). Department of Defense Trusted Computer System Evaluation Criteria, DOD 5200.28-STD.

[15] Deux, et al., (1990). "The Story of O2", IEEE Transactions on Knowledge and Data Engineering.

[16] Dogac, A., Altinel, M. Ozkan, C., Durusoy, I., (1995). "Implementation Aspects of an Object-Oriented DBMS". SIGMOD Record, Vol. 24, No. 1, March 1995.

[17] Elmasri, R., Navathe, S., (1994). Fundamentals of Database Systems, The Benjamin/ Cummings Publishing Company, Inc.

[18] Fishman, et al. (1987). "Overview of the Iris DBMS". ACM Transactions on Office Information Systems, vol. 5, No. 1, January 1987.

[19] Garcia-Molina, H., Hsu, M., (1995). "Distributed Databases". In W. Kim editor, Modern Database Systems, ACM Press, 1995.

[20] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sunderam, V. PVM, Parallel Virtual Machine, A Users' Guide and Tutorial for Network Parallel Computing. MIT Press, 1994.

[21] Janssen, B. Severson, D., Spreitzer, M., (1995). ILU 1.8 Reference Manual. Xerox Corporation, March 1995.

[22] Karlapalem, K., Navathe, S., Morsi, M., (1995). "Issues in Distribution Design of Object-Oriented Databases". In M. Ozsu, U. Dayal, and P. Valduriez, editors, Distributed
Object Management, Morgan Kaufmann, 1995.

[23] Kim, W., (1993). "Object-Oriented Database Systems: Promises, Reality, and Future". In W. Kim editor, Modern Database Systems, ACM Press, 1995.

[24] Kim W., Ballou, N., Chou, H., Garza, J. and Woelk, D. (1989). "Features of the ORION Object-Oriented Database System". Object-Oriented Concepts, Databases and Applications, Kim and Lochovsky (editors), ACM Press, 1989.

[25] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. "The ObjectStore database system.". Communications of the ACM, 34(10):50-63, October 1991.

[26] Lawrence, L., (1993). "The Roles of Roles". Computers & Security, 12(1993) 15-21, Elsevier Science Publishers Ltd.

[27] Lecluse, C., Richard, P. and Velez, F., (1990). "An Object-Oriented Data Model", Readings in Object Oriented Database Systems, Zdonik and Maier (editors) Morgan Kaufmann.

[28] Lecluse, C. and Richard, P. (1989). "The O2 Database Programming Language", Proceedings of the Fifteenth International Conference on Very Large Databases, August 1989, p. 411.

[29] Lunt, T. (1989). "Access Control Policies: Some Unanswered Questions". Computers and Security, February, 1989.

[30] Lunt, T.,(1992). "Sea View". In T. Lunt editor, Research Directions in Database Security, Springer-Verlag, 1992.

[31] Lunt, T., (1995). "Authorization in Object-Oriented Databases". In W. Kim editor, Modern Database Systems, ACM Press, 1995.

[32] Nyanchama, M. Osborn, S., (1993), "Role-Based Security: Pros, Cons, & Some Research Directions". ACM SIGSAC Review, 2(2):11-17, June 1993.

[33] Nyanchama, M. Osborn, S., (1993), "Role-Based Security, Object Oriented Databases & Separation of Duty". Technical Report no. 442, University of Western Ontario, October, 1993.

[34] Nyanchama, M., Osborn, S., (1994), "Access Rights Administration in Role-Based Security Systems". In Database Security VIII: Status & Prospects, August 1994.

[35] Nyanchama, M., Osborn, S., (1995). "Database Security Issues in Distributed Object-Oriented Databases". In M. Ozsu, U. Dayal, and P. Valduriez, editors, Distributed Object Management, Morgan Kaufmann, 1995.

[36] ObjectBroker White Paper. Digital Equipment Corporation, 1995.

[37] The Orbix Architecture. Iona Technologies, Ltd., January 1995.

[38] Orbeline User's Guide. PostModern Computing, 1994.

[39] Olivier, M., Solms, S., (1994). "A Taxonomy for Secure Object-Oriented Databases". ACM Transactions on Database Systems, Vol. 19, No. 1, March 1994, p. 3-46.

[40] OMG-CORBA (1995). The Common Object Request Broker: Architecture and Specification. Revision 2.0, Object Management Group, July 1995.

[41] OMG-COSS (1995). CORBAservices: Common Object Services Specification, Object Management Group, March 1995.

[42] OMG-TC-95.3.3 (1995). OSTF RFP3 Submission CORBA Security. OMG Document Number 95-3-3, March 1995.

[43] OMG-TC-93.11.3 (1993). IBM/JOSS Object Services Persistence Service Specification, Submission to OMG, OMG TC Document Number 93.11.3. Revised November 15, 1993.

[44] OMG-TC-93.5.7 (1993). IBM Object Persistence Service, Submission to OMG, OMG TC Document 93.5.7.

[45] Pissinou, N., Makki, K. and Park, E. K. (1994). "Towards a Framework for Integrating Multilevel Secure Models and Temporal Data Models". Proceedings of the ACM International Conference on Information and Knowledge Management, pp. 280-287, Gaithersburg, Maryland, November 1994.

[46] Pissinou, N., Makki, K., Vanapipat, K. and Rajashekhar, B. (1996). "On Building Distributed Applications". Proceedings of the International Conference on Parallel Processing, vol. II, Bloomingdale, IL, August, 1996.

[47] Rabitti, F., Woelk, D., and Kim, W. (1988). "A Model of Authorization for Object-Oriented and Semantic Databases". Proceedings of the International Conference on Extending Database Technology, 1988.

[48] Rabitti, F., Bertino, E., Kim, W., and Woelk, D. (1991). "A Model of Authorization for Next Generation Database Systems". ACM Trans. on Database Systems, vol. 16, no. 1, p88-131, March 1991.

[49] Sandhu, R., Thomas, R., Jajodia, S., (1992). "Supporting timing-channel free computations in multilevel secure object-oriented databases". In C. Landwehr and S. Jajodia editors, Database Security, V: Status and Prospects, 1992, North-Holland.

[50] Schmidt, D., Vinoski, S., (1995). "Object Interconnections-Modeling Distributed Object Applications". SIGS C++ Report magazine, February 1995.

[51] Schmidt, D., Vinoski, S., (1995). "Object Interconnections-Comparing Alternative Client-side Distributed Programming Techniques". SIGS C++ Report magazine, May 1995.

[52] Singhal, V., Kakkad, S., Wilson, P. (1992). "Texas: An Efficient, Portable Persistent Store". Fifth International Workshop on Persistent Object Systems, San Miniato, Italy, September 1992.

[53] Soley, R., Kent, W. (1995). "The OMG Object Model". In W. Kim editor, Modern Database Systems, ACM Press, 1995.

[54] SOMobjects Developer Toolkit, Technical Overview, Version 2.0, IBM Corp. November 1993.

[55] SunSoft's DOE Product family-Product Overview, Sun Microsystems, Inc. 1995.

[56] Thomsen, D. (1991). "Role-Based Application Design and Enforcement". In S. Jajodia and C. E. Landwehr, editors, Database Security, IV: Status and Prospects, p. 151-168. North-Holland, 1991.

[57] Ting, T., Dermurjan, S., Hu., M., (1992). "Requirements Capabilities and Functionalities of User-Role Based Security for an Object-Oriented Design Model". In C. E. Landwehr and S. Jajodia, editors, Database Security V: Status & Prospects, p. 275-296. North-Holland, 1992.

[58] Vinoski, S. (1993). "Distributed Object Computing with CORBA". C++ Report magazine, July/August 1993.

[59] Wells, D., Blakeley, J., (1995). "Distribution and Persistence in the Open Object-Oriented Database System". In M. Ozsu, U. Dayal, and P. Valduriez, editors, Distributed Object Management, Morgan Kaufmann, 1995.