INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Université d'Ottawa • University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

COCARD, Remus

AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M.A.Sc. (Electrical Engineering)

GRADE - DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

A Generic Database Adapter for Multi-Tier CORBA based Applications

Dan Ionescu

DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

B. Pagurek

N. Georganas

J.-M. De Koninck, Ph.D.

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

SIGNATURE

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

A Generic Database Adapter

For Multi-tier

CORBA based Applications

By

Remus Cocard

A thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements for the degree of

Masters of Applied Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa
January 2002

Abstract

This thesis presents a generic solution to the problem of database adapters for multi-tier CORBA based applications. This solution is based on the concept of *metatype classes*, which provide information about the types of objects that are being stored in a database. The concept of a *metaobject protocol* is introduced and discussed. The Database Adapter is proposed as a generic solution for managing persistent information for multi-tier CORBA based applications. A framework for developing multi-tier applications using a generic adapter is presented. A test application is developed to evaluate the management of objects and to measure the impact on performance. It is found that the ability to manage objects through the use of a generic adapter, without modifying the adapter when managing different new types of objects, results in only a slight increase in operation invocation time, but results in huge savings in development and testing time.

Acknowledgments

I would like to thank my supervisor, Dr. Dan Ionescu, for his guidance, encouragement, optimism and support throughout my research. His ability to set goals and challenge one to achieve them is responsible for the complexity of this thesis.

I also wish to thank to CGI Inc. Ottawa for supporting my Master's program by giving me the time, financial support, and encouragement to complete it.

I have a special thank-you to my beloved wife Narcisa for her understanding and support throughout my research.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

BOA	Basic Object Adapter
CORBA	OMG standard for distributed computing applications.
DBA	Database Administrator
DDO	Dynamic Data Object
DDL	Data Definition Language
DII	Dynamic Invocation Interface. Runtime operation requests used by clients.
DSI	Dynamic Skeleton Interface for server implementation
GDA	Generic Database Adapter
GDAI	Generic Database Adapter Interface
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDL	Interface Definition Language defined by the OMG for CORBA.
IIOP	Internet Inter-ORB Protocol
IOR	Interoperable Object Reference for objects in separate CORBA processes
IPC	Inter-process Communication
IR	Implementation Repository that registers CORBA server executables
ISO	International Standards Organization
MOP	Meta-Object Protocol
OID	Object Identifier
OMA	Object Management Architecture
OMG	Object Management Group.
OODBMS	Object-Oriented Database Management System
OQL	Object Query Language
ORB	Object Request Broker, the CORBA process that provides communications between client and server processes
OTS	Object Transaction Service
PDS	Persistent Data Service
PID	Persistent Identifier

PO	Persistent Object
POA	Portable Object Adapter
POM	Persistent Object Manager
POS	Persistent Object Service as specified by the OMG
RDBMS	Relational Database Management System
SII	Static Invocation Interface. Compiled proxies used by client processes.
SQL	Sequential Query Language
SSI	Static Skeleton Interface for server implementation.
TCP/IP	Transmission Control Protocol/Internet Protocol
TMN	Telecommunication Management Network
UML	Unified Modeling Language

1 Introduction

1.1 Motivation and Research Objectives

As we enter a new century, Information Technology is present throughout our society. New technologies are being developed at a rapid rate, resulting in fast growth of system complexity, and an urging need for applications evolution and for integration of legacy systems. A result of this phenomenon is the trend towards distributed computing environments in almost all the segments of our industry. A standard for distributed computing called Common Object Request Broker Architecture (CORBA)[19], has been created in an effort to manage the complexity of developing these systems, by a consortium of major software vendors and developers called the Object Management Group (OMG). As part of the CORBA standard, a software component called Object Request Broker (ORB)[7] is being used in order to facilitate the transmission of messages between different servers and clients potentially located on different platforms, using different operating systems and programming languages. The entire network complexity is hidden within the ORB itself, shielding the developer from the burden of dealing with this directly. This complex feature is facilitated by the definition of a common interface between distributed objects or components that form an application, using a common language: the OMG's Interface Definition Language (IDL).

An aspect of these large distributed systems, not directly addressed by the CORBA specification, is how all this information being exchanged between different systems is to be stored persistently on and retrieved from persistent storage for future use. As the amount of information being processed and exchanged between different systems increases dramatically the need for managing this information by storing/retrieving it to/from persistent storage increases also. Lots of CORBA applications are developed all over the world for different purposes and the mechanism for managing persistent information is re-invented each time. Even more, during the development cycle for the same application, each time new classes of objects are added to an existing application,

the same development procedures have to be repeated again and again. This can result in a significant time increase for the development and testing processes, which in turn would increase the costs for developing a specific application.

Different implementations of the CORBA standard provide a flexible distribution model but they have only basic support for persistence. On the other hand different RDBMS (Relational Database Management Systems)[38,41,42] and OODBMS (Object-Oriented Database Management Systems)[38] provide powerful support for persistence but support only a limited form of distribution. The creation of a Generic Database Adapter (GDA) which would manage information stored in relational databases and/or object oriented databases is required. The Generic Database Adapter (GDA) is proposed in this thesis to perform the following functions:

- Create multiple, distributed databases located on the same machine or different machines as part of a computer network;
- Store information persistently in one or multiple databases located on the same machine or different machines;
- Retrieve information stored persistently in one or multiple databases located on the same machine or different machines;
- Change information stored persistently in one or multiple databases located on the same machine or different machines;
- All operations on the information stored persistently are transaction controlled, eliminating the possibility to have corrupt information;
- Group together related information;
- Query the information stored persistently;
- Browse through and/or change the result of the query;
- All application exceptions are handled and propagated back to the client; and
- Support for concurrent multi-client access.

Note that the Generic Database Adapter exports an interface (GDAI) that can be used by any client that needs to create, retrieve, change, and query information stored persistently in one or multiple databases located on one or distributed systems.

An example of a distributed system requiring such a Generic Database Adapter is the "BaseLayer" CORBA-based network management system developed at the Machine Intelligence Research Laboratory, University of Ottawa [12, 13, 14]. One of the goals of the BaseLayer application is to develop a methodology for persistently storing objects created (newly defined and previously defined) in a distributed network management environment, and retrieving and/or changing these objects without having to alter the Generic Database Adapter in any way. As objects are instantiated on CORBA servers and managed by remote clients through CORBA object references, the client can decide if the state of those objects has to become persistent or not, and if persistently stored objects have to be retrieved, changed, or removed. All the changes affecting an object or a group of objects are either entirely committed to the persistent storage or all the changes are rolled-back and all the objects are reverted back to the states they had before the change operations started. The client can control the number of operations that have to be grouped together in order to avoid having corrupt information. The client can retrieve objects or groups of related objects and execute different operations on them. The system must ensure that persistently stored objects retain their state for as long as the client decides.

This thesis presents a Generic Database Adapter to support the managing of objects in a distributed environment. A framework is presented for storing, retrieving, and changing of CORBA objects and groups of CORBA objects. An implementation of this framework is constructed to demonstrate the use of generic database adapters. Results are presented on the impact to performance and data integrity caused by the Generic Database Adapter. Finally, recommendations are made for an alternate implementation solution for the Generic Database Adapter which will use the Front-End approach[7], and for an integrated solution which will integrate also Relational Database Management Systems (RDBMS).

1.2 Organization of the Thesis and Contributions

This thesis is organized to provide an incremental understanding of the object oriented databases, relational databases, and the problem of generic database adapters used in a multi-tier distributed environment, including the concept of metaobject data. A complete implementation mechanism is proposed for the Generic Database Adapter, and implementation results are presented.

Chapter 2 describes the need for and the problems of database adapters in multi-tier distributed environments. Recent work in the field of database adapters for multi-tier distributed applications is reviewed. The requirements for a generic database adapter for multi-tier distributed database related applications are defined.

Chapter 3 presents an overview of the Common Object Request Broker Architecture. The aspects of CORBA that are applied in subsequent chapters are presented in detail while other concepts are mentioned for completeness.

Chapter 4 presents an overview of the Database Management System paradigm and the most important Database Management Systems architectures that are used nowadays throughout the economy. Advantages and disadvantages are presented for each Database Management System described.

Chapter 5 presents the Object Store Database Management System chosen for the implementation of the Generic Database Adapter. Technical aspects of the Object Store DBMS that are relevant to implementing a generic database adapter are examined and presented.

A complete implementation and description of a Generic Database Adapter is presented in Chapter 6. Rational Rose, UML and the Objectory methodology are used to describe the implementation of the Generic Database Adapter presented in this thesis. The results of the functional and performance tests are presented and discussed.

Finally, in Chapter 7 conclusions are presented on the costs and benefits of the Generic Database Adapter. The impact of using a Generic Database Adapter on large scale distributed systems is discussed. Recommendations are made for another implementation of the Generic Database Adapter and for the extension of the adapter for using Relational Database Management Systems.

This thesis contains several research contributions, which can be summarized as follows:

1. An innovative concept in database related multi-tier CORBA based applications, is presented for the first time (the Generic Database Adapter).

2. A fully functional implementation for a Generic Database Adapter is presented and described. Full functional and performance tests are performed and described. The superior advantage in using Generic Database Adapters is presented and demonstrated.

3. Recommendations are made for future research in Generic Database Adapters.

2 Generic Database Adapters for Multi-tier Distributed Applications

This chapter describes the need for and the problems of database adapters in multi-tier distributed environments. Related work in the field of database adapters in multi-tier distributed systems is discussed. The requirements for a generic database adapter in database related applications are defined.

2.1 The Problem of Database Adapters in Multi-tier Distributed Environments

A multi-tier distributed object-oriented application typically consists of multiple client processes referencing objects on remote server processes, which are stored/retrieved to/from persistent storage by the means of a database adapter and DBMS[7]. Throughout the thesis a three-tier architecture will be used as a representative of a multi-tier architecture.

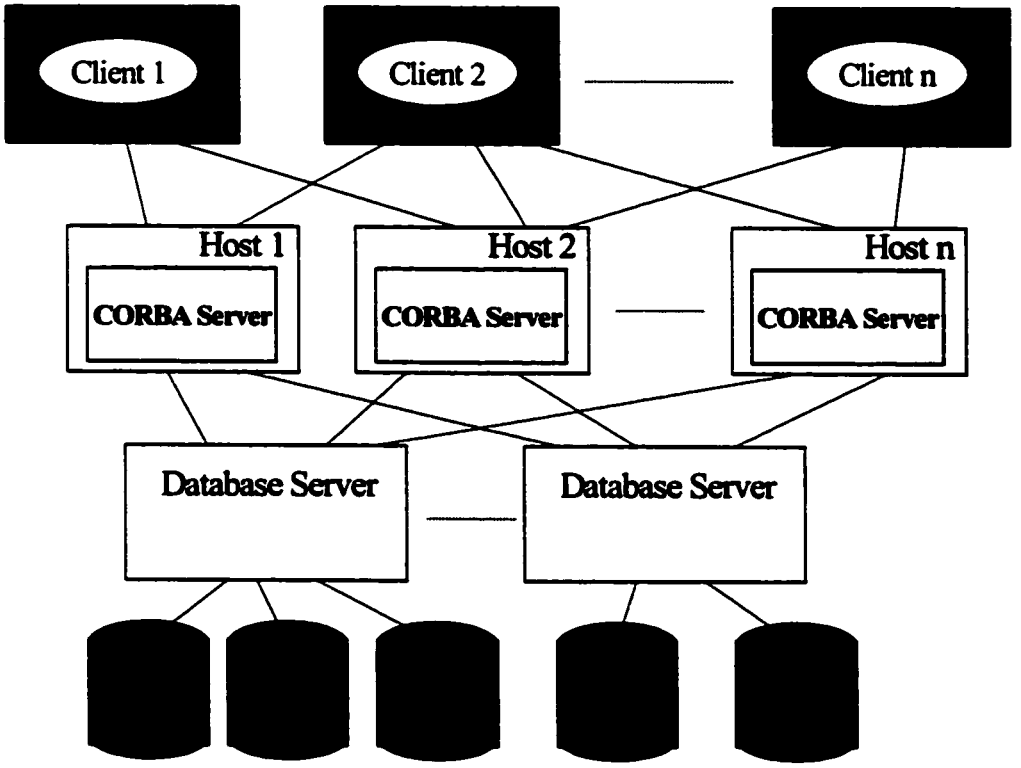

Figure 2 - 1 Three-tiered architecture.

A three-tier architecture allows a separation of the information presentation mechanism from the means by which the information is stored. This allows presentation and storage to be developed independently from each other which means applications can more easily take advantage of evolving GUI technology or faster storage systems.

Like stand-alone software applications, multi-tier distributed applications must have the possibility to store/retrieve processed information to/from persistent storage. Unlike stand-alone applications, multi-tier distributed applications must be able to access multiple distributed databases. The access to persistent storage is achieved by the means of a database adapter which should manage persistently stored information and which should be as generic as possible. A generic database adapter represents a database adapter which does not have to be altered in order to manage existing stored information or information that is to be stored persistently regardless of the changes in the structure of the information that is stored or is to be stored persistently. The generic database adapter must be independent of any of the following elements: (1) the application architecture, (2) the software implementation, (3) the type of the persistent storage (RDBMS or OODBMS) or (4) the structure and type of the information being stored/retrieved to/from the persistent storage.

There are two different approaches for implementing a database adapter:

- The persistent CORBA objects approach
- The front-end approach

For both approaches the client does not have to be aware of the type of the persistent storage chosen or the details of how it is used.

2.2 The Persistent CORBA objects Approach

The information can be managed by a relational DBMS, by an object oriented DBMS, or by a file system. If an object oriented DBMS is used, the adapter gains generality, if a single object can be used as both a CORBA object and a persistently stored object at the same time. Thus when an invocation is made on one of these objects, the same object will be referred as a CORBA object, and when the object will be stored back in the database, the object will be referred as a valid persistent database object. If a relational DBMS is

used, then an automated relational to object-oriented mapping must exist between the existing relational database table and the CORBA objects (a relational table for each object, with a column for each member variable). This solution is known as the *Persistent CORBA objects approach[7].*

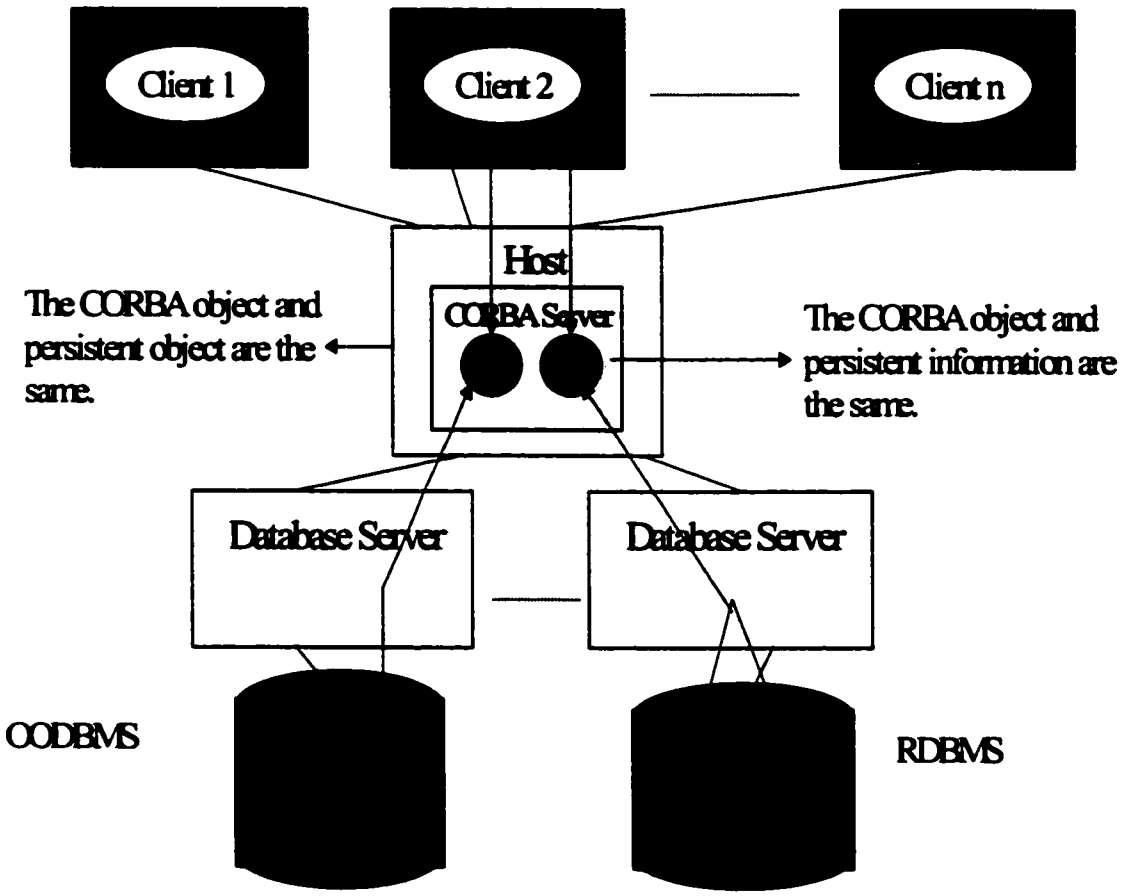

Figure 2 - 2 Persistent CORBA object approach for OODBMS and RDBMS.

2.3 The Front-end approach

If the application does not require the CORBA objects to be persistent, wrapper CORBA objects can be created for the persistently stored objects if an object oriented DBMS is used. If a relational DBMS is used, there must be a relational to object oriented mapping established (in some cases the opposite mapping also) between the existing relational database tables and the CORBA objects. In both cases the CORBA objects created will act as front-end objects that manipulate persistent information on behalf of the clients. This solution is known as the *Front-end* approach[7].

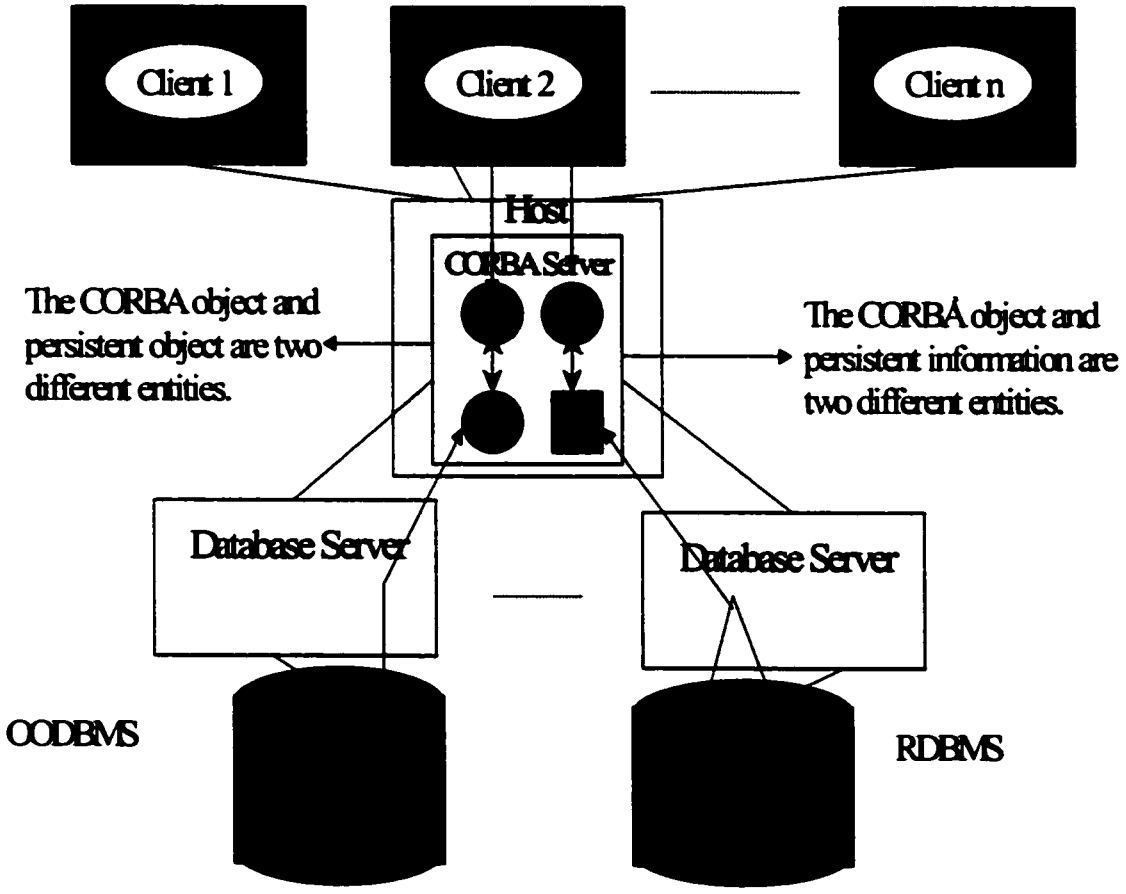

Figure 2 - 3 Front-ending approach for OODBMS and RDBMS.

The problems with regular adapters, regardless of the approach being chosen, or the type of databases being used (RDBMS, OODBMS) are:

- A new adapter has to be created each time a new application is developed, and/or
- The adapter has to be modified, or adapter specific information has to be used, each time the structure of the existing information is changed, or new information has to be stored/retrieved to/from persistent storage, and
- An IDL interface has to be defined and implemented for different database specific operations (create database, open database, remove database, create object, retrieve object, remove object, query database, etc.).

As the number of projects developed as multi-tier distributed applications increases dramatically, and the above mentioned operations have to be repeated each time in order to manage persistent information, these solutions become increasingly costly in terms of time required for application development and testing, and application complexity and size. A means for developing and enhancing such applications, without executing any modifications of the persistent storage managing mechanism, is essential. Thus, the development of a generic database adapter, that does not have to be reinvented for each new project and does not have to be changed each time modifications to the structure of the persistent information are needed, is greatly welcome.

Many industries, such as e-commerce or financing, could benefit from the ability to use a generic database adapter without having to reinvent it endlessly. For example, companies using different types of databases as part of the same application or different applications, would enormously benefit from such a solution. Such a technology would provide a significant advantage to firms developing database related applications.

2.4 Related Work

Today's real-world applications are used to be both decentralized and highly data intensive. This fact can be seen in most globally distributed applications. Any technology that tackles these problems has to provide an integrated strategy for solving both the distribution problem and the data management problem. Most applications, especially fast-processing systems, are composed of classes that have highly interconnected relationships with each other. By directly supporting extended relationships, Object databases offer a higher level of functionality than Relational databases [10]. In distributed environments object databases require a high degree of client-server coupling and mainly address the data transport problems, while under-emphasizing issues of distributed computing. This problem could be solved by the integration of the used database system with a standard distributed platform like Common Object Request Broker Architecture (CORBA). In contrast to database systems, CORBA provides a flexible transparent distribution model with a larger-scale set of services across heterogeneous vendors and products. For example, it provides the behavior invocation or the method dispatch and allows clients to use multiple data storage products. The integration of both technologies allows the use of powerful database facilities in heterogeneous CORBA environments. This chapter describes some of the existing solutions and the deficiencies presented by them.

2.4.1 Standard approaches to CORBA/ODBMS integration

The CORBA standard does not support object persistence directly, but provides the possibility of integration with different database systems. The last CORBA 2.0/IIOP specifications define two standard ways: the Persistent Object Service [6,7,18] and the Object Database Adapter [6,7].

2.4.1.1 The Persistent Object Service

The aim of the POS is to use standard IDL interfaces that allow it to be used in conjunction with other CORBA services. Implementations of the POS that successfully provide this level of reuse have been unsuccessful due to the lack of OMG standardization of the underlying components (lack of specified semantics of the IDL operations, incomplete specifications for the re-use of other Object Services by the POS). Other approaches to providing persistence have been more successful but resulted in non-standardized implementations. Because of these problems, the Persistent Object Service has not been completely implemented and therefore cannot be used in developing multi-tier CORBA based applications.

2.4.1.2 Object Database Adapter approach

The Object Database Adapter (ODA) approach – an object adapter for the objects stored in the database – provides a tight integration between the Object Request Broker (ORB) and a database system. The idea of a special object adapter for persistent objects (ODA) was pursued in the ODMG specifications [34].
The CORBA specifications define the standard Basic Object Adapter (BOA) that does not provide object persistence, but allows to be replaced by another adapter. The BOA mediates between the ORB, the generated skeletons and the object implementation [6]. With the assistance of skeletons (or of the Dynamic Skeleton Interface), it adapts the object implementation interface to the ORB-private interface.

The ODA is an object adapter for objects stored in a database. It does not completely replace the BOA, but represents its extension that depends from a particular database system [7]. It is responsible for the managing of the persistent state of implementation objects and aimed to simplify their programming. The implementation objects are stored in the database and are dynamically loaded into the server's memory upon request arrival. Unlike the BOA, it cannot assume that every implementation is in the memory and should be prepared to deal with other typical database features, e.g. locks and transactions, since persistent objects are generally shared. There were different implementations of the Object Database Adapter but none of them embraced the idea of

having a Generic Database Adapter. The disadvantage presented by these implementations is the need to repeat the same set of operations each time new objects belonging to new classes have to be managed by the ODA, and the need to recompile the code corresponding to the Object Database Adapter.

2.4.2 Non-standard approaches

Apart from the standard approaches there are many non-standard approaches. Almost all of them do not assume the usage of some special software and require a programmer's intervention [43]. Usually it is a construction of the middle-tier layer between CORBA clients and a database that will wrap an original database interface and provide an equivalent CORBA compatible interface. This method is not new and traditionally a covering layer will be named as wrapper and approaches based on this principle as wrapper approaches. For persistent objects according to the wrapper's architecture the whole group could be divided on the two following subgroups: query and mediators approaches.

2.4.2.1 Query approach

Many current relational and object-oriented database systems have a client/server architecture where the most of all computations proceed on the server side. In this case CORBA clients communicate with the database system via a wrapper which handles a particular set of queries. This approach assumes that the overhead of translating query results between the ODBMS and the ORB data models is not prohibitive [43,44]. The results of these queries are then sent to the client via an IDL interface with a statically defined set of types and operations. Such approach for the wrapper construction is named the query approach.

The approach works well until the size and complexity of query results increase. The wrapper can deal with large numbers of objects in processing a query, but the results must be expressed in relatively simple defined data types. Consequently this limitation leads to a relatively small IDL interface that contains equivalent IDL interfaces only for

basic database types and clients cannot communicate directly with database entities as normal CORBA objects.

From the other side, since the interface does not have data model specific entries, the wrapper will be independent from the schema of the particular database and can be successfully reused with other databases. As long as the interface contains only basic types that are usually similar among different database systems, it can be easy re-implemented for the other database system. This universality of the interface makes CORBA clients that use them independent from the used database system. The disadvantage in using this approach is the fact that the result have to be expressed based on simple defined data types, and the fact that clients cannot communicate directly with database entities as normal CORBA objects.

2.4.2.2 Mediators approach

A primary strength of Object databases lies in their ability to model complex objects and inter-relationships among them [44]. In the common case of an ODBMS that adds database features to the C++ like ObjectSore, database entities are instances of usual C++ classes. In addition to the state they have also methods. To make them available to the clients as usual CORBA objects, a wrapper should wrap an interface of every database class by an equivalent CORBA compatible class. This class implements all their methods by their equivalents from the database class and is responsible for the correct data mapping between CORBA C++ and ObjectStore C++ data models. The instances of these classes act as usual CORBA objects as well as database clients. In other words they mediate between CORBA skeletons and database objects converting data parameters and delegating all function calls in both directions. Because of this basic function, these CORBA objects are called mediators and correspondingly the approach as the mediators approach.

In this approach the wrapper's IDL interface becomes larger than the IDL interfaces used for the query approach, and requires the implementation of equivalent IDL interfaces for the complete database schema. The clients receive more natural database representation,

but become dependent from a particular data model. The disadvantage presented by this method is the fact that a programmer's intervention is required each time new objects belonging to new classes are to be stored persistently in the database (the programmer has to write new wrapper classes for these new classes, and recompile the whole application).

2.4.3 Deficiencies of Previous Approaches

All the solutions previously described require the programmer's intervention each time a new object, whose state has to be stored persistently, is added to a new or existing application. This implies that new code, for performing the same set of operations (store, retrieve, query, remove, etc.) has to be added, and the application has to be re-compiled and released. This implies that the time required, and implicitly the cost, for developing and maintaining multi-tier CORBA based applications, that use the solutions afore described, will go up each time the existing functionality has to be enhanced or modified.

3 CORBA for Multi-tier Distributed Applications

This chapter presents an overview of the *Common Object Request Broker* Architecture. Having CORBA as one of the two main standards used to achieve the goal of this thesis, a detailed review is considered necessary. Many of the concepts that support the creation of a generic database adapter for multi-tier distributed applications are based on the theories, principles, and functions defined by version 2.0 of the CORBA standard. Examples from existing literature on various applications of CORBA that relate to the topic of this thesis are discussed. The elements of CORBA that are used in subsequent chapters are presented in details while others are presented for completeness. The technical aspects of the CORBA standard that are relevant for the implementation of a generic database adapter are examined and presented.

3.1 The Common Object Request Broker Architecture

A distributed object system consists of multiple client programs referencing objects located on different remote servers. CORBA is a non-proprietary standard for developing distributed object systems. The CORBA standard was developed by the members of the Object Management Group (OMG), the world's largest information technology consortium. The CORBA standard consists of the core (the original CORBA) and a growing number of standards, known as CORBAservices and CORBAfacilities, that extend the core [6]. The concepts proposed in this thesis are based on revision 2.0 of the CORBA standard released in July 1996 [19].

3.1.1 The Object Management Architecture

The Object Management Architecture (OMA) defines the functions and relationships between the principal components that create an implementation of the CORBA standard. The OMA, shown in Figure 3-1, is composed of the core CORBA infrastructure which

provides the basic ORB functionality, standards for **CORBAservices** that extend ORB support for applications and user applications, and a number of CORBA frameworks, called **CORBAfacilities**, which are defined for specific domains such as e-commerce or telecommunications [7,19]. An example of a CORBAfacility implementation for Telecommunication Management Networks (TMN) is the BaseLayer application [13,14].

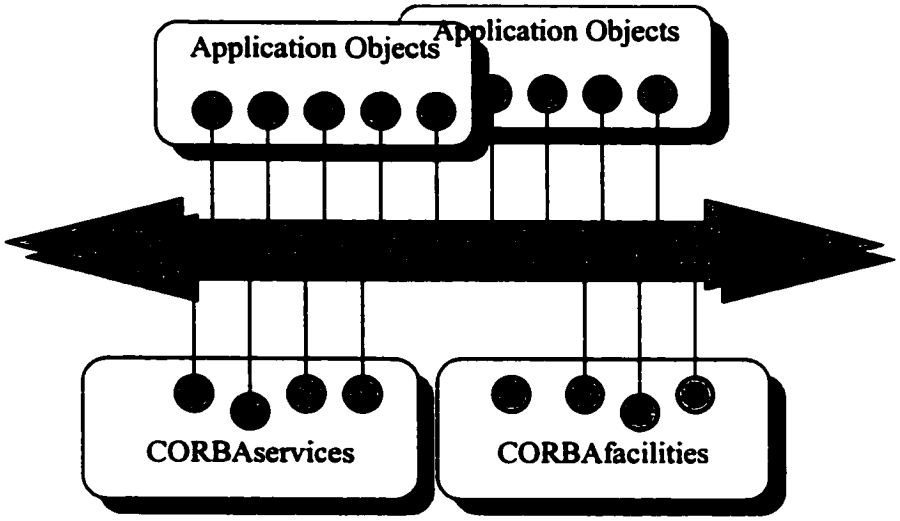

Figure 3 - 1 The Object Management Architecture (OMA).

This section briefly describes the specific features of CORBA and CORBAservices that are of interest to this thesis, without giving a full description of them, taking into consideration the fact that they are quite extensive and freely available from OMG [19].

3.2 Concepts and Terminology

CORBA provides platform-independent programming interfaces and models for portable distributed object-oriented computing applications. Its independence from programming languages, computing platforms, and network protocols makes it highly suitable for the

development of new applications and their integration into current distributed systems and legacy systems.

CORBA introduces a set of new terms besides the ones that are borrowed from other related technologies. The following list contains the most important terms required for understanding CORBA systems [1,6,7,25,26]. Descriptions for each of these terms are provided.

- An *Object Request Broker (ORB)* is a software component that allows clients to make requests on objects across the network, between operating systems, and between programming languages. The ORB receives method invocation requests from the client and delivers them to the target object. If the client and target object exist in separate memory spaces, there is an ORB associated with the client process and a separate ORB associated with the server process for the target object. As part of the Common ORB Architecture, all ORBs, regardless of vendor, hardware platform, or software language implementation, can process standard CORBA requests between clients and server objects.
- A *CORBA object* is a "virtual" entity capable of being located by an ORB and having client requests invoked on it. It is virtual in the sense that it does not really exist unless it is made concrete by an implementation written in a programming language. The realization of a CORBA object by programming language constructs is analogous to the way virtual memory does not exist in an operating system but is simulated using physical memory [1].
- A *client* is an entity that invokes a request on a CORBA object. A client may exist in an address space that is completely separate from the CORBA object, or the client and the CORBA object may exist within the same application. An application that acts as the client in the context of a request may act as the server for another request.
- A *server* is an application in which one or more CORBA objects reside. Clients invoke request upon server objects. A server application may also act as a client to another server.

- A *target object*, within the context of a CORBA request invocation, is the CORBA object that is the target of that request. The CORBA object model is a single-dispatching model in which the target object of the request is determined solely by the object reference used to invoke the request.
- A *proxy* is an object that acts as a representative for a remote object. Client applications may use proxies in their memory space in order to pass client requests to a CORBA object on a remote server. This enables the client application to use the CORBA object in the same manner as local objects, making the network communications transparent to the developer or user.
- A *request* is an invocation of an operation on a CORBA object by a client. A Request flows from a client to the target object in the server, and the target object returns a result if the request requires one.
- An *object reference* is a handle used to identify, locate, and address a CORBA object. To clients, object references are opaque entities. The CORBA object model makes the location of a CORBA object transparent to the caller. Clients use object references to direct requests to objects, but they cannot create object references from their constituent parts, nor can they access or modify the contents of an object reference. An object reference refers only to a single CORBA object, and must be unique within the scope of its domain. The CORBA object model requires that a particular object reference must denote the *same* object throughout the object's lifetime. Once an object is destroyed, all its references must become permanently non-functional.
- A *servant* is a programming language entity that implements one or more CORBA objects. Servants are said to *incarnate* CORBA objects because they provide concrete implementations for those objects. Servants exist only within the context of a server application. In C++ or Java, servants are object instances of a particular class. There can be multiple classes of servant objects associated with a class of CORBA objects, but only one servant instantiation may be associated with a CORBA object at a time. Client applications need not know how servants are implemented.
- *Creation* is the initial instantiation of a virtual CORBA object and its object reference. From a client perspective, CORBA objects are normally created by

invoking normal CORBA operations on a *factory object* [8] within a server. Creation of a CORBA object may or may not include the *activation* of the object.

- *Activation* is the act of starting an existing CORBA object to allow it to service requests. Activation does not imply CORBA object creation since a CORBA object cannot be activated if it does not already exist. Activation may cause the creation of the servant.
- *Deactivation* is the act of shutting down an active CORBA object by removing its association with its servant. Deactivation does not imply CORBA object destruction, as the CORBA object continues to exist as a virtual entity and can be reactivated at a later time. Deactivation may result in the destruction of the servant.
- *Destruction* is the act of removing the CORBA object from existence. All object references to a CORBA object that has been destroyed are invalid.
- *Incarnation* is the act of associating a concrete servant object with a CORBA object so that it may service requests. This requires the instantiation of a servant object on the server and its association with the CORBA object. In other words, incarnation "gives bodily form or substance to" [39] a virtual CORBA object.
- *Etherealization* is the opposite of incarnation and refers to the act of destroying the instance of the servant of a CORBA object. That is, etherealization takes away the "body" of the CORBA object on the server. The CORBA object still exists as a virtual entity and may again be reincarnated when it activated by a client.

It is important to distinguish between the lifecycle of a virtual CORBA object and a concrete servant object. The terms activation, deactivation, creation and destruction are used to refer to the changes in the state of the CORBA virtual object. The terms incarnation and etherealization refer to the association and disassociation of a servant object with a CORBA object, respectively [6]. The lifecycle of a CORBA object is shown in Figure 3-2.

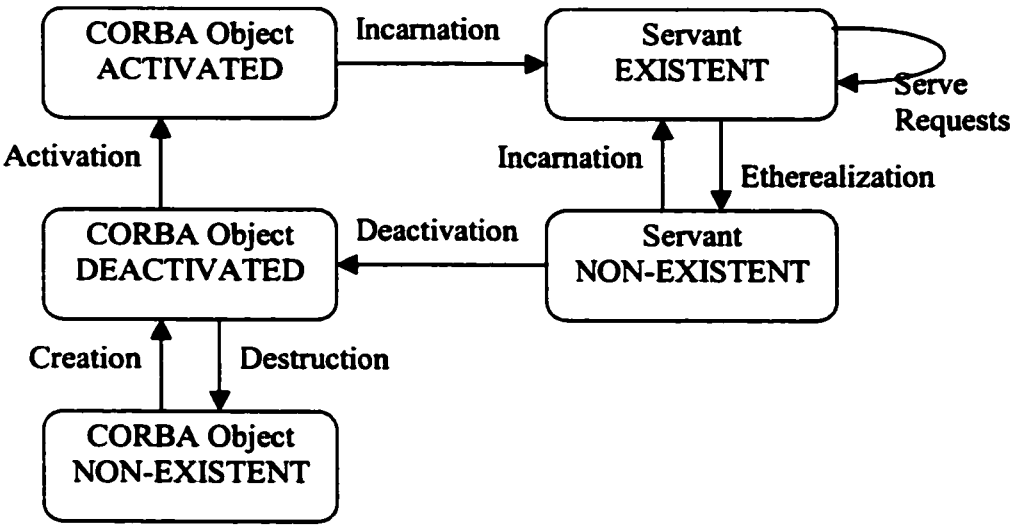

Figure 3 - 2 Life Cycle of CORBA Objects and Servants

3.3 The Object Request Broker

The core CORBA infrastructure is comprised of the Object Request Broker (ORB), its underlying inter-ORB communications protocol, the client-side interface, and the server side interface. These are shown in Figure 3-3 [7].

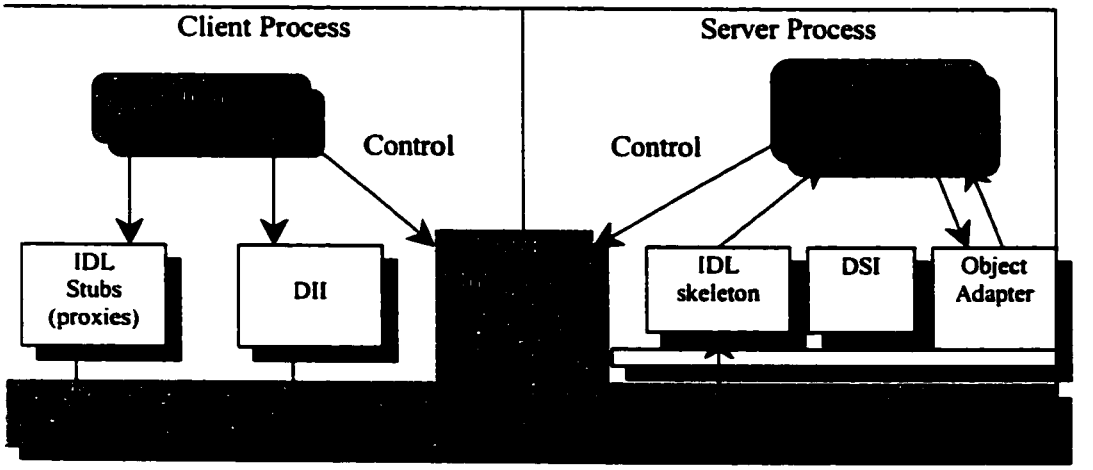

Figure 3 - 3 Client and Server sides of the ORB.

The ORB allows a client to make a request on an object across a network, between operating systems, between different programming languages, and between ORBs from different vendors. In order to accomplish this, a standard protocol is used for message passing between ORBs called the General Inter-ORB Protocol (GIOP). Each ORB implementation is required to support the Internet Inter-ORB Protocol (IIOP) for passing GIOP messages over Transmission Control Protocol/Internet Protocol (TCP/IP) networks [19]. This thesis will discuss only ORBs using IIOP and not other environment-specific inter-ORB protocols (ex. ORBs for real-time systems).

As this thesis presents a Generic Database Adapter the functions and components of the ORB, that are of interest to this thesis, are presented and discussed in the following sections.

3.3.1 The Object Adapter

In CORBA, object adapters serve as the glue between servants and the ORB. The intent of an object adapter, as defined by [8], is to adapt the interface of one object to a different interface expected by a caller. The object adapter enables a client to invoke requests on a servant object without knowing the object's true interface, only its CORBA interface [7]. In C++, servants are instances of C++ objects. They are derived from skeleton classes produced by compiling IDL interface definitions. To implement operations, the servant class must override the virtual functions of the skeleton base class. The C++ servants are registered with the object adapter to allow it to dispatch requests to the servants when clients invoke requests on the CORBA objects incarnated by those servants.

The object adapter fulfills three key requirements of the ORB [7].

1. It creates object references, which allow clients to address objects.
2. It ensures that each target object is incarnated by a servant.
3. It takes requests received by a server-side ORB and further directs them to the servant incarnating each of the target CORBA objects.

In version 2.0 of the CORBA specification, the standard object adapter required by all ORBs is the Basic Object Adapter (BOA). There can be other forms of Object Adapters, such as a Database Adapter that can be used to provide persistence to CORBA objects using an Object-Oriented Database Management System (OODBMS) [7, 25]. In the new CORBA 3.0 specification, the BOA will be replaced with the Portable Object Adapter (POA). The POA specification provides a full suite of features and services intended to allow developers to write scalable, high-performance server applications. This thesis is based on the CORBA 2.0 specification and does not address the functions of the POA, although many of the concepts proposed may be applied to the POA.

3.3.2 Server-side Static Skeleton Interface

In the development of server applications, skeleton classes for servant objects are produced by compiling IDL interface definitions. The servant class implements the operations for the IDL interface by overriding the virtual functions of the skeleton base class. This approach is known as the Static Skeleton Interface (SSI) and is by far, the most common approach for developing server applications. The SSI provides static mapping between the IDL interface and implementation class. The Object Adapter uses this mapping for invoking the incoming request for a CORBA object on its underlying servant object. When using the SSI, the Object Adapter implements the adapter pattern as described in [8] to map an operation request from the IDL interface to the servant implementation class.

The adapter pattern has two possible structures, one using inheritance and one using delegation, to pass an invocation on an interface object to a servant object [8]. Both these patterns can be used in the design of the Static Skeleton Interface for server-side application classes. In developing C++ servants for CORBA 2.0 using Orbix version 2.3 [25,26], these patterns are known as the BOAImpl and the TIE approaches, respectively. These patterns only apply to developing servants for server applications; they do not apply to client applications.

In the BOAImpl approach, shown in Figure 3-4, the inheritance form of the adapter pattern is used. The IDL compiler produces the CORBA proxy class that is used by the client application, and the SSI class for the server application. This SSI class is called the "BOAImpl" class in Orbix as the class name is suffixed with BOAImpl. The C++ servant class must inherit from the BOAImpl class to provide an implementation for the CORBA object.

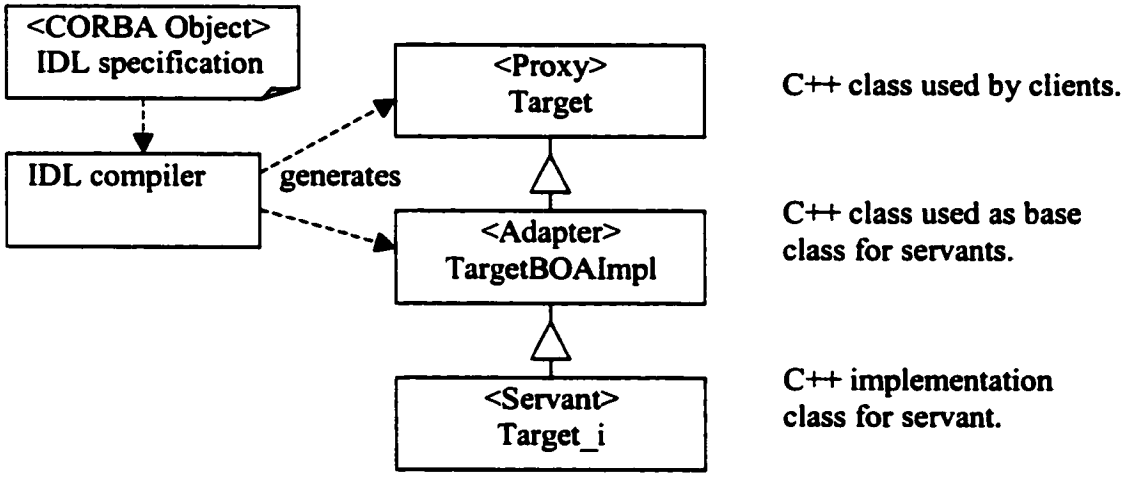

Figure 3 - 4 BOAImpl Approach, the inheritance form of the adapter pattern.

The second form of the adapter pattern uses delegation to pass invocations from the interface to the implementation object. In Orbix, this is achieved by the use of an adapter called a "TIE" object [6,7]. The TIE class is generated by a macro from the servant class. The TIE object takes the servant implementation object as part of its constructor. It then delegates all incoming requests for the CORBA object to the servant. The TIE approach to implementing server objects is shown in Figure 3-5.

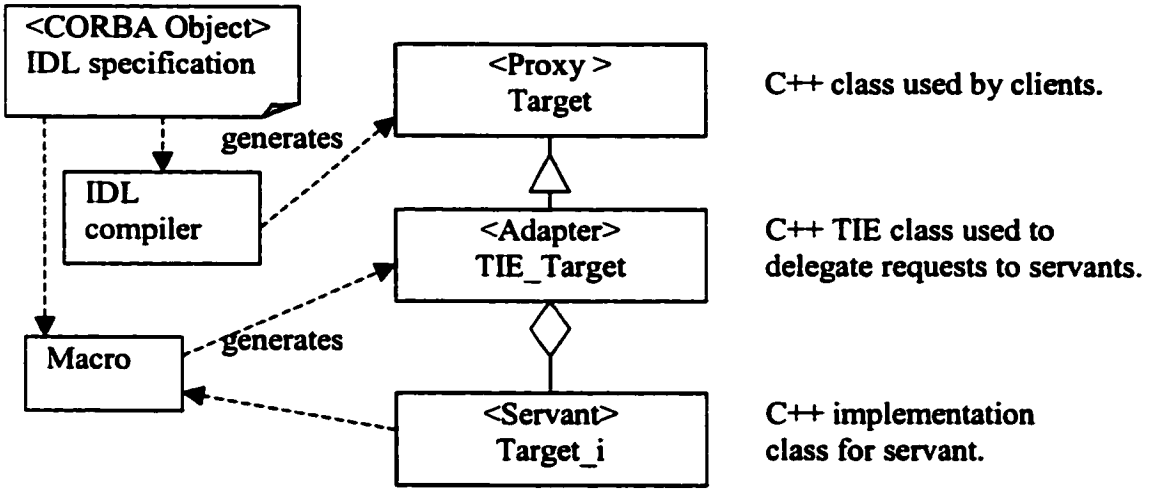

Figure 3 - 5 TIE Approach, the delegation form of the adapter pattern.

The advantage of the TIE approach over the BOAImpl approach is that the servant class does not have to inherit from any BOAImpl classes. This allows the reuse of existing classes or the use of classes that must inherit from a hierarchy that is incompatible with the BOA hierarchy. The drawback of the TIE approach is that servant incarnation is more complicated because a TIE object, which comprises the servant object, must also be created. Both the TIE and BOAImpl approaches can be used in the same implementation but applications are usually developed using one approach exclusively.

3.3.3 Client-side Static Invocation Interface

Client applications using CORBA can be written in any programming language for which a compatible ORB is available. The design and implementation of a client application is related to the server application only through the use of a common IDL specification. The use of stub code generated from an IDL specification is known as the Static Invocation Interface (SII) for client applications. The SII provides the client application with a set of CORBA classes that it can process. Within the client application, these CORBA objects are accessed in the same manner as native objects. The client typically uses the IDL

generated stub code to instantiate a local proxy for a CORBA object. The proxy exists within the client address space and is referenced by the client application as a native object. The proxy acts as a representative for a remote CORBA object that is incarnated by a servant on a remote server. The proxy maintains a CORBA object reference to the target object on the remote server. Only one proxy object exists in the client address space for a single remote target object, regardless of the number of local references to the target object. An example of a client proxy is shown in Figure 3-6.

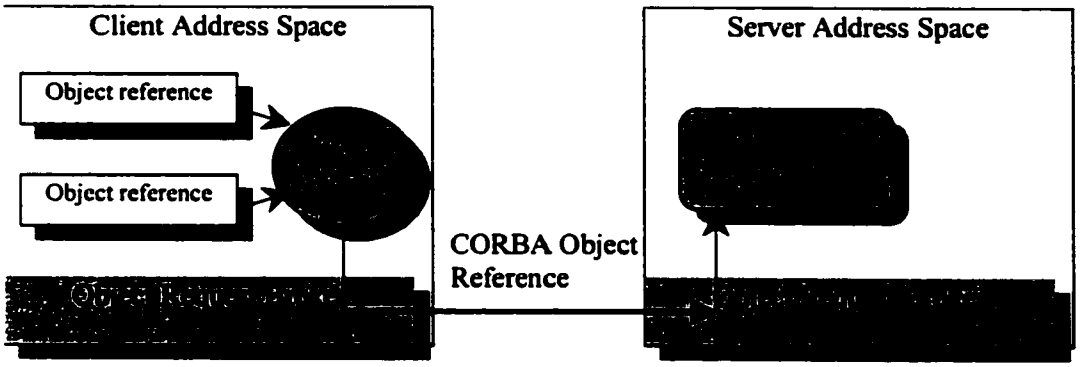

Figure 3 - 6 Client proxy for a target object on a remote server.

The client-side ORB is responsible for processing all invocations on the client object by implicitly creating an IIOP request message, marshaling the request, transmitting the request over the network to the remote server, receiving and unmarshaling any return values, and passing the result back to the client application. The transmission of operation or attribute requests on a CORBA object over a network is transparent to the client.

3.4 The Interface Definition Language (IDL)

The Interface Definition Language (IDL) is used to describe the interfaces of objects in CORBA. The language is part of the OMG specification [19] for CORBA and is also an International Standards Organization (ISO) standard [11]. IDL allows interfaces to be

defined independently of the languages used to implement and use these interfaces. IDL is not a programming language and can not be used to implement the functionality of an interface. The IDL specification must be mapped to a programming language such as C++, Smalltalk, Java, COBOL, or Ada, to provide an implementation for an interface. This mapping is usually provided as part of the CORBA development environment for a specific language. This section presents the basic features of IDL required to support the concepts proposed in this thesis. A detailed description of the IDL standard can be found in [6, 7, 19, 27].

The OMG IDL is CORBA's fundamental abstraction mechanism for separating object interfaces from their implementations. Because IDL describes interfaces but not implementations, it is a purely declarative language. There is no way to write executable statements in IDL, and there is no way to specify the object state (execution and state are implementation concerns). The core concepts supported by IDL are interfaces, operations, exceptions, and inheritance. These are presented in the following sections.

3.4.1 Interfaces

Every CORBA object has exactly one interface, but there can be an unlimited number of objects of the same interface in a distributed system. In this respect, IDL interfaces correspond to class *definitions* and CORBA objects correspond to class *instances*. Interface instances can be implemented in a single address space, in multiple processes on the same machine, or in multiple processes on different machines. A CORBA object is an interface instance that is remotely addressable. As such, an IDL interface defines the smallest granularity of distribution supported by CORBA. This determines the extent to which an application can be distributed over different physical address spaces; functionality can only be distributed if there is an interface defined to access that functionality.

The syntax of IDL interfaces is deliberately simple because it must support mapping to multiple implementation languages, and because it does not have to support execution. IDL interfaces are strictly declarative and each forms a namespace for the following types of constructs enclosed within the interface definition [6]:

- Constant definitions
- Type definitions
- Exception definitions
- Attribute definitions
- Operation definitions

Note that interface definitions cannot be nested within another interface. However, multiple interfaces can be defined within a shared namespace called a *module*. Modules combine related interface definitions into a logical group and prevent pollution of the global namespace. The use of modules to scope the interface definition is essential to maintaining the global namespace in large heterogeneous distributed systems. Modules can contain other modules so it is possible to create nested hierarchies using the namespaces of modules. Interfaces in separate modules can be addressed by using the module name as part of the name up to the global level; this is referred to as the *fully scoped name* of an interface.

IDL interfaces only define the interface to an object and do not imply anything about the implementation. This is characterized by a few subtleties of the interface definition for a CORBA object [7]:

- All definitions made in an IDL interface are public. There is no concept of private members in IDL. If some aspect of the object is not to be made public, then it is simply not included in the IDL definition.
- IDL interfaces do not have member variables. IDL does not support the concept of a variable because member variables, or attributes of an object, store state information. The state of an object is considered an internal of the object, which makes the state an implementation concern, not part of the interface.

- Interface attributes do not necessarily map to object member variables as they could be implemented in a number of ways: a static data value, an element of an array, a field in a file, etc. As such, an interface attribute cannot be assumed to represent the state of the CORBA object, although it could be used to access state information in the object's implementation.
- The parameter-passing mode and name of each argument in an operation must be specified. Return types must be specified for all operations.
- Interface names become type (class) names once they are declared.
- Interface instances can be passed as parameters to operations or return values between objects in separate address spaces.
- There are no constructors or destructors for an interface. Clients cannot create CORBA objects of any interface on any arbitrary server in the system. Object creation is a responsibility of the server. The use of the factory pattern to export an interface for creating objects is common in server design.
- A CORBA object must have a unique identity for a given interface on a given server. This identifier is referred to as the Object's *marker*. The marker is normally set at object creation and may or may not be made public through the object's interface. A CORBA object is considered to exist as long as a server can incarnate its servant for an object reference indicating the object's marker, interface and server.

3.4.2 IDL Types

IDL provides a set of basic types that form the primitive components of an interface. These basic types, shown in Table 3-1 [6], correspond to common primitive types found in implementation languages such as C++ or Java, and are described in detail in [19]. The most significant basic type is the universal container type `any`, which can be used to transmit IDL types that are unknown at compile time. A value of type `any` can contain a value of any other IDL type, such as `long`, a user-defined type, object references, or another value of type `any`. Values of type `any` are type-safe because they contain a field

describing the type of the value and enforce run-time type checking on the extraction of the any value.

Table 3-1 IDL Basic Types.

Type	Range	Size
short	-2^{15} to $2^{15}-1$	≥ 16 bits
long	-2^{31} to $2^{31}-1$	≥ 32 bits
unsigned short	0 to $2^{15}-1$	≥ 16 bits
unsigned long	0 to $2^{32}-1$	≥ 32 bits
float	IEEE single-precision	≥ 32 bits
double	IEEE double-precision	≥ 64 bits
char	ISO Latin-1	≥ 8 bits
string	ISO latin-1, except ACSII NUL	Variable-length
boolean	TRUE or FALSE	Unspecified
octet	0-255	≥ 8 bits
any	Run-time identifiable arbitrary type	Variable-length

IDL also supports the following user-defined types:

- Type definition statements (typedef) used to create new user-defined named types for existing types;
- Enumerated types (enum) with ordinal values increasing from left to right, but otherwise undefined in value;
- Structures (struct) containing one or more named members of arbitrary type;
- Unions (union) with a discriminator to indicate which member is active;
- Arrays (typedef Type name [ ] [ ]) of multiple dimensions with declared bounds;
- Sequences (typedef sequence<Type, bound> name) are variable-length list of any element type that can be bounded or unbounded;
- Constant and literal (const Type) declarations of integer, floating-point, and boolean constant named values.

3.4.3 Interface Attributes

Attribute definitions can occur only as part of an interface definition. An IDL attribute is defined as a symbolic name with its type declared as an IDL basic type, user-defined type, or an object as previously declared in an IDL Interface [6,7,19]. An attribute defines a pair of operations that the client can call to send and receive a value. These attributes do not have to be implemented as member variables of the servant class that incarnates the CORBA object defined by the interface. Attributes can be viewed as operations to access values or objects through the object reference to the CORBA object in which they are declared. Attributes can be declared as readonly, which specifies that the client can get the current value of the attribute, but not set it. Servant objects usually provide operations to get and set (if not readonly) attribute values. IDL attributes vary from IDL operations, defined below, in that they cannot raise user-defined exceptions, though they can raise system exceptions.

An attribute of an interface can be of that interface type (a self-referential type) provided that a forward declaration is made for the interface name within the module. Forward declarations are required wherever an interface is used as an attribute or operation argument type before the actual interface is defined within a module.

3.4.4 Interface Operations

Operation definitions can occur only as part of an interface definition. Each operation name within an interface declaration must be unique [6,7,19,34]. IDL does not allow overloading of operations. Each operation declaration must contain:

- a return result type;
- an operation name; and
- zero or more parameter declarations, each comprised of a directional attribute, a type declaration, and an argument name.

The directional attribute determines the parameter-passing mode as one of in, out, or inout, where:

- in indicates that the parameter is sent from the client to the server.
- out indicates that the parameter is sent from the server to the client.
- inout indicates that a parameter is initialized by the client, sent to the server, possibly modified and sent back to the client.

These directional attributes are required to reduce network traffic by only passing parameters in the direction required, and to determine the responsibility for memory management in the client and server.

Operation calls from a client to server are normally synchronous in CORBA. The client process will block until the operation is executed on the server object, even if no return value is expected. For applications where this is not desired, IDL permits an operation to be declared as oneway. A oneway operation must adhere to the following rules to ensure that no return values are possible:

- It must have a return type void.
- It must not have any out or inout parameters.
- It must not have a raises expression for user exceptions.

The CORBA specification for how an ORB must implement the oneway operation does not guarantee that it will arrive at the server or that it will not block the client [6]. As blocking is an implementation issue, it may be preferable to implement non-blocking operations using the Dynamic Invocation Interface. This would reduce the complexity and uncertainty of the IDL interface. There are also other CORBA services available that can provide non-blocking behavior for clients.

An IDL operation may raise an exception to indicate that an error has occurred. Exceptions are defined as part of an interface or module and may include one or more member variables of specified types. These are treated as arguments returned to the operation caller that attempts to indicate the reason for the error. As well as user-defined

exceptions, CORBA defines a set of standard exceptions that may indicate system errors, such as inter-ORB communication failure or that a CORBA object is non-existent.

3.4.5 Exceptions

IDL operations or attribute implementation functions can raise exceptions to indicate that an error has occurred [7]. CORBA defines two types of exceptions:

- User-defined exceptions
- System exceptions

User-defined exceptions are exceptions that can be defined as part of the IDL specification by defining the exception structure and by adding an IDL *raises* clause to the specific operations definitions. An IDL attribute can not throw a user-defined exception. The user-defined exception is a data structure that contains member fields, whose type can be one of the pre-defined IDL types (string etc.).

```
module Finance {
        interface Account {

                exception WithdrawalFailure {                          // exception
                        string reason;
                };
                exception LimitExceeded {                              // exception
                        string reason;
                };

                const long a = 1;                                  // constant
                typedef long NewValue;                             // user-defined type

                attribute long accountNumber;                          // attributes
                readonly attribute string accountName;

                void    withdraw(in NewValue v) raises(WithdrawalFailure);     // operation
        };
        interface CheckingAccount : Account {

                const long a = 2;                          // redefined constant
                typedef unsigned long NewValue;            // redefined typedef

                // Can not redefine superclass attributes or operations
                long    transfer(in NewValue v) raises(LimitExceeded);
                void    setAccountName(in string n);
        };
        interface SavingsAccount : Account{
                ...
        };
        interface PremiumAccount : CheckingAccount, PremiumAccount{
                ...
        };
};
```

Figure 3 - 7 Interface inheritance and user-defined exceptions.

System exceptions are a set of predefined CORBA exceptions that all IDL operations can throw.

3.4.6 Interface Inheritance

Another important feature of IDL is the support for Inheritance [6,7,19,25]. Scoped resolution for inheritance works the same as for C++. The root of the inheritance tree for all IDL objects is implicitly defined as Object. As inheritance gives rise to polymorphism, a derived interface can be treated as if it were a base interface. And

because all IDL interfaces directly or indirectly inherit Object, all interfaces are type-compatible with type Object. This allows IDL operations to be specified as passing Object type parameters and return values. It is then up to the client to narrow the object reference to the desired subclass type. This is a common technique in CORBA and is used in order to accomplish the Generic Database Adapter presented in this thesis.

The CORBA IDL also supports multiple-inheritance. This is useful for interface aggregation. The usual type compatibility rules apply when passing CORBA objects as a base interface instead of its derived interface. The limitation on multiple inheritance is that operations and attributes must not be inherited more than once from separate base interfaces to avoid conflicting type definitions. Multiple-inheritance is a convenient technique for adding an interface with a new service to an interface that is part of a separate hierarchy. Because IDL only provides interface inheritance, it does not imply that the use of multiple inheritance in the interface requires multiple inheritance in the under-lying implementation.

The last important aspect of IDL that is relevant to this thesis is the distinction between the interface inheritance hierarchy and the implementation hierarchy. IDL inheritance only applies to interfaces, the implementation options are completely unconstrained. The structure of the IDL hierarchy need not be reflected in the implementation. The inheritance hierarchy of the implementation classes can be completely independent of the interface hierarchy.

3.4.7 The Implementation Repository

The Implementation Repository is the component of an ORB that is typically responsible for maintaining registration information about servers and their activation mode. It is also involved in establishing the initial connection between clients and servers.

An Implementation Repository has the following responsibilities.

- It maintains a registry of known servers.
- It records which server is currently running on which host and at which port number.
- It starts servers on demand if they are registered for automatic start-up.

The Implementation Repository also maintains a mapping from a server's logical name to the file name of the executable code that implements that server. This registry may also include other activation information such as the host and specific port number for the server to use, security information, and the activation mode for launching the server. This mapping is usually set when the server's executable file is registered with the Implementation Repository.

An Implementation Repository can support servers that are launched on different hosts. Each server maintains configuration information indicating the host and port number on which its Implementation Repository resides. The set of servers and hosts supported by the same interface repository is known as the *location domain.*

The Implementation Repository can also launch a server if it is not active and is registered for automatic registration. Servers can also be launched when the ORB is started or manually started. The server can be set to run indefinitely, until it is manually shutdown, it exits due to an error, or it can be set to terminate on a timeout after a period of inactivity. In any case, a server can be launched using different activation modes that determine how the server process is implemented by the underlying operating system.
The following primary activation modes are supported [7]:

- Shared activation mode. This is the default activation mode and is used by most applications including the framework presented in this thesis. In this mode, all objects with the same server are managed by the same process on a given host.
- Unshared activation mode. In this mode, individual objects of a server are registered with the Implementation Repository and a single process handles all

invocations for an individual object. The server process is activated by the first invocation of that object and one process is created per active registered object. The motivation for this mode is to support objects that cannot or should not run in the same process.

- Per-method-call activation mode. In this mode, individual operation or attribute names are registered with the Implementation Repository. Each operation invocation results in the creation of an individual process, which is terminated when the operation is complete.

The shared and unshared activation modes also support different sub-modes:

- Multiple-client. One server process is launched to serve requests from all client processes from an unlimited number of users. This is the default sub-mode used by most applications, including the framework presented in this thesis.
- Per-client. This sub-mode launches one process for a given server for all client processes from the same user. It launches a separate server process for different users. This requires a security facility to verify each client's user name.
- Per-client-process. This sub-mode launches a separate server process for different users. It also launches a separate process for a given server for each client process from the same user.

3.5 CORBA Services

The CORBAservices are sets of optional utilities defined in IDL that form part of the Object Management Architecture specified by the OMG [19,25]. They extend the core CORBA specification by providing commonly used services for CORBA objects and distributed applications. ORB vendors may provide an implementation for any of these services or a developer can use them as a framework in designing an application. Each CORBA service must adhere to the IDL specification provided by the OMG to support

inter-operability between implementations. As such, CORBAservices are an example of the re-usability and inter-operability of open applications that can be provided by CORBA.

A short description of some of the CORBAservices available in the CORBA 2.0 specification is provided. A detailed description is then presented for the CORBAservices used for the implementation of the Generic Database Adapter (GDA).

- **Persistent Object Service**: defines a general framework for designing how a CORBA object can be made persistent, allowing it to maintain its state across multiple invocations. It defines how a database and an object should communicate to store and restore the object to and from the database.
- **Query Service**: defines a simple collection type and a framework for making queries over the objects stored in such a collection. These queries use the Sequential Query Language (SQL) for Relational Databases and the Object Query Language (OQL) for Object Oriented Databases. A query is passed as a string, along with an indication of the type of the query language.
- **Concurrency Control Service**: facilitates controlled concurrent access of multiple threads or transactions to a CORBA object in order to prevent its state from becoming corrupted. It provides a number of IDL interfaces to a locking facility that an object can use to control concurrent access.
- **Object Transaction Service (OTS)**: provides a two-phase commit protocol for transactions that span multiple operations on an object. It controls the commit and roll back operations for these transactions to ensure the integrity of the object's state.
- **Naming Service**: allows a client in a distributed system with multiple objects residing on multiple servers or hosts to easily find an object it requires using its name previously registered by the server with the Naming Service. A hierarchical name that is independent of the server or host on which the object resides, can be given to that object by the server during the registration process. Any time a client

intends to get a reference to a specific object, it resolves the name of the object using the Naming Service to receive a CORBA object reference.

- **Event Service**: allows a client or server to distribute an event message to any number of objects that are interested in that event, through an 'Event Channel'. Rather than communicating directly with one or more objects by invoking operations on them, the client sends that event using the Event Service, thus avoiding a direct communication between the sender and receiver.
- **Security Service**: restricts access to authorized clients and restricts each client in the set of operations that it can invoke on objects or groups of objects. The Security Service defines a general security framework that can be implemented in many ways, including through the use of access control lists or capabilities.
- **Trading Service**: allows a client to find an object that is registered with the Trading Service based on a set of properties and searched for by specifying a set of constraints, instead of specifying the desired object by name and using the Naming Service. These constraints may include the interface of the object, the type of the object, the values of one or more of its attributes, or a relationship with other objects.
- **Externalization Service**: allows an object's data to be written to and read from a stream of bytes so that is can be copied to another storage system.

The CORBAservices important to this thesis are: the Persistent Object Service, the Query Service, Object Transaction Service, and the Concurrency Control Service. The Object Transaction Service, and the Concurrency Control Service do not have to be implemented in order to accomplish the goal of this thesis, because the normal concurrency control and transaction control provided by the Database Management System (DBMS) will be used instead. All the other services are discussed in detail here as they have a significant role in the creation of the Generic Database Adapter.

3.5.1 The CORBA Query Service

The CORBA Query Service as defined by OMG must allow querying against any objects, with arbitrary attributes and operations [6,19,25]. The main purpose of the query service is to allow clients to invoke queries on arbitrary collections of other objects. The queries are predicat based and may return collections of objects. They can be specified using object derivatives of SQL (SQL-92) and/or other styles of object query languages (OQL-93). The term query denotes general manipulation operations including selection, insertion, updating and deletion. This section discusses the CORBA Query Service and why it was found not suitable for the implementation of a Generic Database Adapter.

The Query Service can be used to return collections of objects that may be:

- Selected from source collections based on whether their member objects satisfy a given predicate.
- Produced by query evaluators based on the evaluation of a given predicate. These query evaluators may manage implicit collections of objects.

In particular for environments using database systems – object oriented, relational etc. – and for other systems that store and access large collections of objects, the Query Service must map well to theses native systems' internal mechanisms for specifying collections and using indexes. To maximize this, the Query Service is based on existing standards for query and extended when necessary to accommodate other design principles.

The Query Service design provides an architecture for a nested and federated service that can coordinate multiple nested query evaluators.

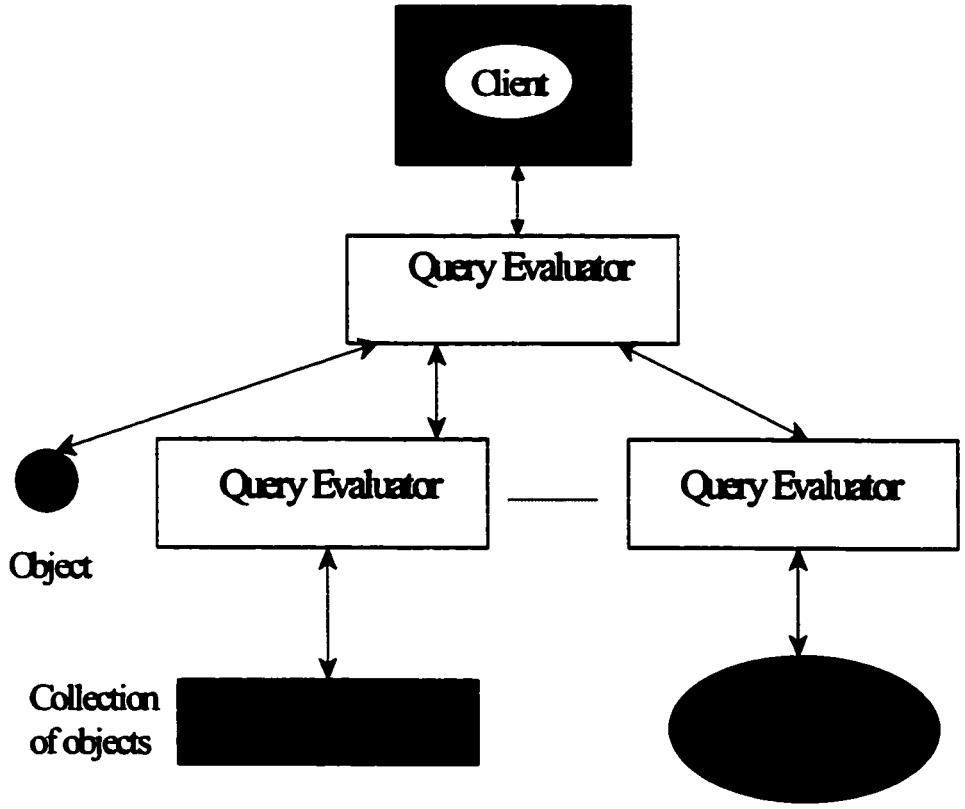

Figure 3 - 8 Query Service Architecture.

Objects may participate in the Query Service in two ways:

- As a CORBA object
- As member of a collection of CORBA objects

In the first case the Query Evaluator is responsible for evaluating the query predicate and performing all query operations by invoking operations on those objects through its published IDL interfaces. In the second case the Query Evaluator passes the query predicate to the collection (which is a Query Evaluator) which then evaluates the predicate and performs query operations on an appropriate object, receives a result, combines such results with all other participating object results, and returns this to the

client. The collections themselves have to be Queryable Collections, which means that they serve as both the result of the query and as a scope for another query.

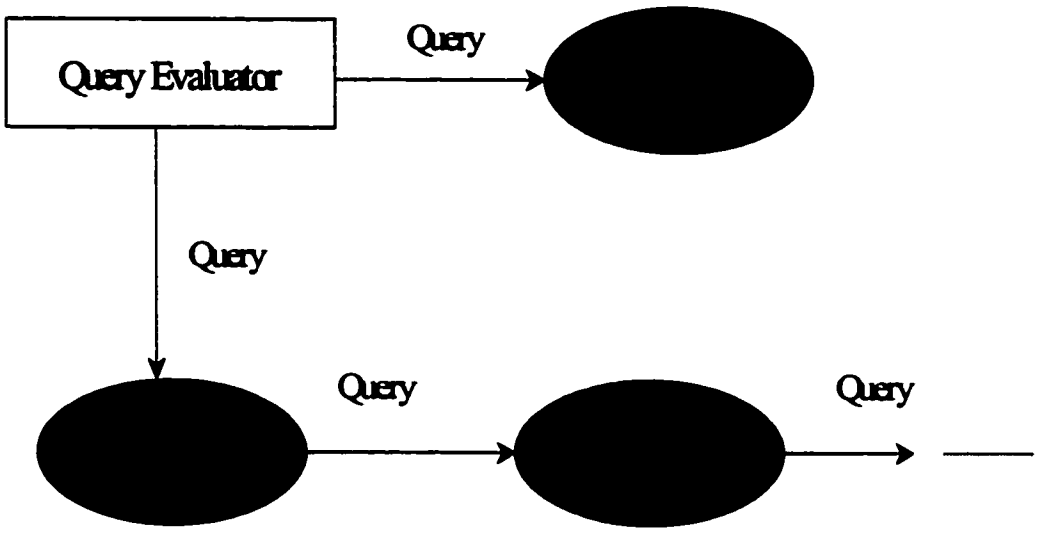

Figure 3 - 9 Queryable Collections.

In order to allow a client to iterate through the members of a collection an interface Iterator is defined. Depending on the type of the collection (ordered, unordered) the members of the collections will be either visited in a defined order or in any order, but they will be visited only once. If members are added and removed from collections at the same time, the iterator can become invalid and it has to be reset before being used again (exception safe cursors).

The Query Service specification does not define evaluation, indexing or optimization mechanisms, it simply provides a mechanism for passing the query to systems that have these mechanisms already implemented and allows their optimization to take effect.

Because the OMG specifications for the implementation of a Query Service are too extensive but also too restrictive at the same time, and the query mechanism for the Generic Database Adapter requires different operations or operations having different signatures in order to query collections of persistent CORBA objects, the OMG standard for the Query Service was considered unsuitable. In turn a specific query mechanism, which is described in the following chapters, has been implemented as part of the Generic Database Adapter.

3.5.2 Persistent Object Service

Many distributed applications require that the state of individual objects to be consistent between operation invocations. If a servant object is etherealized and re-incarnated as part of a CORBA object's lifecycle, its state must be made persistent [7]. Since the Generic Database Adapter is intended for the general case of multi-tier distributed applications, maintaining persistence of a CORBA object is one of the main features in managing the persistent state of a CORBA object. The Generic Database Adapter must provide a means of retrieving and storing a CORBA object's persistent state. As such, a persistence mechanism for the Generic Database Adaptor is required. A number of techniques for providing a persistence mechanism were investigated, including the CORBA Persistent Object Service, but were discarded because they failed to meet some criteria of the Generic Database Adaptor. This section discusses the Persistent Object Service (POS) and why it was found not suitable. This section also discusses the concept of a database adapter and its advantages in supporting the implementation of a Generic Database Adapter.

CORBA objects are considered to have a *dynamic state* and a *persistent state.* The dynamic state is considered to be the state of transient variables within a servant's implementation code. These are internal variables that do not determine the state of the object. The persistent state is comprised of the variables and attributes of a CORBA object that must be maintained while the object is deactivated, that is, while no servant exists to maintain attribute values, so that its state can be later restored. This means that a client has to observe the object in the same state, if the state of the object was not changed between two different moments in time, even if the server providing the object was shut-down and restarted.

The principle concepts of persistence in CORBA used by the Persistent Object Service are defined by [5]:

- A *persistent object* (PO) is an object, whose lifetime can exceed the lifetime of the application that uses it.
- The *persistent state* of a PO is the set of values corresponding to the n attributes of the object considered relevant to the object's state.
- The *dependencies* of a persistent object are the set of all objects targeted by a reference from the PO.
- The *transitive closure of dependencies* of a PO is the set of all objects reachable from the PO via object references transitively over all nested dependencies.

The CORBA Persistent Object Service (POS) is an abstract framework for designing how a CORBA object can be made persistent, allowing it to maintain its state across multiple activations [7]. It defines interfaces that can be used to encapsulate the mechanisms for making objects persistent. The implementations for these mechanisms are not specified by the POS but can include Relational Database Management Systems (RDBMS), Object-Oriented Database Management Systems (OODBMS), and flat files. The OMG deliberately leaves the functionality core of the Persistence Service unspecified [5]. The intention of the POS is to provide a general architecture to store the state of CORBA objects without relying on any implementation technologies while also minimizing duplication of functions with other CORBA services [17, 18, 19]. The implementation of

the POS as specified by the OMG has been attempted many times, with limited success [5, 6, 7].

The OMG specification for the architecture of the Persistent Object Service (POS) is shown in Figure 3-10 [5]. It requires objects that are to be made persistent to implement a standard Persistent Object (PO) interface that provides a Persistent Identifier (PID) attribute. The PID is used by a Persistent Object Manager (POM) to identify the persistent state of the object in the Persistent Data Service (PDS). The PDS provides the interface between the PO and the underlying data store. Any persistence operation on the PO is passed to the POM, which then uses the object's PID to access its persistent state using the PDS. The transfer of data between the PO and the PDS is provided by a *protocol*. This protocol could be one of the OMG specified protocols: Direct_Access (PDS_DA) protocol, ODMG-93 protocol, Dynamic Data Object (DDO) protocol, or some other application dependent protocol [7]. Because the POS is only a general architecture for persistence, many parts of the architecture are not standardized by the OMG [5].

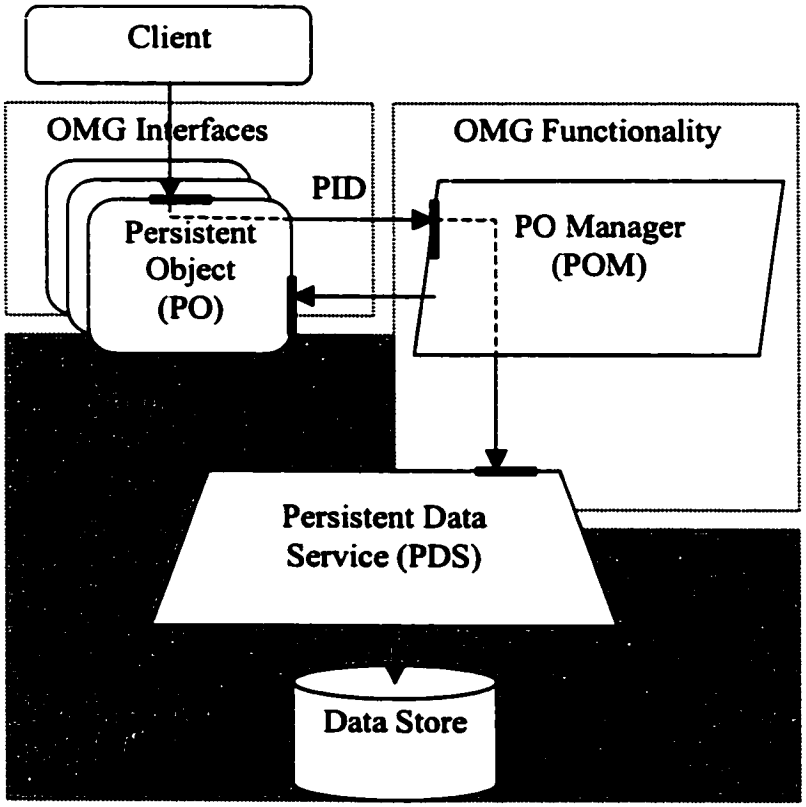

Figure 3 - 10 Architecture of the OMG Persistent Object Service.

The OMG specification defines only the interfaces for the PO, POM, and PDS, all of which must support five common operations: *connect()*, *disconnect()*, *store()*, *restore()*, and *delete()* [5]. These interfaces can be exported by the PO to allow clients to explicitly manage the object's persistence. Conversely, these operations can be hidden from the client and used by the server process to transparently manage the object's persistent state. The approach used affects how the object's persistent attributes are accessed. One way is to equip persistent objects with two IDL interface operations, *save_state()* and *load_state()*, that defines how to store and load the object's persistent attributes compactly. This approach requires public access to the attributes that define the object's persistent state. Another approach is to directly access the attributes of the servant object that implements the CORBA object. This requires that the servant class provide a *friend* construct [8, 9] that allows access to its attributes by another server object, such as the

POM. The direct access approach allows servant attributes not exposed in the IDL interface to be included as part of the object's persistent state. The decision to make all of the object's persistent attributes public or not determines which approach to take in implementing the POS.

The aim of the POS is to use standard IDL interfaces that allow it to be used in conjunction with other CORBAservices. Implementations of the POS that successfully provide this level of reuse have been unsuccessful due to the lack of OMG standardization of the underlying components [5]. Other approaches to providing persistence have been more successful but resulted in non-standardized implementations. Because the Generic Database Adapter requires different operations or operation having different signatures, in order to manage the persistent state of a CORBA object, it cannot be constrained to the standard interfaces provided by the POS. This makes the OMG standard for the POS too restrictive for the Generic Database Adapter.

Another common approach to providing persistence for CORBA objects is the use of a *Database Adapter* [7]. A Database Adapter provides a virtual memory approach for incarnating CORBA servants. The servant that implements a CORBA object is also a database object that is automatically made persistent by a Database Management System (DBMS). This approach uses a three-tiered architecture for providing CORBA objects to clients. In this approach, shown in Figure 3-15, the CORBA server acts as a server to clients and is itself a client to the DBMS database server. The DBMS is responsible for swapping servant objects in and out of the CORBA server memory space and maintaining persistent storage of all of the servant's attributes and dependencies. This approach is considered the best starting point for implementing the Generic Database Adapter.

4 Database Systems

In any type of organization, considerable resources and activity are dedicated to the gathering, filing, processing and exchange of data, based on well established procedures in order to achieve specific goals [38]. This chapter presents an overview of the Database Management System paradigm and the most important Database Management Systems architectures that are used nowadays throughout the economy (financial services, computer industry, health sector, transportation sector, etc.).

A database system is basically a computerized record-keeping whose overall purpose is to store information and to allow users to retrieve and update that information on demand [42]. The information stored, retrieved, or updated represents anything that is of significance to the individual or organization. A database system involves four major components:

- Data (or information),
- Hardware,
- Software, and
- Users

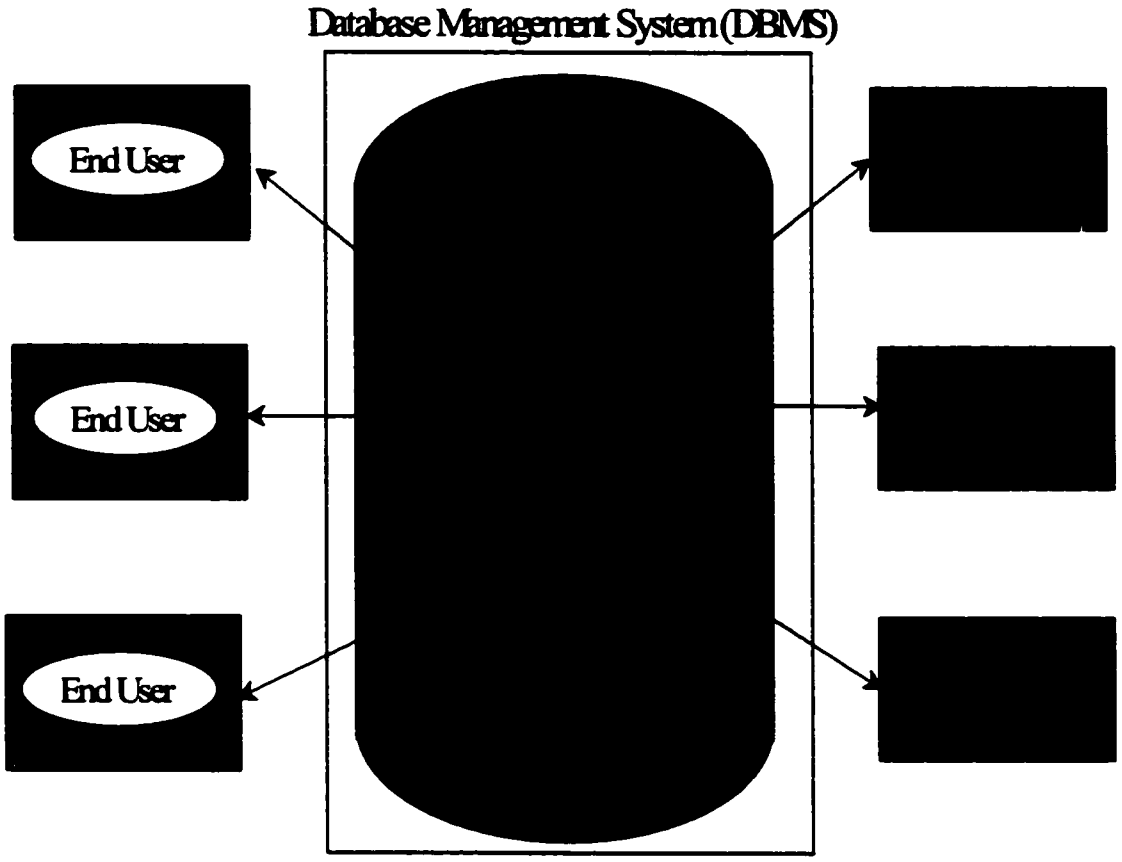

Figure 4 - 1 Simplified Representation of a Database System.

The data in a database must be both integrated and shared [38]. Integrated means that, the database can be seen as a unification of several distinct files, with any redundancy amongst those files already eliminated. Shared means that, individual pieces of data in the database can be shared among different users, which can have access to the same piece of data, possibly at the same time (concurrent access), and possibly for different purposes. As a result of these two features, any given user will typically be concerned only with some small portions of the database, which means that the same database will be perceived by different users in many different ways.

The hardware components of the system consist of the following elements:

- The storage volumes are made mostly of magnetic disks, which are used to hold the stored data, together with the associated I/O devices, device controllers, I/O channels etc.
- The hardware processors and associated main memory that are used to support the execution of the database system software.

There are three broad classes of database users:

- The application programmers responsible for writing database application programs which access the database by issuing appropriate requests (typically SQL statements) to the DBMS,
- The end users who interact with the system from online workstations or terminals, and
- The database administrators or DBAs which are responsible with the maintenance of the database itself.

Besides the fact that a database is seen as a collection of persistent data, it is also seen as a collection of "given facts" which correspond to true propositions that evaluate to true or false, unequivocally (ex. Supplier S1 is located in Ottawa) [42]. Generally databases are based on a logical element called the data model, which is an abstract self-contained, logical definition of the objects, operators etc., that together constitute the abstract machine with which users interact. The implementation of a given model is a physical realization on a real machine of the components of the abstract machine that together constitute the model. There are different types of data models:

- The relational data model (ex. for relational databases), and
- Ad-hoc data models (ex. for object oriented databases).

4.1 The Database Management System

Between the physical database, which is a collection of persistent data used by the application, and the users of the system, is a layer of software named the database

manager, or database server, but most commonly, the Database Management System (DBMS) [42]. All requests for access to the database (adding and removing files or tables, retrieving data from and updating data in such files or tables, etc.) are handled by the DBMS, which result in the shielding of database users from hardware level details. Thus the overall purpose of DBMS is to provide the user interface to the database system.

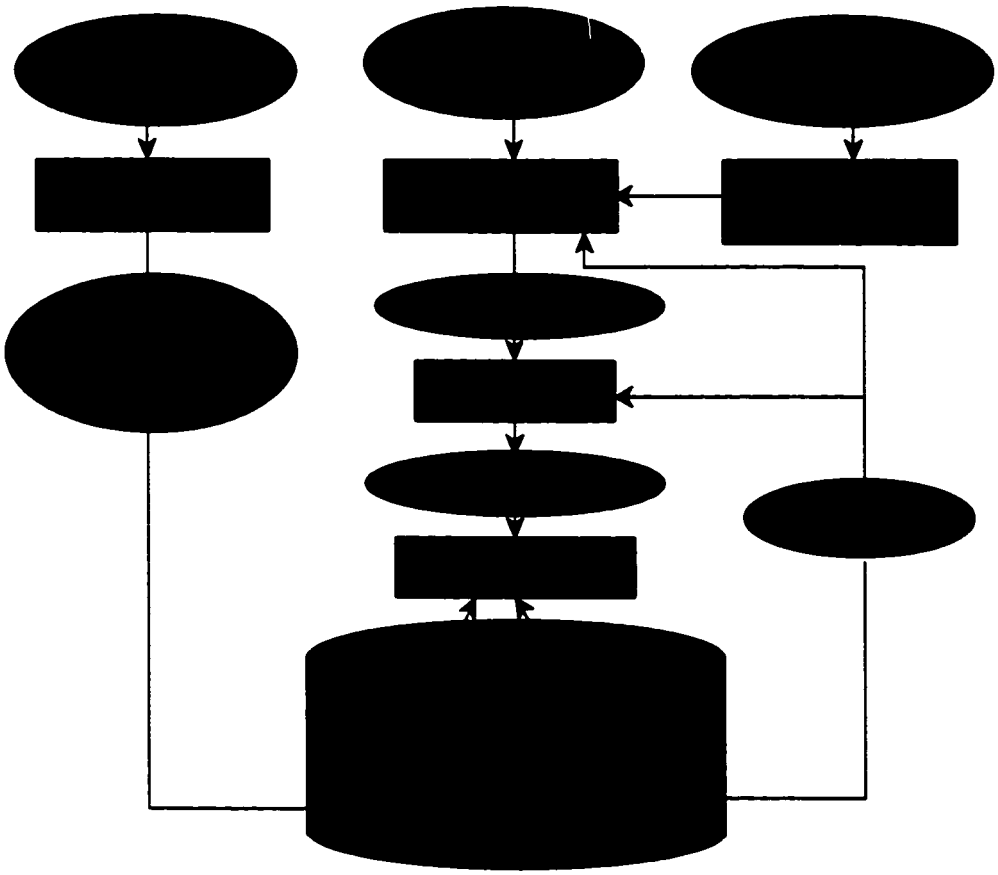

Figure 4 - 2 Major DBMS functions and components.

The functions presented in figure 2-5 will include support for at least all of the following:

- Data definition – The DBMS must be able to accept data definitions (external schemas, the conceptual schema, the internal schema, and all associated mappings) in source form and convert them to the appropriate object form. This means that the

DBMS has to include a DDL processor or a DDL compiler components for each data definition language (DDL).

- Data manipulation – The DBMS must be able to handle requests to retrieve, update, or delete existing data in the database or to add new data to the database. This means that the DBMS must include a DML processor or DML compiler component to deal with the data manipulation language (DML). These kinds of requests are issued from prewritten application programs.
- DML requests – can be planned or unplanned. A planed request is one which was foreseen well in advance of the time at which the request is executed. An unplanned request, by contrast, is an ad-hoc query, a request for which the need was not seen in advance. These kinds of requests are issued interactively via a query language processor.
- Optimization and execution – Each DML request, regardless of the type, must be processed by the optimizer component whose purpose is to determine an efficient way of implementing the request. The optimized requests are then executed under the control of the run-time manager.
- Data security and integrity – The DBMS must monitor user requests and reject any attempts to violate the security and integrate constraints previously defined.
- Data recovery and concurrency – One component of the DBMS called the transaction manager or processing transaction monitor (TP monitor), must enforce certain recovery and concurrency control.
- Data dictionary – The DBMS must provide a data dictionary, which is regarded as system database rather than a user database, and contains data about data (metadata). Matadata represents the definitions of other objects in the system. All of the various schemas and mappings and all of the various security and integrity constraints will be stored, in both source and object form, in the dictionary. The dictionary itself should be integrated into the database it defines and thus includes its own definition, and it should be possible to be queried.
- Performance – All of the tasks mentioned above should be performed as efficiently as possible.

The three most important types of database systems are:

- Relational Databases (ORACLE, SYBASE, INGRES)
- Object Oriented Databases (Object Store, Objectivity, O2)
- Object/Relational Databases - In an effort to profit from the best elements of both worlds (relational and object oriented), several vendors (Universal Database version for DB2, Oracle 8I Universal Server, Database Server, or Enterprise Server, Universal Data Option for Informix Dynamic Server) have released "object/relational" DBMS products. This technology is still at the beginning and will not be described as part of this thesis.

4.2 Relational Databases

Relational databases have contributed considerably to the impact of database technology. In particular, these systems have proved to be an effective tool enabling data to be used by several users simultaneously, incorporating high level and easy to use computer languages. Furthermore these systems afford efficient facilities and a set of functions which ensure confidentiality, security and the integrity of the data they contain. For all these capabilities, relational databases are one of the basic elements of technology used in the development of advanced data systems.

Any relational system is based on a formal foundation, or theory, called the relational model of data [41,42], which means that:

- the data in the database is perceived by the users as a collection of relation variables called relvars, or more informally tables, and nothing but tables (Structural aspect)
- the tables satisfy certain integrity constraints, like primary and foreign keys (Integrity aspect)
- the operators available to the user for manipulating the tables, for purposes of data retrieval, are operators that derive tables from tables (Manipulative aspect). The most important operators are restrict, project, and join. The restrict operation, which is also known as select, extracts specified rows from a table. The project operation extracts

specified columns from a table. The join operation joins two tables together on the basis of common values in common column.

The relational model specifies that, the entire information content of the database is represented in only one way, as explicit values in column positions in rows in relations (The information principle) [42]. Every relation has a heading and a body, where the heading is a set of column-name:type name pairs, and the body is a set of rows that conform to the body. As a result of this fact in relational systems types and relations are necessary and sufficient to represent any data (at the logical level). The heading of the relation can be regarded as a predicate, and each row in the body denotes certain true propositions, obtained by substituting certain argument values of the appropriate type for the parameters of the predicate. This method of representation is the only method available in a relational system (there are no pointers connecting one table to another, only common columns repeated in one table and another). Another very important feature of the relational systems is the closure property. This means that the result of any of the operations presented above, is the same kind of object as the input object (they are all tables). Thus the output from one operation can become input to another one and it is possible to write nested expressions (e.g. take a projection of a join, or a join of two restrictions, or a restriction of a projection). Tables (relvars) can be updated by means of the relational assignment operation (e.g. the update operations INSERT, UPDATE, DELETE).

In a relational database the process of navigating around the stored data, in order to satisfy the user 's request is performed automatically by the system, not manually by the user (automatic navigation). Because of this, relational systems are often said to be non-procedural because the user specifies only what, not how (it doesn't specify a procedure for getting the information). The automatic navigation is the responsibility of yet another important system component, the optimizer, which determines how to implement user requests by choosing from a set of possible solutions based on the following considerations:

- which revlars are referenced in the request;
- how big the revlars are;

- what indexes exist;
- how selective those indexes are ;
- how the data is physically clustered on the disk;
- what relational operations are involved, etc.

Another important element of the relational model is the catalog, which contains detailed information, called descriptor information or metadata, regarding all of the various objects that are of interest to the system (e.g. relvars, indexes, users, integrity constraints, security constraints, etc.).

The original relvars in a given database are called base relvars, they have independent existence (are autonomous), and their values are called base relations. Relations which are obtained from base relations by means of some relational expressions are called derived relations. Another kind of relvars are also supported by relational systems called views, which do not have independent existence (are not autonomous), and they can be obtained by evaluating certain derived relations at a given time. Users can operate on views in the same way they operate on base relvars.

Another component of the relational model is a transaction, which is a logical unit of work involving several database operations. The most important features of a transaction are the fact that a transactions is:

- Atomic - they either execute in their entirety (commit) or do not execute at all (roll-back).
- Durable - if the transaction successfully commits, all updates executes within the limits of the transaction are guaranteed to be applied to the database, even if the system subsequently fails at any point.
- Isolated – transactions are isolated from each other, in the sense that database updates made by a transaction are not made visible to any other transaction until and unless the first transaction successfully commits. If the transaction rolls back, all updates made by the transaction are canceled.

Also, the interleaved execution of concurrent transactions is guaranteed to be serializable, which means that the same result is produced if the transactions are executed one at a time in an unspecified serial order.

The SQL is the standard language for dealing with relational databases, and it should be based on the current version of the standard SQL/92 [41].

4.3 Object Oriented Databases

Over the past decade much interest has been generated in object-oriented databases. They are considered as serious competitors to relational databases, at least for some kind of applications (Computer-aided design and manufacturing, computer integrated manufacturing, computer-aided software engineering, science and medicine). The main interest for object oriented databases was generated by the fact that some of the features needed in DBMSs have existed for years in object programming languages, thus the idea of incorporating those features into a database system [38]. Because the object paradigm has been very successful in meeting the goal of raising the level of abstraction in the programming languages arena, different researchers and vendors tried to apply the same paradigm to the database arena also. The idea of not having the users deal with machine oriented constructs such bits and bytes (or even fields and records), but rather with objects and operations on those objects that more closely resemble their counterparts in the real world, was obviously more attractive.

Unfortunately there is no common model, no formal foundation for the concepts, and no standard for object-oriented models. The data model corresponding to an object oriented database is ad-hoc to a certain degree, because it is not based on formal logic, and there is no common data model for all object-oriented databases. Besides this, the navigational model of computation for object oriented databases is a manual navigation. Because the data is presented to the users in the form of sets of structures of objects, users have to use the operators provided for manipulating these structures in order to navigate through the

database. In spite of these, a database requires a proper data model where certain generally accepted concepts concerning the model, can be grouped together into a core model or basic model. The core model offers a sufficiently powerful solution to satisfy the requirements of advanced applications and identify the main differences compared with conventional models. The core model does not capture integrity constraints, such as uniqueness of the values of an attribute or the range of values that an attribute can assume, or semantic relationships, such as the notion of "part of/between" pairs of objects and object associations.

The principal concepts of the core model are:

- Objects and identity,
- Complex objects,
- Encapsulation,
- Classes,
- Inheritance, and
- Overloading, overwriting, and late binding.

In object-oriented systems, each real world entity is represented by an object which is characterized by a state and behavior [38]. The state is represented by the values of the object's attributes, and the behavior is defined by the methods acting on the state of the object upon invocation of corresponding operations. Furthermore, each object has a unique identifier called its object ID or OID (even for two distinct objects that are duplicates of each other, are identical in all user visible respects). Immutable objects serve as their own OIDs, by contrast mutable objects have addresses as their OIDs, which can be used elsewhere in the database to refer to objects in question. Such addresses are not exposed to the user, but they can be assigned to program variables and to instance variables within the objects. Because objects can be referenced (via the OIDs) by any number of objects, they can effectively be shared by those other objects. This means that they can belong to any number of collections simultaneously.

There are two types of objects:

- Immutable objects correspond to values – e.g. integers, character strings.
- Mutable objects correspond to variables – e.g. an instance of a certain class.

Object identity introduces two different notions of equality between objects:

- The identity equality "=" – two objects are identical if they are the same object (they have the same OID)
- The value equality "==" – two objects are equal if the values of all their attributes are equal

Certain models support a third type of equality called shallow equality, where two objects are shallow-equal (they are not identical) if all their attributes share the same values and the same references.

The values of an attribute can be other objects, immutable or mutable (complex objects). If the values are immutable, when the object is loaded from disk they are immediately visible, but if the values are mutable further disk access is needed in order to retrieve them.

Usually the concepts of type and class are used interchangeably, but if they are used in the same language, the type is used to indicate the specification of the interface of a set of objects, while class is used to indicate the structure and implementation of a set of objects. Each class has a set of methods that can be applied to individual objects, and they are the only elements that can see the internal structure of the objects. Individual objects are called instances of a class. Objects are encapsulated, which means that the internal structure of an object is not visible to the users of the object. The advantage provided by encapsulation is that the internal representation of the objects can be changed without requiring applications that use those objects to be re-written. From a database point of view, encapsulated objects are described as having a private memory and a public interface:

- The private memory consists of instance variables (data members, or attributes), whose values represent the internal state of the object. Usually, instance variables are completely private and hidden from the users, but there are cases when they are not

hidden from the users (public), or they are a variation on private instance variables called protected. Protected means that the protected instance variable is visible to the code that implements the methods defined for the class, and any subclass of it.

- The public interface consists of interface definitions for the methods that apply to a specific object. The code that implements the methods, like the instance variables, is hidden from the users. All methods are invoked by means of messages, which in turn are just operator invocations.

  D HIRE_EMP (E) – which means that a message is sent to department D to hire employee E.

There are two different mechanisms for generating objects:

- By using the equivalent of a new operation
- By using prototype objects

In the first case, the values of the attributes for each object are stored separately, but the definitions of the operations and of the methods do not have to be repeated. The second approach involves generating a new object from and existing one, by modifying its attributes and/or behavior. The first approach is more appropriate where the application environments are more established, since it makes difficult to experiment on alternative structures of objects, whereas the second approach is more advantageous in environments that change more quickly and where are fewer established objects.

In almost all object oriented models, an attribute has associated with it a domain which specifies the classes of the possible objects that can be assigned as values to that attribute. For example the fact that an attribute of a class A has a class A' as a domain, implies that each instance of A assumes as the value of the attribute an instance of A', or of a subclass of it. Thus an aggregation relationship is established from class A to class A'. If A' is in turn defined in terms of other classes, the set of classes in the schema is then organized in an aggregation hierarchy.

Another important feature of the base model is the migration of instances between classes. This means that an object can become an instance of a class that is different from the class which it was generated (evolution of objects), and it means that an object can modify its own characteristics – attributes and operations – while maintaining the same identity.

One of the most important issues is the persistence of instances of classes, There are two basic approaches:

- Persistence is an implicit characteristic of all instances of classes, which means that the creation of an instance has the effect of inserting the instance in the database. This is typically used in systems where classes also have an extensional function.
- Persistence is an orthogonal characteristic, which means that the creation of an instance does not have the effect of inserting the instance in the database. In order to make an instance persistent the instance has to be inserted in a persistent collection of objects

An intermediate approach is to categorize classes into persistent classes and temporary classes, having all instances of persistent classes created as persistent instances, whereas this wouldn't happen with instances of temporary classes.

Persistent objects can be deleted in two ways:

- By providing an explicit delete operation. This solution raises the problem of reference integrity, if there are other objects referencing the deleted object those references will become invalid. A solution to this problem would be to keep a reference count (CORBA objects) and to delete the object only when the reference count is zero.
- By not keeping any additional information and allowing freely delete operations. The problem with this solution is that referencing references to deleted objects cause exceptions and thus it requires additional code in applications and methods in order to handle these exceptions.

In order to simplify the management of classes by the system, and to ensure the uniform application of the object-oriented themselves, the notion of metaclasses is needed. If each object is an instance of a class, and classes themselves are objects, then the system must support the concept of metaclasses in the sense of the class of a class. In turn a metaclass is an object and therefore must be the instance of a metaclass on a higher level. Generally metaclasses can be directly accessed and manipulated by the user.

Another important element of the core model is inheritance. With inheritance, a class called subclass can be defined on the basis of the definition of another class called superclass. The subclass inherits the attributes, methods and messages of its superclasss, but it can have its own specific attributes, methods and messages which are not inherited. In some systems a class can have several superclasses, in which case we have multiple inheritance, whereas other impose the restriction of a single superclass, in which case we have single inheritance.

The core model includes also support for overriding, overloading and late binding. Sometimes it is useful to send the same message to different objects belonging to different subclasses of a specific superclass, and depending on the type of the objects, the redefined operation for that specific subclass to be invoked (overriding). In order to provide this functionality the system must be able to bind the names of the operations at run-time (late binding). Sometimes it is useful to have methods having the same names but different signatures as part of a specific class (overloading).

Some semantic extensions to the core model, which are still at research level, and which are not available in the data models of the various object oriented database management systems are listed here:

- Composite objects – objects can be defined in terms of other objects.
- Associations – represented by means of references between objects.
- Integrity constraints – assertions which must be satisfied by objects in a database.

The object-oriented model is one of the most promising approaches in software development especially in the development of data-intensive applications, in spite of the lack of a common data model and a common standard.

4.4 Comparative Database Architectures

This chapter describes the two most dominant database architectures used in the current client/server world to writing database engines [41]:

- The Multi-Process Database Engines
- The Single-Process Multi-Threaded Database Engines

A complete comparison of the two architectures is also presented, emphasizing on the advantages and disadvantages in using one over the other.

4.4.1 Multi-Process Database Engines

This type of database architecture is characterized by multiple executables running simultaneously in order to perform user-query work. In this architecture, each time a user logs in, it actually starts a separate instance of the database engine itself. Thus each user login is running its own instance of the database application. In order to coordinate many users accessing the same sets of data, these executables work with other "global coordinator" tasks to schedule operations among these various users. Applications in a database of this type communicate using a proprietary *inter-process communication* (IPC) facility. Multi-process database engines typically are used on mainframe databases, and are popular on the MVS operating system, because MVS provides IPC facilities for applications.

The most popular example of a multi-process database engine, which is not run on mainframe, is the Oracle Corporation's Oracle Server, which is a true multi-process

database engine. Sixteen types of executables are loaded by the Oracle Server to perform different tasks. User connections start user database executables. System executables manage multi-user access to data tables, transaction logging and versioning, and other features, such as distributed transactions, data replication, etc. Each time a user connects to an Oracle database, it loads a distinct instance of the Oracle database executable. Queries are passed to that executable, which works in concert with the other executables on the server to return results sets, manage locking, and other necessary access functions. The following figure illustrates how multiple database engines require multiple instances of the database executable. In turn, this requires substantial system resources in order to provide memory resources and handle operating system resources.

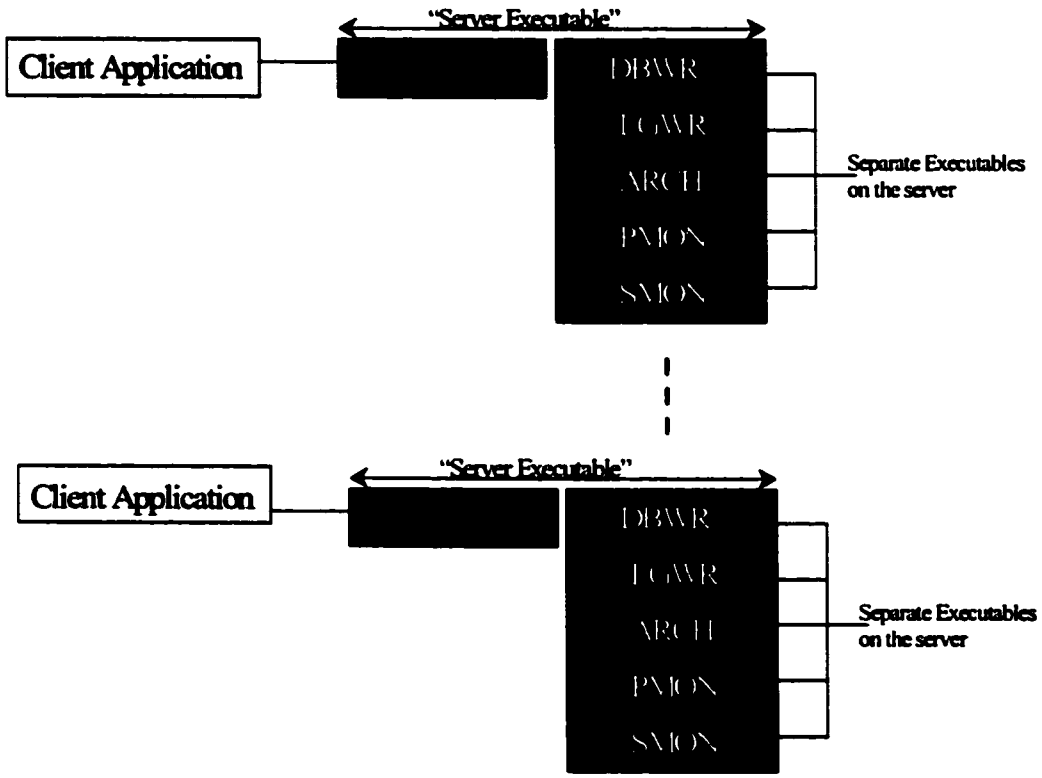

Figure 4 - 3 Multi-process database engines.

4.4.1.1 Pros and Cons of Multi-Process Database Engines

Because most of the multi-process engines were developed before operating systems supported features as threads and preemptive scheduling, "breaking down" a single operation meant writing a distinct executable to handle that operation, which in turn enabled two important benefits for database processing:

- A database could support multiple simultaneous users, providing for data centralization on a network.
- It provided scalability through the addition of more CPUs and the physical machine.

In a multitasking operating system, the OS divides processing time amongst multiple applications (tasks) by giving each task a "slice" of the CPU's available work time. In this way there is still only one task at one time. However, the tasks share the processor because the OS grants a particular percentage of the CPU's time to each task. As a result multiple applications appear to running simultaneously on a single CPU. The real advantage comes when multiple CPUs can be used by the operating system. The capability to schedule distinct tasks to distinct processors gives applications the true capability to run simultaneously. The disadvantage is that this architecture consumes higher system resources than the multi-threaded architecture, but appear to scale to dramatically larger platforms more easily than their counterparts.

4.4.2 Single-Process Multi-Threaded Database Engines

Instead of relying on a multi-tasking operating system to schedule applications on a CPU, a multi threaded database engine takes on this responsibility for itself. This capability gives the database engine a greater portability, because the database manages scheduling individual task execution, memory, and disk access. As a result of this, multi-threaded systems are more efficient for a given hardware platform by creating threads to process the incoming client requests. In addition because the database itself manages these multiple threads, coordinates the multiple operations it must perform, and sends

instructions to the operating system for final execution, there is no need for a costly and inefficient inter-process communication mechanism. In this way, the database time-slices individual operations by taking individual threads, one at a time, and sending the user instructions on those threads to the operating system. By having the DBMS time-slicing threads, instead of having the OS time-slicing applications, the database uses a finite element of work for a variety of operations (user instructions, locking data pages, disk I/O, cache I/O, etc.), instead of having multi-process DBMS, which uses specialized applications for each. This kind of architecture is used for the implementation of all Sybase SQL Server versions.

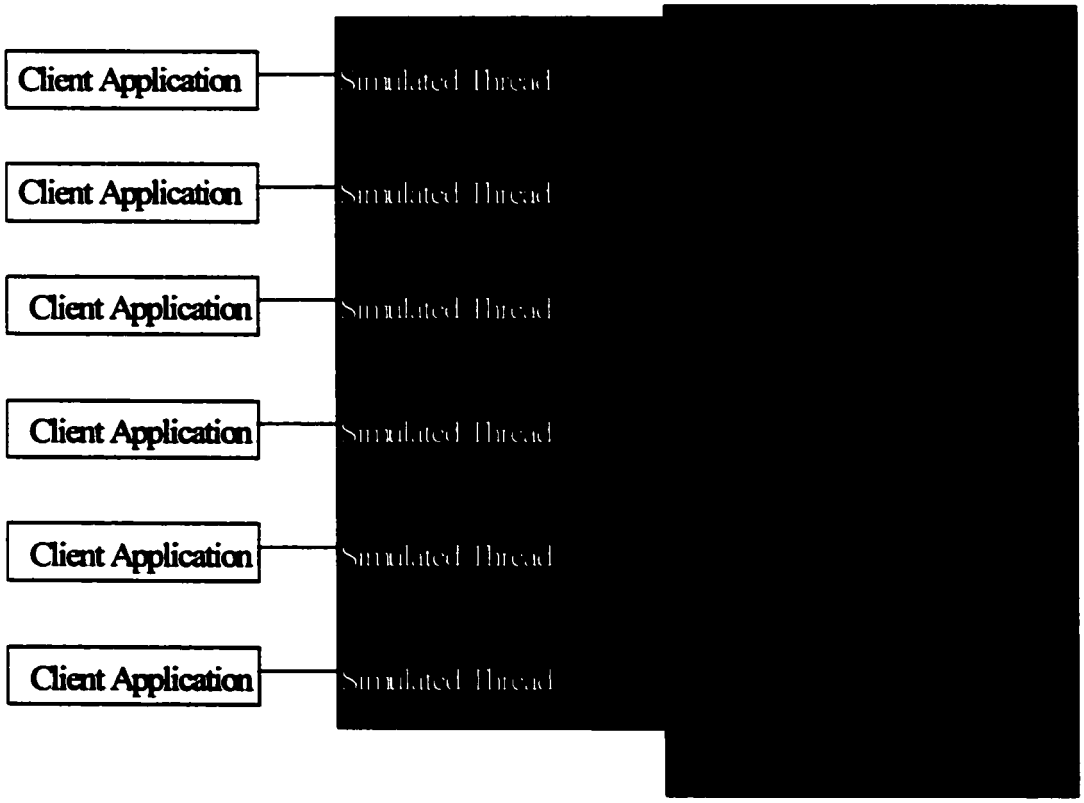

Figure 4 - 4 Single-process Multi-threaded database engines.

4.4.2.1 Pros and Cons of Multi-Threaded Database Engines

Database Management Systems using the multi-threading approach, are ones of the fastest DBMS systems for any given database size or hardware platform. The performance is derived from the efficiency of a multi-threaded DBMS, which by running multiple internal threads consumes more effectively far fewer system resources, relative to other multi-process databases. One of the most important benefit resulted from this efficiency, is the reduction in RAM requirements for the server, saving memory for data and procedure caching. The cache is a critical element for any database application, and maximizing both the quantity and usage of that cache is an important performance factor. Single-process multi-threaded DBMSs are not inherently restricted from using multi-processor hardware (SMP), but the SMP support competes with portability, because different hardware vendors implement SMP in different ways. The disadvantage of using this type of architecture, even for implementations using SMP, is: because threads are the most finite element of work, only a single thread can be scheduled to one CPU at any given time, therefore there is no parallelization of work for a specific user; there is a single task, running on a specific CPU at any given time. Any performance benefits resulting from SMP are the result of another processor being made available before the existing one would be, which is a scheduling benefit not a true processing benefit.

5 Object Store

After taking into consideration the advantages and disadvantages, presented by the two different types of DBMSs previously described in this thesis, and the fact that any implementation of the CORBA standard uses an object approach, the Object Store database management system is the DBMS chosen to implement the concepts proposed in this thesis. Many of the concepts that support the implementation of a Generic Database Adapter for multi-tier CORBA based applications are based on the object oriented database model developed by Object Design. The aspects of Object Store that are applied in subsequent chapters are presented in details while other features are presented for completeness [28,29,30,31]. The technical aspects of the Object Store DBMS that are relevant to implementing a generic database adapter are examined and presented. Object Store has different solutions for different programming languages, but as part of this thesis only the implementation for C++ is discussed [28,31].

5.1 Overview

Object Store is a high-performance, distributed, heterogeneous object-oriented database system, which is characterized by the following features:

- objects are stored in the same format as transient C++ objects,
- access to persistent objects via normal C++ programming constructs,
- multiple Object Store servers can run in a distributed architecture,
- multiple client applications may access multiple servers/databases,
- any client can share a database with any other client, plus

DBMS features:

- transaction semantics and concurrency control,
- distributed, heterogeneous access to persistent data,
- backup, restore, and recovery features,
- schema evolution,
- query facility, and

- inter-process notification.

Object Store is implemented as a multi-threaded server which provides access to different clients, by using light weight threads. Object Store provides full support for C++ applications including:

- transparent support for pointers
- seamless support for inheritance
- full support for templates
- exception mechanism
- class library (collections, queries, and idexes)

Storing persistent data with Object Store, is a lot like storing transient data with C++; memory must be allocated, and a value must be assigned to that memory. The only difference is that when persistent data is stored, persistent memory must be allocated. Thus any type of data can be stored in either the persistent memory (Object Store database) or transient memory (stack or heap). The Object Store clients view Object Store as network-wide virtual memory space that spans multiple servers. This makes it a lot easier for the application developers to manage data in an application, and it also results in faster performance by using the hardware and the operating system virtual memory process designed for high speed data access.

5.2 Object Store Distributed Architecture

The component elements of the Object Store architecture are [31]:

- the Object Store clients
- the Object Store Server
- the Object Store database
- the Cache Manager
- the Communication Segments
- the Client Caches

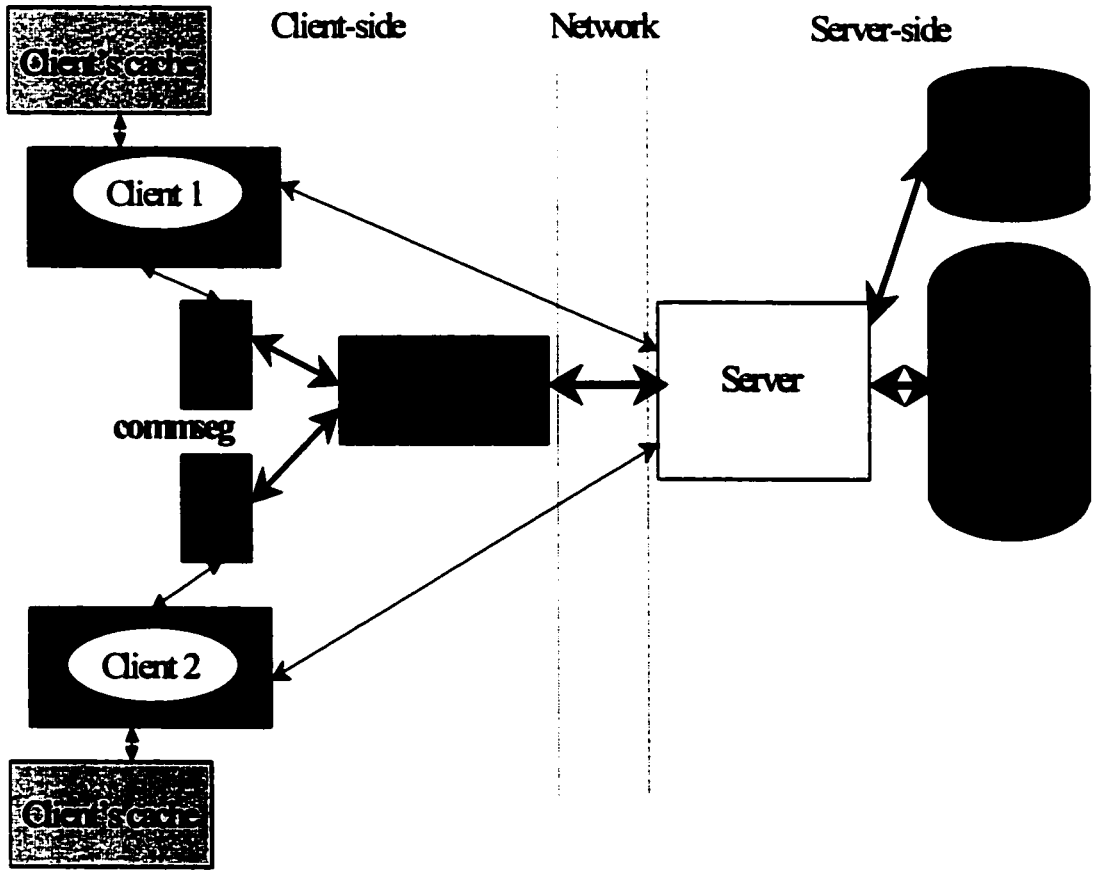

Figure 5 - 1 Client-Server Interaction Architecture.

There is at most one Object Store server running on each host having the following responsibilities:

- provides I/O services to the databases

- manages multi-client concurrency control
- provides on-line backup services
- ensures transaction-consistent databases
- participates with other servers to provide two-phase commit

Each server process has one file which acts as a Transaction Log. The Transaction Log provides the following services:

- recovery mechanism – because the log is written before commit completes
- faster transaction commits – because most updates are written to the log, taking into consideration that direct writes to the database is less efficient, which are periodically propagated to the database by the server.

There can be multiple clients running on each host having the following responsibilities:

- execute transactions
- allocate objects
- manipulate objects
- de-allocate objects
- maintain a cache of recently accessed database pages (done automatically)
- map persistent objects into address space (done automatically)

There is one Cache Manager per each host running Object Store clients. The role of the Cache Manager is to act as a lock manager for all the Object Store clients on a host machine. Before a client can access a specific persistent object, it has to acquire a specific permit (read-only or write) and a specific lock (read-only or write) on the database page containing the object from the Object Store server. When another client requests the same page, the Cache Manager, at the request of the server checks if there is an existing client having a lock on the page containing the object requested. If the lock on the page is not needed anymore, the Cache Manager retrieves the lock and the server assigns the lock on that page to the next client.

There is one client cache for each Object Store client used as a swap space for in-memory persistent objects and there is one Commseg (communications segment) for each Object Store client used to track the status of cached pages as follows:

- used by the client to determine whether a page can be accessed and locked, and
- used by the Cache Manager to check and/or to give-up lock upon server's request.

Object Store databases can be accessed only via the Object Store server and they can be stored in two different ways:

- file database – native system files configured as Object Store databases, or
- stored in raw partition configured for Object Store's raw file system

The first alternative is recommended during the development process because of the easiness to use and configure, and the second solution is recommended for application deployment because the databases are invisible to non-Object Store application and because a database can span multiple partitions, thus more easily supports very large applications.

5.3 Object Store Databases

In an Object Oriented database data is organized logically as a network of objects (objects pointing to other objects) [31,32]. Each database has one or multiple persistent objects as starting points for navigating into the database, called entry points. The access to the entry point objects, and from here to any persistent object in a database, is done via a database root, which can be looked up by name.

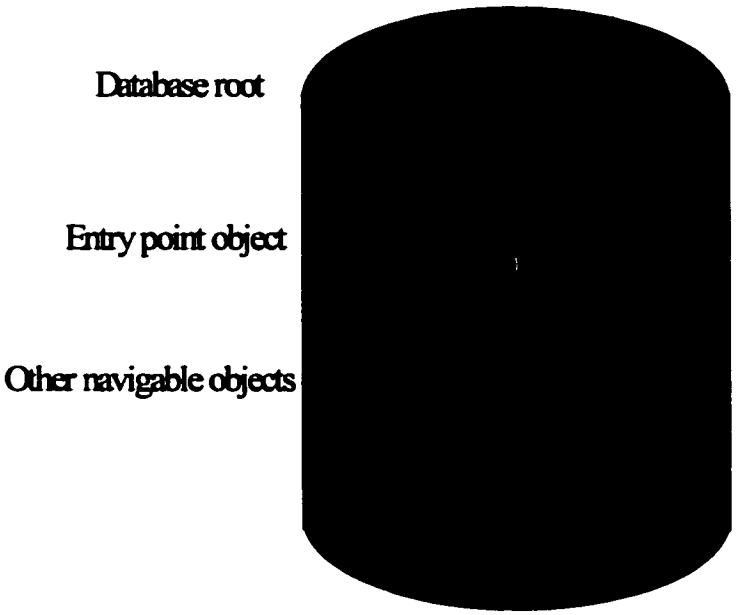

Figure 5 - 2 Database logical organization.

Databases can be distributed, thus multiple clients, can access multiple databases, managed by multiple servers. Each database contains segments (a variable sized unit of database organization), and they contain in turn database pages (a 4k or 8k unit of data). By default each database is created with two segments:

- A schema segment – it contains database schema objects and database roots, and
- A default data segment – contains all persistent objects, unless objects are specifically allocated on other segments.

Databases may contain multiple segments (the standard unit of allocation for clustering data), each segment may contain multiple object clusters (a fixed-sized unit of data), and both can be added at run time. Objects may be allocated in a specific segment or object

cluster. The Object Store API for C++ provides methods to create, open, close, and destroy a database, methods to create and destroy segments and clusters, and methods to create, search and delete database roots.

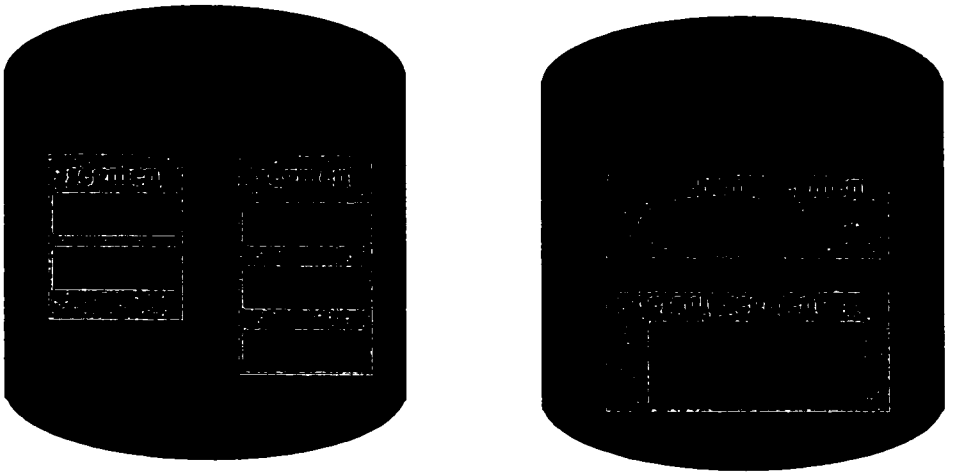

Figure 5 - 3 Physical Database Organization.

5.4 The Schema

Each Object Store database has associated schema information, which is a group of data definition objects, which describe the structure of data in a database [28,31]. Without schema, the layout of the objects that are stored or are to be stored in the database would not be known. The database schema is generated as part of the build process, based on the information on the classes of objects the database is to contain, and is typically stored in the schema segment of the Object Store databases. At the same time each application has an application schema, which is an Object Store database containing definitions for all user-defined types in the application, and is generated from header files using a schema generator utility [31]. The role played by the application schema is twofold:

- It is used to generated the schema segment when a new database is created, and
- It is used to validate the schema in an opened database to ensure that the description of the types (as described in their header files) matches the descriptions kept in the database's schema segment.

5.5 Object Store transactions

A transaction is a logical unit of work [31], a consistent and reliable portion of the execution of the program. In Object Store, database persistent objects can only be accessed within the limits of a transaction, but if persistent data is accessed from outside of a transaction, an exception is raised. While the transaction is in progress read and write operations can be executed on persistent objects. The transaction can either be committed or aborted at any time by the Object Store client. By committing the transaction, all changes made to persistent data during the transaction are made permanent in the database and visible to other processes. These changes are made permanent and visible only when the transaction commits. Changes to persistent data are undone or rolled-back if the transaction in which they were made is aborted.

Transactions are central in keeping shared data consistent in database applications. A transaction has four properties known as ACID properties:

- Atomic – all work on persistent data either succeeds completely or fails completely.
- Correct – the developer is responsible for defining consistent transaction boundaries.
- Isolated – no changes to persistent data are visible until the transaction commits.
- Durable – the server ensures that all committed data is safe on disk and recoverable

Transactions in a database system serve two general purposes:

- They support fault tolerance
- They support concurrent database access

In support to fault tolerance, transactions have the following properties:

- Either all the changes to persistent memory corresponding to a transaction are made successfully, or none are made at all. If a failure occurs in the middle of a transaction, none of its database updates are made
- A transaction is not considered to have completed successfully until all its changes are recorded safely on stable storage. Once a transaction commits, failures such as server crashes or network failures cannot erase the transaction's changes.

Transactions support concurrent database access by preventing one process's updates from interfering with another process's reads or updates. Object Store's concurrency control facilities prevent this interference by ensuring that transactions have the following properties:

- A transaction's changes to persistent data are private and invisible to other processes until the entire transaction completes successfully.
- Other processes' changes to persistent data are invisible to a transaction.

There are two different ways to mark off transactions in Object Store:

- Lexical transactions, which require that the transaction has to be contained in a lexical context. The developer marks off lexical transactions with some Object Store transaction macros (OS_BEGIN_TXN, and OS_END_TXN).
- Dynamic transactions provide dynamically defined boundaries, which are marked off by the developer using Object Store APIs.

Furthermore, for multi-threaded applications, thread-locking facilities are provided by Object Store, in order to prevent threads from interfering with one another. The thread-locking mechanism works by either serializing the transactions of different threads or serializing access by different threads to the Object Store run time. No two threads are ever in the Object Store run time at the same time. While Object Store supports multithreaded clients, it currently supports only one active independent transaction per process, which results into having one thread in a transaction at a time, or several threads sharing the same transaction. For multi-threaded applications, there are two types of transactions:

- Local transactions
- Global transactions

When one thread enters a local transaction, this has no effect on whether other threads are within a transaction [28]. If one thread enters a global transaction, all the other threads starting a transaction, enter the same global transaction.

Local transactions synchronize access to the Object Store run time by serializing the transactions belonging to different threads, that is by making the transactions run one after another without overlapping. After one thread starts a local transaction., if another thread attempts to start a transaction or enter the Object Store run time, it is blocked by a mutex lock until the local transaction completes. As a result two threads cannot be in a local transaction at the same time.

Global transactions allow for a somewhat higher degree of concurrency. After one thread enters the Object Store run time, if another thread attempts to enter the Object Store run time, it is blocked until control in the first thread exists from the run time. Although two threads cannot be in the Object Store run time at the same time, there can be some interleaving of operations of different threads within a transaction.

Local transactions usually provide better performance, but for some applications, global transactions might be preferable. Some of the benefits of using global transactions are:

- Provide for a greater degree of concurrency
- Any attempt to access persistent memory from outside a transaction results in an exception.

Some of the disadvantages are:

- Global transactions have extra overhead, compared to local transactions, in the form of extra memory management, particularly if there is a lot of cache replacement.
- With global transactions, the application must synchronize the threads so that no thread attempts to access persistent data while another thread is committing or aborting.

5.6 The Object Store Locking Mechanism

As most DBMSs, Object Store tries to interleave the operations of different processes' transactions in order to maximize concurrent access of resources. When scheduling operations, Object Store conforms to the two-phase locking discipline, except in the case of multiversion concurrency control (MVCC) [28,31]. This discipline has been proven correct in the sense that it guarantees serializability, which means that it guarantees that the result of the schedule will be just the same as the result of some non-interleaved schedule of the transactions' operations.

When a certain application accesses data in the database, it is given exclusive access to that data for the entire duration of the transaction, by locking that data. The lock is on the page containing the object accessed (but can be at segment level, or database level), and the server maintains how each client uses pages in order to manage concurrency. Because no other process can access it, this can affect performance of other concurrent processes. In order to increase concurrency, the Object Store locking mechanism treats reading operations differently than writing operations. If data from a page is read, that page is read-locked by the application, thus other processes can read-access that page, but it prevents them from writing to it. If an application is writing to a page, that page is write-locked by the application, which prevents other processes from reading or writing to that page. This type of concurrency model is called "multiple reads or one exclusive write", and is the default model for Object Store. The entire locking mechanism is managed automatically by Object Store, but explicit locks can be acquired on a range of pages.

Another concurrency model implemented by Object Store is the Multi-Version Concurrency Control (MVCC). Because this model allows multiple readers and one writer to access the same page of data at the same time, it provides users with a mechanism to achieve a higher level of concurrency on database access.

If multiple clients try to access data in conflicting ways, the following types of lock conflict can occur:

- Lock wait – when two or more clients try to access the same object in conflicting ways
- Spurious lock wait – when two or more clients try to access different objects on the same page in conflicting ways
- Deadlock – when two or more clients cause a wait cycle where none can proceed

In a lock wait situation, a client must wait until the conflicting locks are released before proceeding. If a dead lock situation is detected by Object Store, the following solutions are applied:

- If the deadlock is detected in a lexical transaction, one client's transaction is aborted and retried automatically.
- If the deadlock is detected in a dynamic transaction, one's client transaction is aborted and an exception is raised.

In order to increase concurrency and performance, the applications have to avoid locking data unnecessarily and locking data for unnecessarily long periods of time. This strategy can be implemented by keeping the locking granularity at page level, by avoiding segment level or database level locks, by keeping the length of the transactions not too long (locks are not being kept for long periods of time) but also not very short (because of the overhead generated by committing transactions is reduced).

5.7 Object Store Exceptions

When running Object Store applications different types of errors and conditions can be generated as a result of:

- A failure to open a database,
- A failure to access persistent storage from within a transaction,
- Network failures,
- Deadlock notifications, etc.

Object store reports these errors and conditions via a hierarchy of exceptions. The exception handling mechanism is implemented using special exceptions macros which must be used together in the same block of code:

- TIX_HANDLE(exception_name) – establishes a handler for and exception named exception_name
- TIX_EXCEPTION – establishes beginning of exception handling code
- TIX_END_HANDLE – establishes end of exception handler.

The code contained between the first two macros is called protected code, and the code contained between the second and third macros is called handling code. When an exception is signaled in the protected code, the control jumps to the handling code. C++ exceptions can be used along with Object Store exceptions with no restrictions. Besides the standard Object Store exceptions applications can extend the existing exception mechanism by creating new exceptions.

5.8 Pointers and Object Store References

The Object Store mechanism to access persistent objects is based on the Virtual Memory Mapping Architecture (VMMA) [33]. Persistent objects are page faulted into the client's address space, automatically by the Object Store mechanism, by simply de-referencing a pointer. VMMA enables Object Store client applications to access persistent and transient objects the same way, therefore within a transaction, a mapped persistent object is accessed as efficiently as a transient object. Because mapping persistent objects to particular addresses is fixed for each transaction, the value of a transient pointer to a persistent object becomes invalid at the end of the transaction.

Each process has a unique virtual address space resource available, which on a 32 bit machine is 4GB. Each Object Store client reserves a range of addresses for mapping persistent objects called persistent storage region (PSR) [31,33]. The size of the PSR is

fixed for a given process, but is configurable to different sizes depending on the platform (from 128MB to 2 GB).

The PSR address range is used to assign addresses to all objects which reside in an Object Store database. VMMA mimics the operating system's virtual memory by swapping objects from RAM into a backing store on disk. In the case of persistent objects, the backing store is the client cache. The Object Store client cache is distinct from the swap used for transient objects, Object Store objects are not being swapped via normal virtual memory.

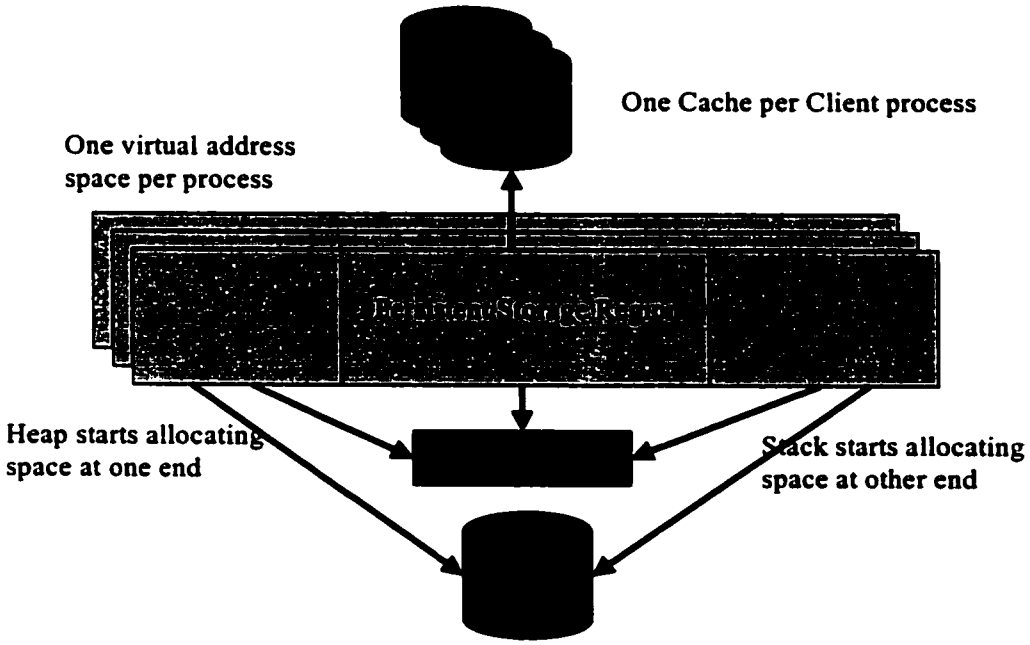

Figure 5 - 4 Address space and the PSR.

The PSR size limits the amount of address space which may be used in a single transaction, because when the first object located on a segment is referenced, address space for the entire segment is reserved and held until the transaction ends. The second motif is, if an accessed segment contains pointers to objects in other segments, address space is also reserved for those segments. The result of this can be that the size of the summed segments (the reach of the segment) can be greater than the PSR, which would

result in an immediate crash of the application. Because of this reason it is not recommended the use of cross-segments pointers. Also as specified, a pointer to a persistent object used in one transaction is valid only for the length of that transaction, and it becomes invalid after this. In order to avoid these two problems Object Store created Object Store references, which represent an object's persistent address, which can be used as cross-transaction pointer and cross-segment pointer. They can be dumped to a string and loaded from a string. The Object Store reference is composed of 12 bytes:

- The first 4 bytes represent the database identifier
- The next 4 bytes represent the segment identifier
- The last 4 bytes represent the offset within the segment

5.9 Object Store Collections

A collection is an object that serves to group together other objects. It provides a convenient means of storing and manipulating groups of objects, supporting operations for inserting, removing, and retrieving elements [29,31]. Collections also support set-theory operations such as intersection and set-theory comparisons such as subset Besides these, collections form the basis of the Object Store query facility which allows you to select those elements of a collection that satisfy a specified condition. The Object Store collection facility provides control over the behavior and representation of the collections (create ordered or unordered collections, collections that either accept duplicates or do not, etc.). The Object Store collections provide some alternatives to linked lists, C++ arrays, and other aggregation data structures, and they are easier to use, more powerful, and they can be used in transient memory and persistent memory. The object Store collections contain pointers to objects, not the objects themselves.

Object Store has the following types of collections:

- os_Set, or os_set – each set is composed of elements in the same way as linked lists and arrays. In contrast to linked lists and arrays, the elements of a set are unordered, but they do not allow multiple occurrences of the same element.

- os_Bag or os_bag – besides the fact that they keep track of what their elements are, but also of the number of occurrences of each element.
- os_List or os_list – are collections that associate a numerical position with each element based on insertion order. Lists can either allow or disallow duplicates. Besides simple inserts and remove, elements can be inserted and removed based on a specified numerical position or based on a specified cursor position.
- os_Collection or os_collection – provides all the features of sets, bags, and lists, allowing duplicate elements and maintaining element order.
- os_Dictionary – Like bags, dictionaries are unordered collections that allow duplicates. Unlike bags, dictionaries associate a key with each element. The key can be any C++ fundamental type or pointer type. For inserting an element, the key together with the element must be specified.
- os_Array or os_array – They are like Object Store lists, excepting that they always provide access to collection elements in constant time. Arrays also allow null elements, and provide the ability to automatically establish a specified number of new null elements.

The os_Set, os_Bag. os_List, os_Array, os_Collection, os_Dictionary, are templated collections (os_List<E*>), and os_set, os_bag, os_list, os_array, os_collection are non templated collections (contain void*).

Cursors are provided by Object Store for iterating over the contents of a collection. The cursor attaches to a collection object and it can be used to traverse the collection by visiting each element.

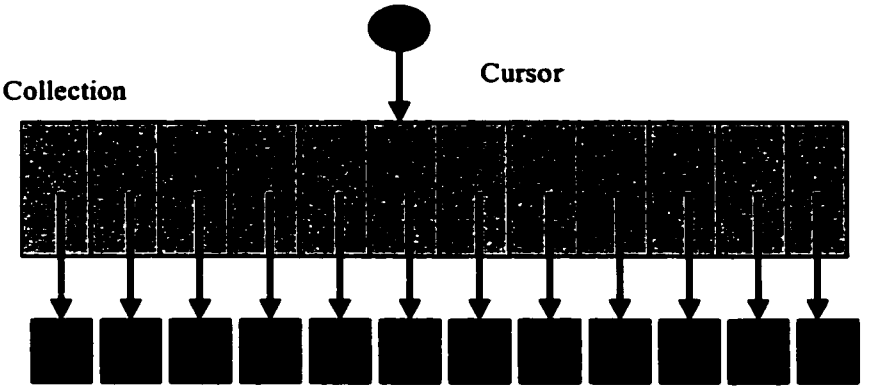

Figure 5 - 5 Collections and Cursors.

By default cursors are unsafe, which means that the cursor might:

- try to visit removed elements
- not visit inserted elements
- miss existing elements
- become confused over state of iteration

By default the elements of a collection are visited using the default iteration, which means that elements are traversed in the order in which they appear in the collection representation:

- if the collection is ordered, this is iteration order
- if the collection is unordered, this is arbitrary order.

The behavior of the iteration can be changed to allow the following options:

- update-safe iteration
- update-insensitive iteration
- restricted cursors

Update safe iteration, guarantees that any element that has been removed and not yet visited will not be visited, and all the elements inserted during iteration will be visited exactly once. For update-insensitive iteration, the cursor creates a snapshot with all collection elements, updates to the original collection being ignored. Restricted cursors allow the program to limit the number of elements visited during iteration, by visiting only elements that meet a certain criteria based on a data member value of element type.

5.10 Object Store Queries

The C++ language makes possible the sort of navigational data access required by typical design applications [29,31]. But, while navigation provides the most efficient form of access in many circumstances, other situations require associative access. Lookup of an object by name or ID number, is a simple form of associative access. Both associative and navigational retrieval are indispensable to databases supporting complex, data intensive design applications. Therefore, among the database services provided by Object Store is support for query processing. A query facility with adequate performance must go beyond support for linear searches. Thus, Object Store provides a query optimizer, which formulates efficient retrieval strategies, minimizing the number of objects examined in response to a query. The query facilities are used from within C++ programs, and they treat persistent and non-persistent data in an entirely uniform manner.

There are three ways to access data in an Object Store database:

- entry point access - by doing a root lookup
- navigational access – by de-referencing a pointer or Object Store reference
- associative access – by finding objects that meet certain criteria, using the Object Store query facility

Instead of iterating across a collection, having the code explicitly applying the test criteria to each element, the query facility selects only those elements of the collection that meet a certain criteria, by having the query expression passed to the collection and having a result returned. The Object Store query facility has its own language based on C++, and it allows the following:

- Multiple-element queries - select all elements of a collection that meet certain criteria, or
- Single-element queries - select one element of a collection which meets a certain criteria, or
- Existential queries - ask whether any element in the collection exists, which meets a certain criteria.

A program can execute a query in two ways:

- directly by invoking a query method on a collection with a query expression argument (criteria), or
- by creating a bound query object and passing it to the query method of the collection.

The first type of query is used when the query criteria is based on constant values (e.g. "this->balance > 10 "), and the second type is used when the query criteria is based on free variables that can change at run-time (e.g. "this->balance > (long)minimum_balance"). The developer has to bind a free variable to a program variable during the development of the application.

In order to perform a specific query, a linear search must be used on a specific collection. In order to improve the performance of the query facility, Object Store optimized the query mechanism by introducing indexes. An index can be added into a collection or dropped from a collection, by simply using the collection API. Thus an index is a keyed data structure owned by a collection, whose keys are based on the values of elements in that collection, and whose elements are pointers to collection elements at that key. A certain collection can have multiple indexes, each based on different keys. There are two types of indexes:

- ordered indexes, which are implemented as B-tree structures, and
- unordered indexes, which are implemented as hash tables.

Unordered indexes optimize exact match queries, which are queries involving "==" or "strcmp" operators, which are faster than queries optimized by using ordered indexes.

Ordered indexes optimize range queries, which are queries involving ">", "<", "<=", or ">=" operators. But if an application needs to optimize queries using a particular value for both range and exact-match queries, an ordered index should be used.

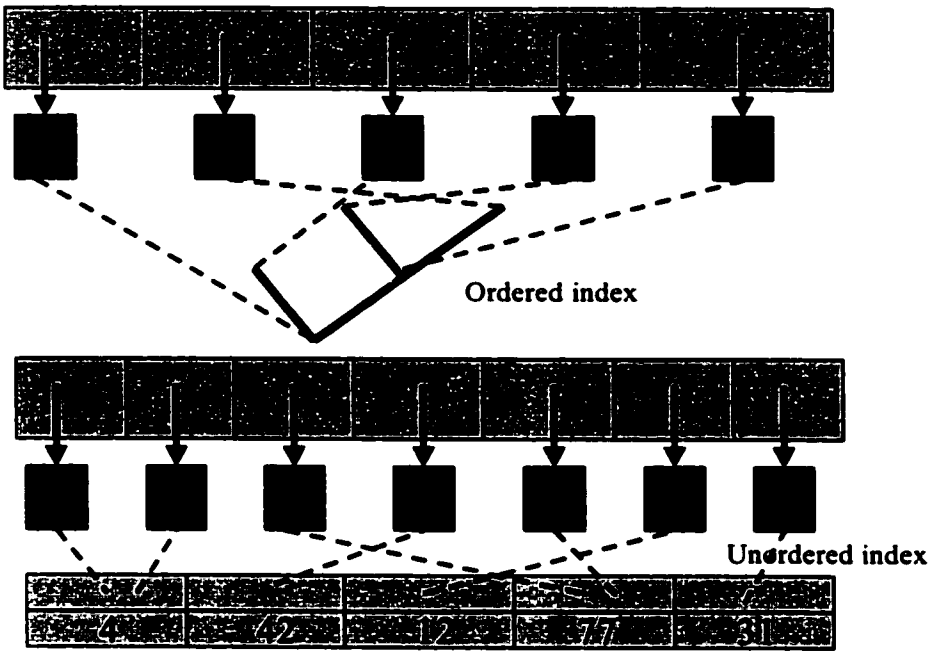

Figure 5 - 6 Ordered and Unordered indexes.

5.11 The Object Store Metaobject Protocol

The Object Store metaobject protocol (MOP) is a library of classes that allow access to Object Store schema information [28,29,33]. Schema information for Object Store databases and applications is stored in the form of objects that represent C++ types. These objects are actually instances of Object Store metatypes, so called because they are types whose instances represent types. Every object representing a type is an instance of a subtype of the metatype named os_type. For example, objects representing classes are instances of os_class_type, which is derived from os_type. Schema information is also represented with the help of various auxiliary classes that are not in the metatype

hierarchy, such as os_member, whose instances represent data members or member functions. Besides performing read access to Object Store application, compilation, and database schemas, the MOP allows to create classes dynamically and add them to the Object Store Database schemas.

Only the read access mechanism to the Object Store database schema is used in order to accomplish the goal of this thesis. Read access schema information always starts in one of two ways:

- By looking up a schema object by name in a database schema,
- By retrieving the class of which a specified object is an instance.

In either case a pointer to a constant object is returned. Other schema objects are the result of performing get functions on these initial constant objects (and the result of performing get functions on these results, and so on). For class-valued attributes, these get functions, in turn, return pointers or references to constant objects.

5.11.1 Retrieving the Object Representing the Type of a Given Object

For a given persistent object, the following operations are possible:

- Get an instance of os_type representing the type of that object can be retrieved, by passing the object's address to the static member function os_type::type_at().
- Get an instance of os_type representing the type of the outermost object containing the given persistent object can be retrieved, by passing the object's address to the static member function os_type::type_containing().

5.11.2 Retrieving Objects Representing Classes in a schema

Object store schemas are represented by instances of classes derived from os_schema.

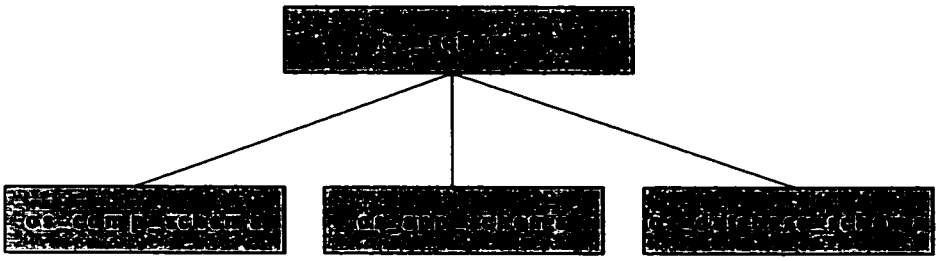

Figure 5 - 7 Schema hierarchy diagram.

The following operations can be performed on a database schema:

- Retrieve the schema object corresponding to each type of schema by using the corresponding static get function for each of them.
- Retrieve a collection of os_class_type objects representing all the classes in a given schema.
- Retrieve the class with a given name in a given schema by using the static function find_type.

5.11.3 The Metatype Hierarchy

All the types in the C++ type system can be divided into the following categories: class types, integer types, real types, enumeration types, array types, pointer types, function types, and the type void. For each of theses categories, there is a subclass of os_type in the metatype hierarchy.

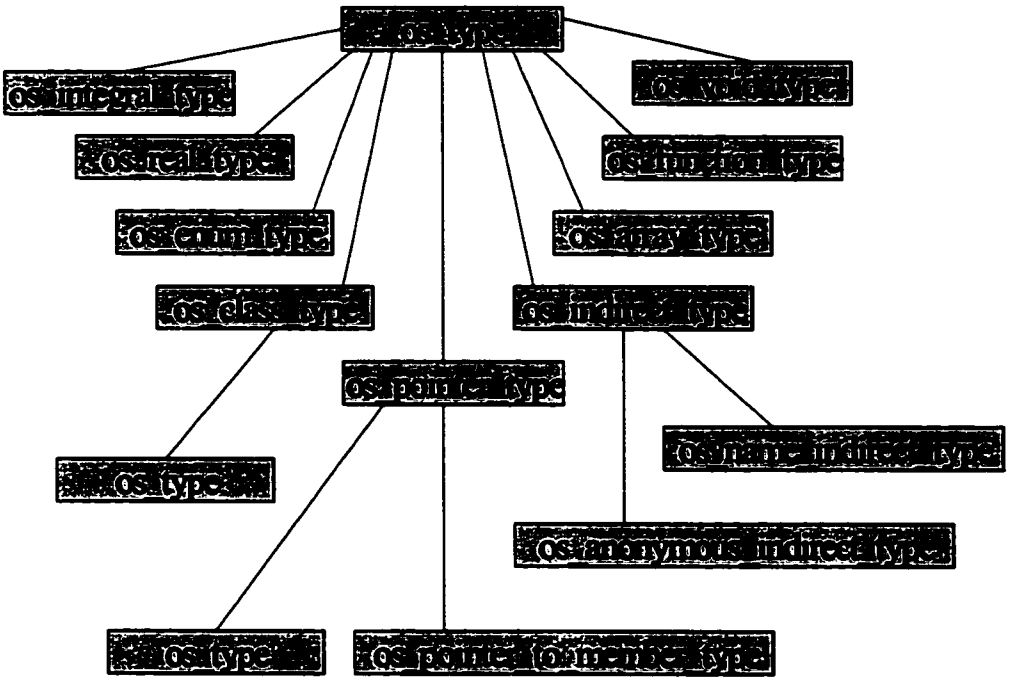

Figure 5 - 8 The Metatype Hierarchy.

The class os_instantiated_class_type, derived from os_class_type, represents the category of template class instantiations (e.g. os_Collection<Account*>). Pointer types, such as void* and part*, are represented by direct instances of os_pointer_type. Reference types, such as Account&, are represented by instances of os_reference_type, derived from os_pointer_type. Pointer to member types are represented by instances of os_pointer_to_member_type, also derived from os_pointer_type. Types with const or volatile specifiers are represented by instances of os_anonymous_indirect_type, and typedefs are represented by instances of os_named_indirect_type.

There are other classes in the metaobject protocol whose instances represent schema objects other than types, such as os_base_class and os_member and its subtypes.

5.11.4 Attributes of MOP classes

It is useful to think of the classes in the metaobject protocol as having attributes which represent pieces of abstract information that can be read using get functions, can be updated using set functions and can be created using create functions. Most of the functions in the MOP are create, get, or set functions, members of the class whose state they access.

5.11.4.1 The Class os_type

The os_type class is the superclass for all the classes in the metatype hierarchy. This class has no create functions, but all of the subtypes can create an instance of os_type.

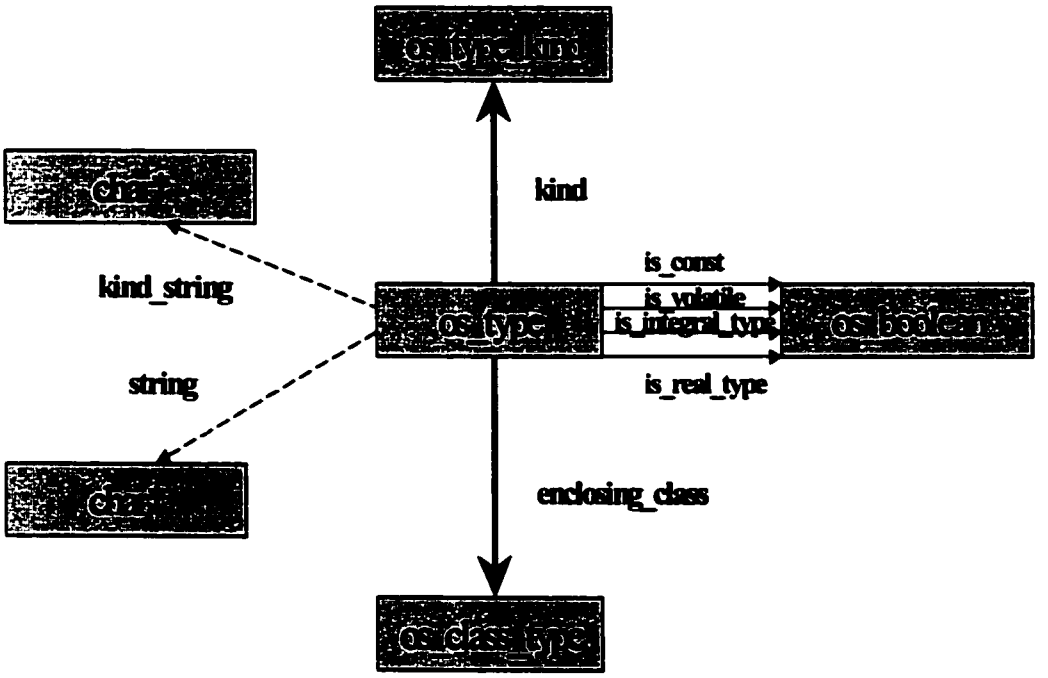

Figure 5 - 9 Attributes of os_type.

The class is being used in this thesis because has type-safe conversion operators to any of types of the subclasses previously presented.

5.11.4.2 The Class os_class_type

An instance of this class represents a C++ class. In addition to classes declared with the keyword class, structs and unions are also represented by instances of os_class_type. The list of all the members of a class can be retrieved using the corresponding os_class_type.

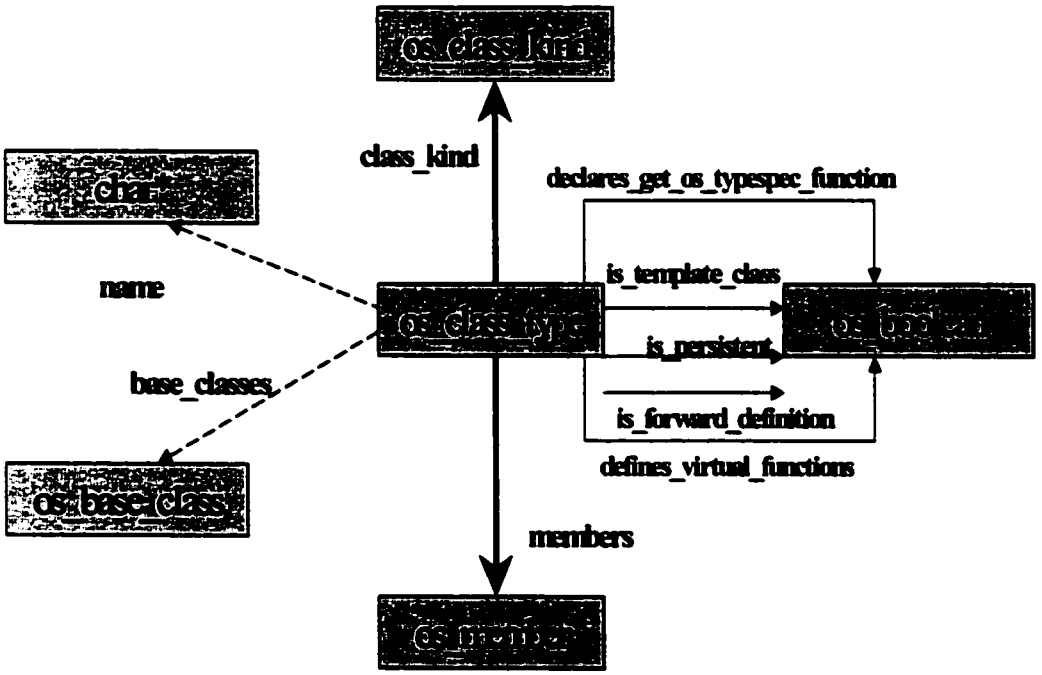

Figure 5 - 10 The abstract state associated with class os_class_type.

5.11.4.3 The Class os_member

An instance of os_member represents a data member or member function. This class has no direct instance. Every instance of this class is a direct instance of one of the subclasses of os_member.

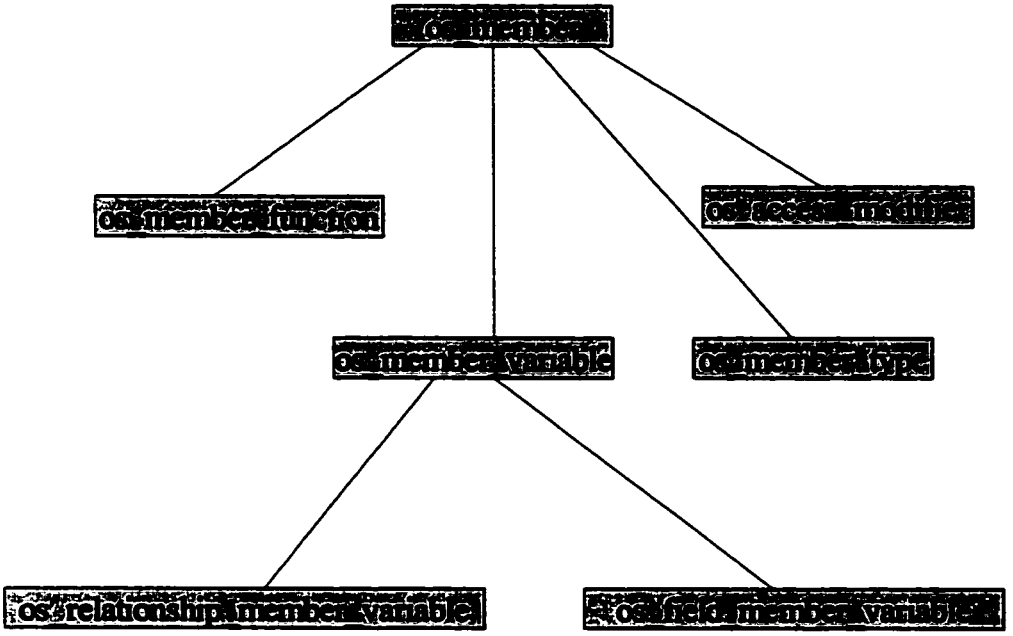

Figure 5 - 11 The os_member class hierarchy.

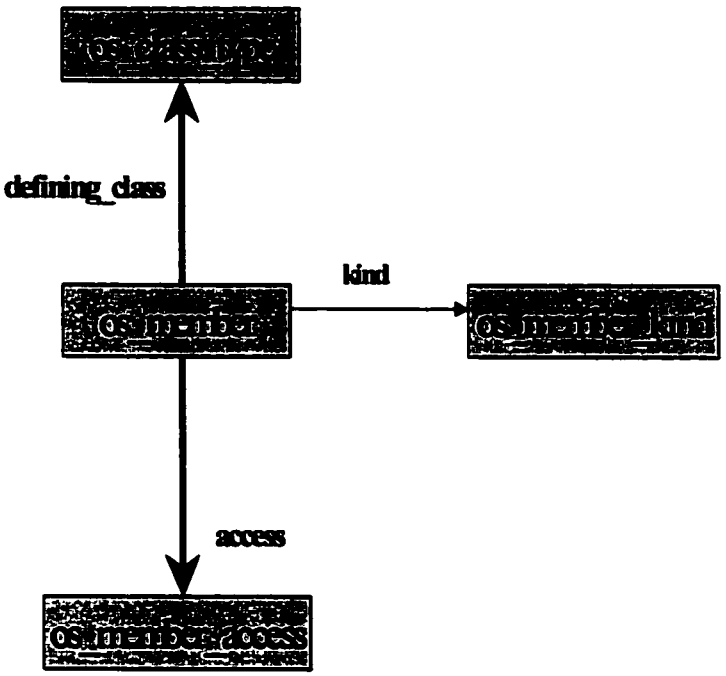

Figure 5 - 12 The structure of the os_member class.

5.11.4.4 The Class os_member_variable

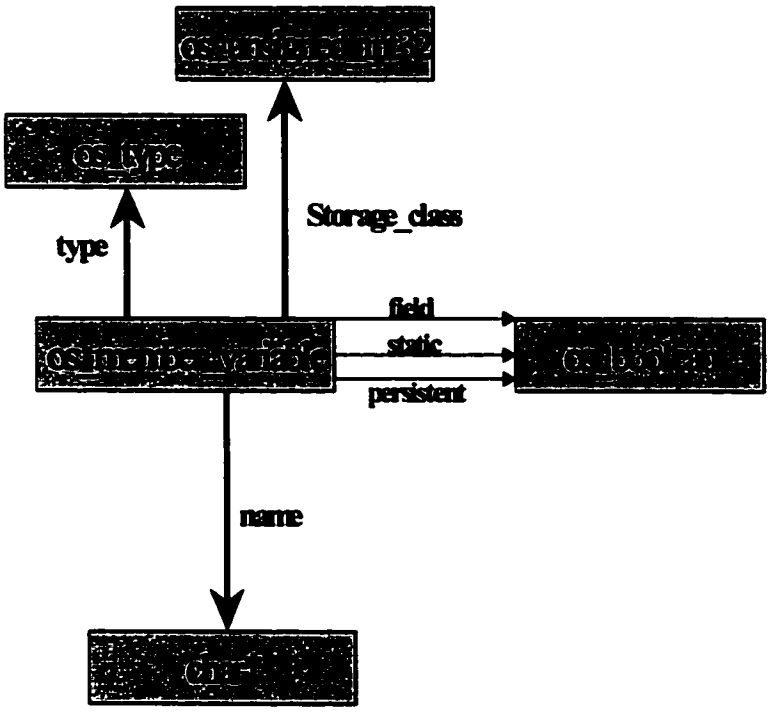

Figure 5 - 13 Attributes of os_member_variable.

5.11.4.5 Fetch and Store Functions

Object Store provides a number of global functions that allow to fetch the value of a specified data member (specified with an os_member_variable) for a specified object, and to store a specified value in a specified data member for a specified object. There are different functions for fetching or storing all the types that a data member can have.

6 A Generic Database Adapter for Multi-tier Distributed CORBA Based Applications

An architectural framework is presented here for realizing Generic Database Adapters in distributed computing environments using CORBA. Generally, the goal of an adapter is to convert the interface of a class into another interface that clients expect (adapter pattern) [8,22,23]. A class adapter uses multiple inheritance to adapt one interface to another, but an object adapter relies on object composition.

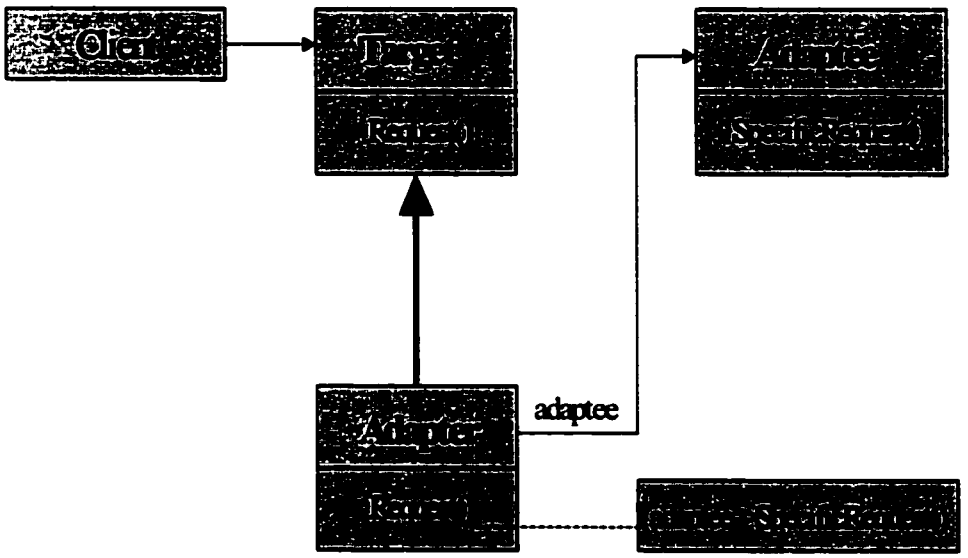

Figure 6 - 1 Object Adapter.

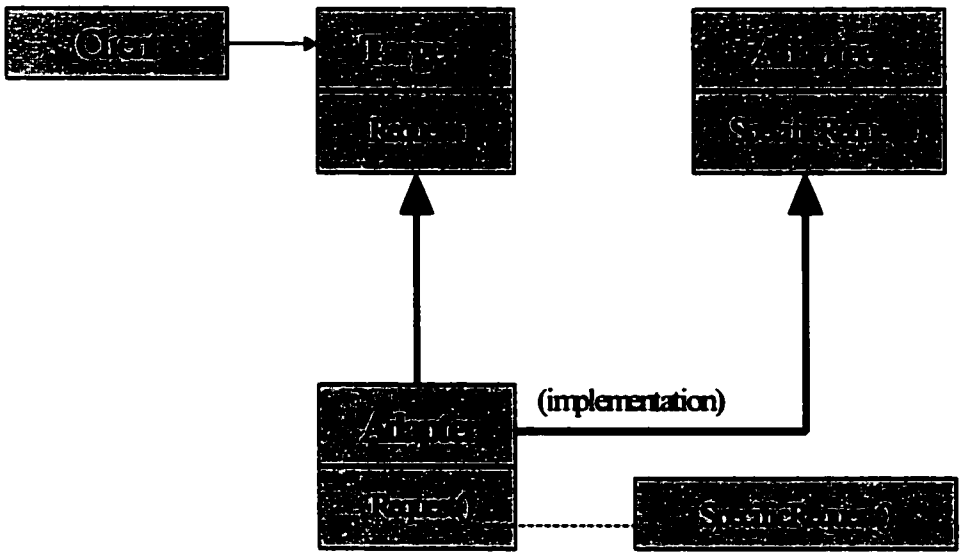

Figure 6 - 2 Class Adapter.

The target defines the domain-specific interface that Client uses, the Client collaborates with objects conforming to the Target interface, the Adaptee defines an existing interface that needs adapting, and the Adapter adapts the interface of the Adaptee to the Target interface.

The role of a Database Adapter is, to provide a virtual memory approach for incarnating CORBA servants. The servant that implements a CORBA object is also a database object that is automatically made persistent by a Database Management System (DBMS). In this case the DBMS is responsible for swapping servant objects in and out of the CORBA server memory space and maintaining persistent storage of all of the servant's attributes and dependencies. This approach is considered the best starting point for implementing a Generic Database Adapter. A generic database adapter represents a database adapter which does not have to be altered in order to manage existing stored information or

information that is to be stored persistently regardless of the changes in the structure of the information that is stored or is to be stored persistently. The generic database adapter must be independent of any of the following elements: (1) the application architecture, (2) the software implementation, (3) the type of the persistent storage (RDBMS or OODBMS) or (4) the structure and type of the information being stored/retrieved to/from the persistent storage. As part of this thesis, only the solution using object-oriented databases (Object Store database) is presented. Any ODBMS provides powerful support for persistence but supports only a limited form of distribution, CORBA on the other hand provides a flexible distribution model but it has only basic support for persistence. At the same time CORBA supports the transmission of operation calls across the network, which means that it allows clients to invoke operations on objects running in remote servers, whereas ODBMS supports the shipping of objects across the network., which means that it allows objects to be copied from storage machines to the address space of the client where they can be accessed directly. By putting the two of them together, CORBA is provided with features necessary for mission-critical applications such as control of concurrent access to persistent objects, data recoverability, data integrity, ACID transactions [31], whereas the Object Database gains extended heterogeneity, by having clients that can be implemented using any programming language for which IDL mapping is defined, better distribution, by having clients that can operate without any knowledge about the logical layout of objects stored in the database, a better authorization mechanism, etc.

All the components of a Generic Database Adapter are specified using the Unified Modeling Language (UML) [10,40] and, where appropriate, the public interfaces are specified using the OMG Interface Definition Language (IDL) [7,8,19]. The IDL specifications and UML design for a Generic Database Adapter, which provides a complete set of database operations, are also presented. The existing set of database operations can be extended in order to support a commercial implementation. Sequence diagrams are used to illustrate the operation of the Generic Database Adapter in managing persistently stored information. One complete implementation solution of the Generic Database Adapter is presented, and recommendations for a second one are provided.

6.1 Functions of a Generic Database Adapter

A Generic Database Adapter requires many components to implement and manage the adaptation of persistently stored objects into CORBA objects (and vice-versa if necessary) in a multi-tier distributed CORBA environment. Such an adapter delivers the following functions:

1. **Database Management**. Provides functionality for creating, opening, and removing distributed object-oriented databases. Also, it maintains centralized information on all the databases accessed during the execution of a specific application.
2. **Object Management**. Provides functionality for creating, storing, retrieving, updating, and removing persistent objects to/from object-oriented databases. Also provides functionality for grouping objects into collections of objects and functionality for creating, maintaining, and removing of such collections. Also it maintains centralized information about all the existing persistent objects.
3. **Query Management**. Provides functionality for retrieving groups of objects based on specific criteria, by querying collections of objects.
4. **Transaction Management**. Provides consistent and reliable access to persistently stored information, by implementing transaction control. This means that all the operations on persistent objects are done within the limits of a database transaction.
5. **Exception Management**. Provides a consistent and reliable exception handling mechanism, by combining together the Object Store and CORBA exception mechanisms.
6. **Concurrency support**. Provides support for concurrent access of multiple clients. Multiple clients can use the generic adapter at the same time with no risk of corrupting persistent information or interfering with each other.
7. **Client Management**. Maintains centralized information on all the clients using the adapter at any moment of time. The Client information includes information

about all the persistent objects accessed during a specific database session, together with transaction information.

8. **Generality Support**. The adapter is independent of the types and structures of the managed objects. This means that the design an implementation of the adapter doesn't have to be changed at all in any circumstances, excepting the case when the existing functionality has to be enhanced.

The aim of the Generic Database Adapter solutions presented in this thesis is to demonstrate the possibility to implement such an adapter, and the advantages offered by such an implementation. The solutions presented here provide all of the following functions: **Database Management, Object Management, Query Management, Transaction Management, Exception Management, Concurrency Support, Client Management, Generality Support.** The design for the first solution and recommendations for another one are presented in the following chapters, by describing the implementation solutions of each of the functions enumerated here, and by presenting the main differences between the two solutions.

6.2 Design Solutions of the Generic Database Adapter

The generic aspect of the adapter results from the fact that, all the objects managed by the adapter are considered as being CORBA objects, or are referenced by void pointers. The client process, as the first tier of the application is responsible to recognize the types of the objects being processed and to have them converted (narrowed) from CORBA objects to the real type expected, in order to invoke operations on them. The database management system, as the third tier of the application, is responsible to store CORBA objects created by different clients. The Database Adapter, as the second tier of the application, is responsible to provide a mechanism for handling persistent objects and operations between the clients and the persistent storage. The types of the objects being handled are being determined based on the name of the type provided by the clients, and

by using the type information stored in the database schema (metadata) for all the types of the objects that are to be stored in the database.

Clients requesting persistent functions from the Database Adapter, which can be CORBA clients or other CORBA servers, have to be aware of the following:

- The types of the objects that are to be stored/retrieved/queried
- The hierarchy of the objects stored persistently (entry points, database names, collection names)
- The structure of the objects that are to be queried (data members)
- The IDL interfaces corresponding to the classes whose objects receive client requests

The entire internal implementation of the Generic Database Adapter is completely transparent for clients requesting persistent services from the adapter. The clients are aware only of the fact that they can store/retrieve/query persistent objects, and invoke operations on those objects (by using the IDL interfaces exported by the Generic Database Adapter and the IDL interfaces corresponding to the classes whose objects have to be stored persistently).

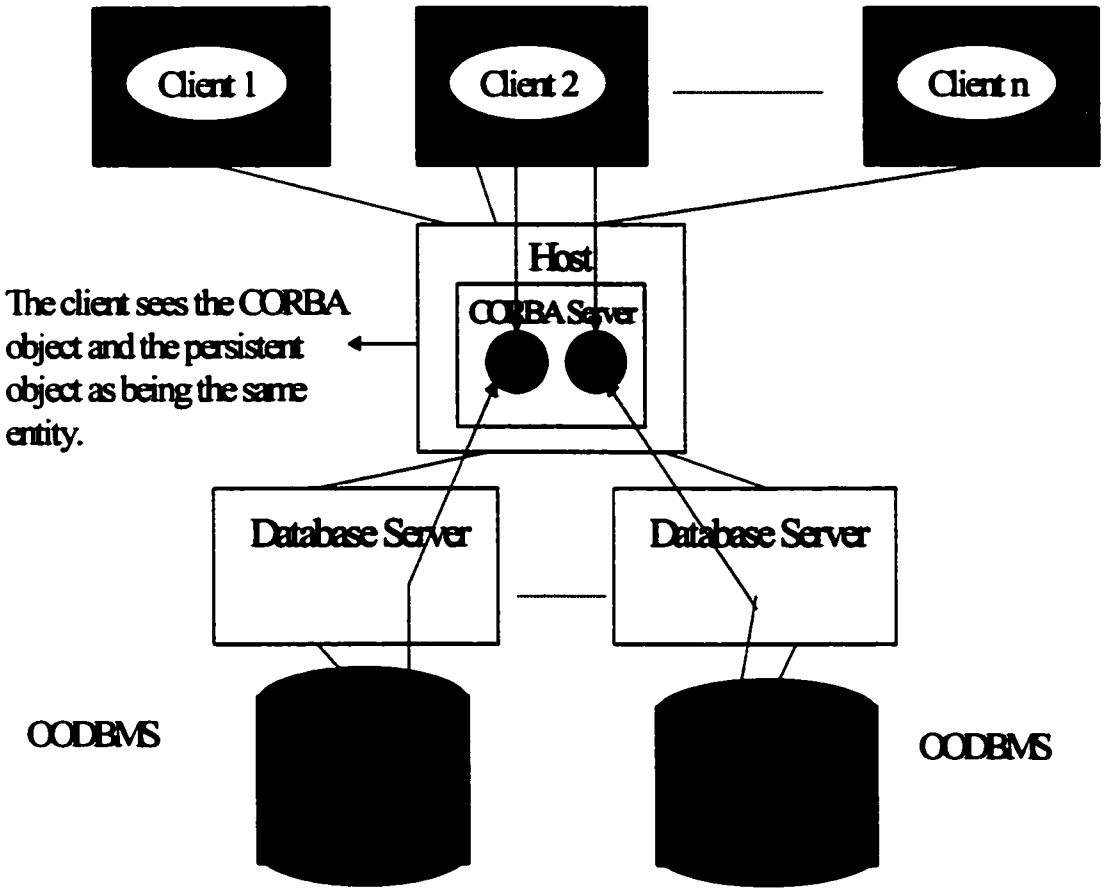

Figure 6 - 3 Outside view of the Generic Database Adapter.

The two solutions presented as part of this thesis are:

- The Persistent CORBA objects solution, and
- The Front End solution.

Only the first solution is designed and implemented completely as part of this thesis, whereas the second solution is described as future research. Both solutions provide exactly the same functionality and provide exactly the same external interface for all Database Adapter clients. The differences between the two solutions consist in the way the internal mechanism for mapping persistent objects into CORBA objects and back to persistent objects are implemented.

6.2.1 The Persistent CORBA Objects Solution

This solution manages persistent objects by having all the classes whose objects are to be persistent, inherit from the BOAImpl classes generated by IDL. Thus the objects created by the Generic Database Adapter can be persistent objects and CORBA objects at the same time, which means that the Database Adapter clients can work directly on persistent objects. The internal implementation of this solution is illustrated in the following diagrams:

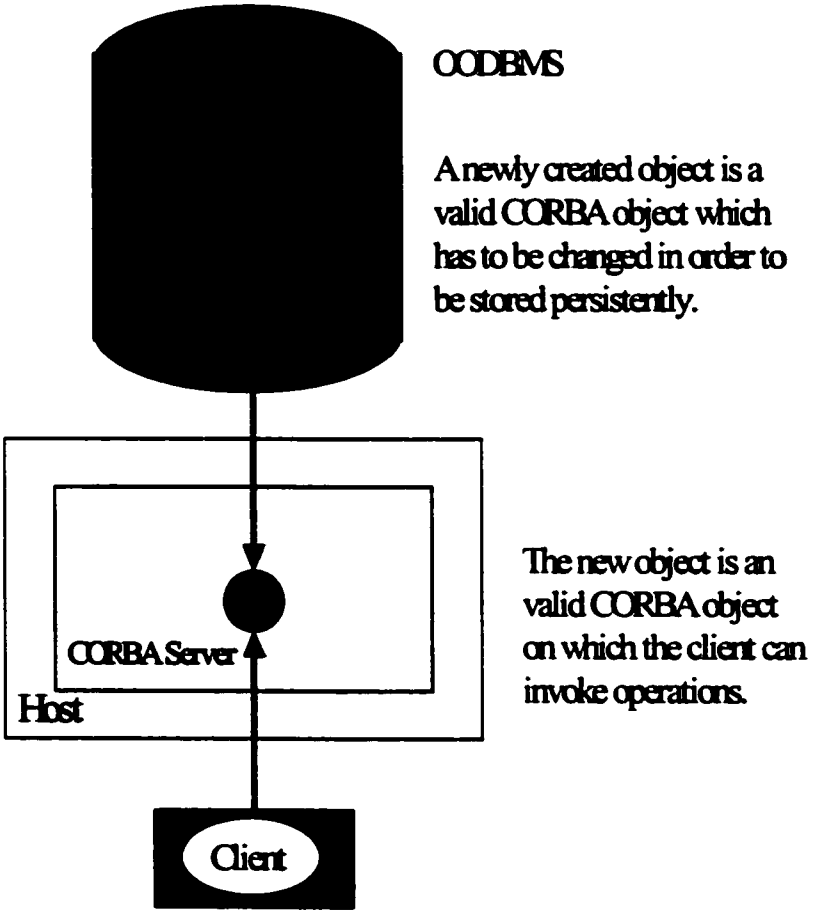

Figure 6 - 4 Create a persistent object.

The advantage of having a CORBA object stored persistently into an Object Store database, presents the advantage that a client can invoke operations directly on that object, because that object is still a valid CORBA object. The disadvantage presented by this solution is that before a specific object can be committed to persistent storage it has

to be changed (the CORBA object contains pointers to transient memory). This means that future retrievals of that object from persistent storage, will result in objects which are not valid CORBA objects, thus they cannot be accessed by CORBA clients. Thus, each time a persistent object that was already committed to the database, has to be accessed, a new CORBA object is created, stored persistently, the content of the existing object is copied into the new one, and the existing object is deleted from persistent storage.

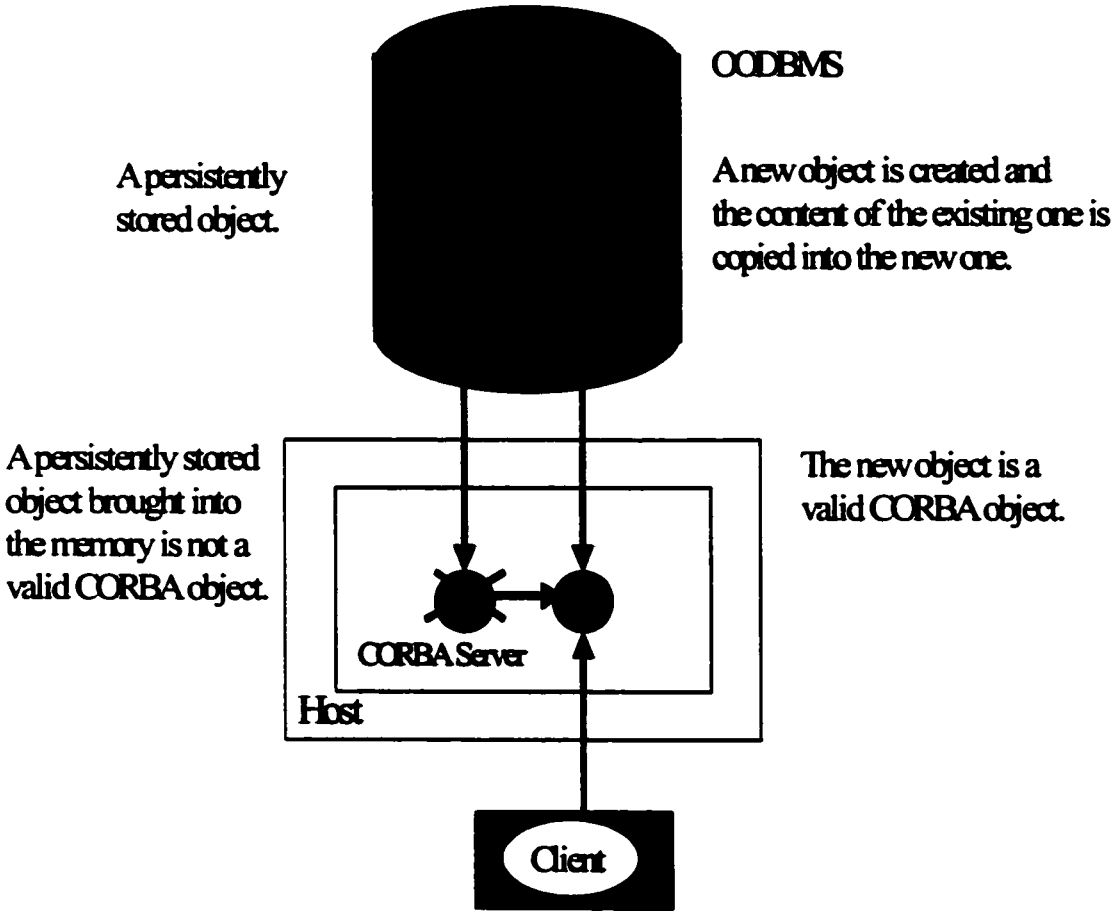

Figure 6 - 5 The retrieval of a persistent object.

6.2.2 The Front-end Solution.

This solution also, manages persistent objects by having all the classes whose objects are to be persistent, inherit from the BOAImpl classes generated by IDL. The difference is that each time a persistent object is retrieved from the database, a new CORBA object has to be created, the content of the persistent object is copied to the new CORBA object and a reference is returned back to the client that requested that object. Thus the objects created by the Generic Database Adapter are not persistent objects and CORBA objects at the same time, instead the Database Adapter clients work on a proxy object whose content is copied internally to the persistent object when the client commits the changes to persistent storage. The internal implementation of this solution is illustrated in the following diagrams:

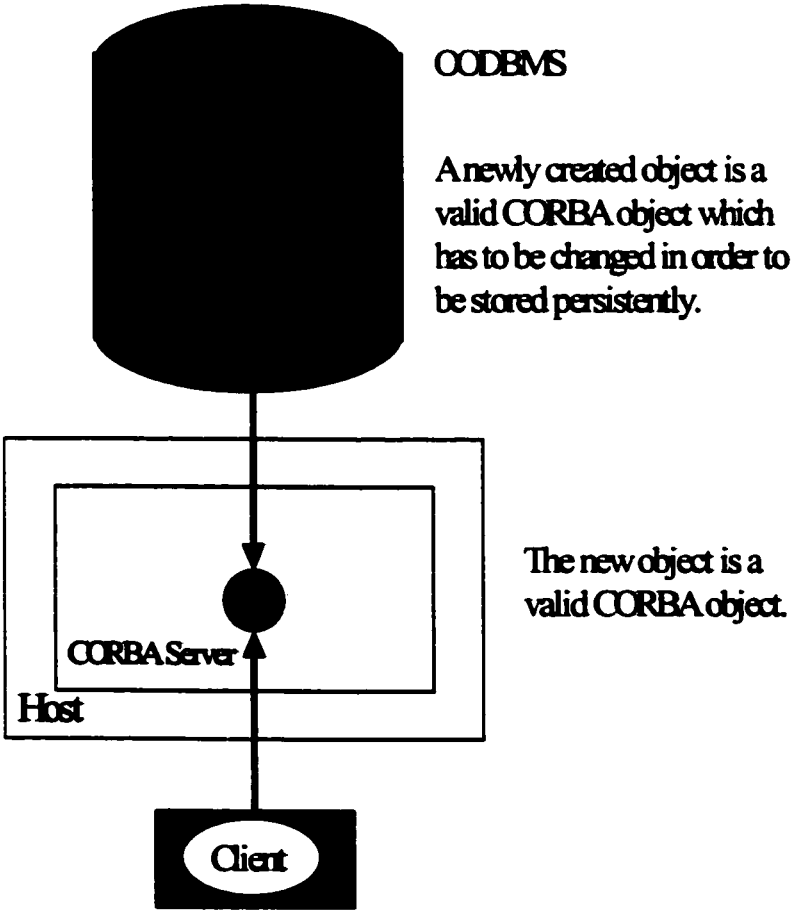

Figure 6 - 6 Create a persistent object.

The advantage of this solution is that no new database objects have to be created each time a persistently stored object is retrieved. The disadvantage of this solution is that a new CORBA object has to be created each time an object is retrieved, thus the client does not work directly on the persistent object, but on a proxy of that object

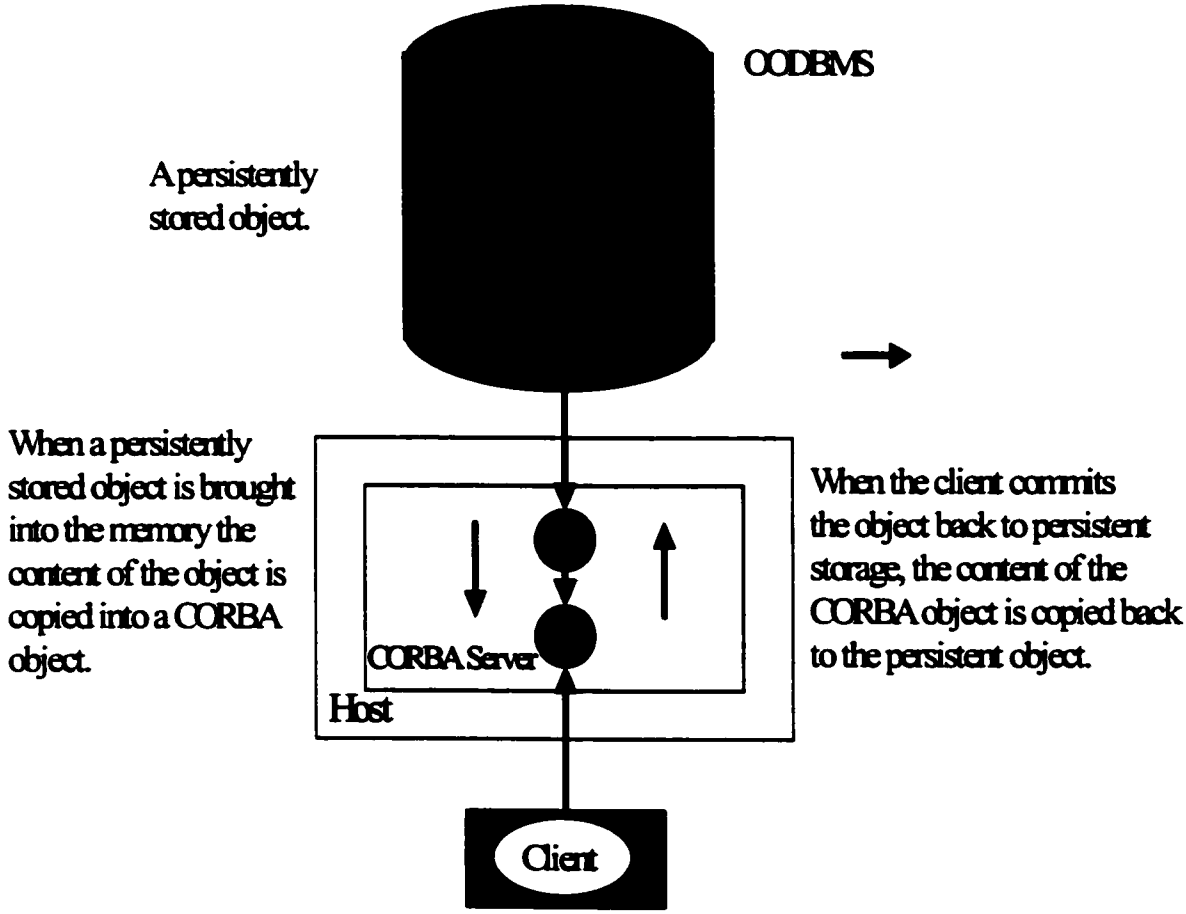

Figure 6 - 7 The retrieval of a persistent object.

6.2.3 Real Use Cases.

Further, the use case diagrams describing the Generic Database Adapter solutions are presented. The diagrams are then followed by a detailed description of the use cases.

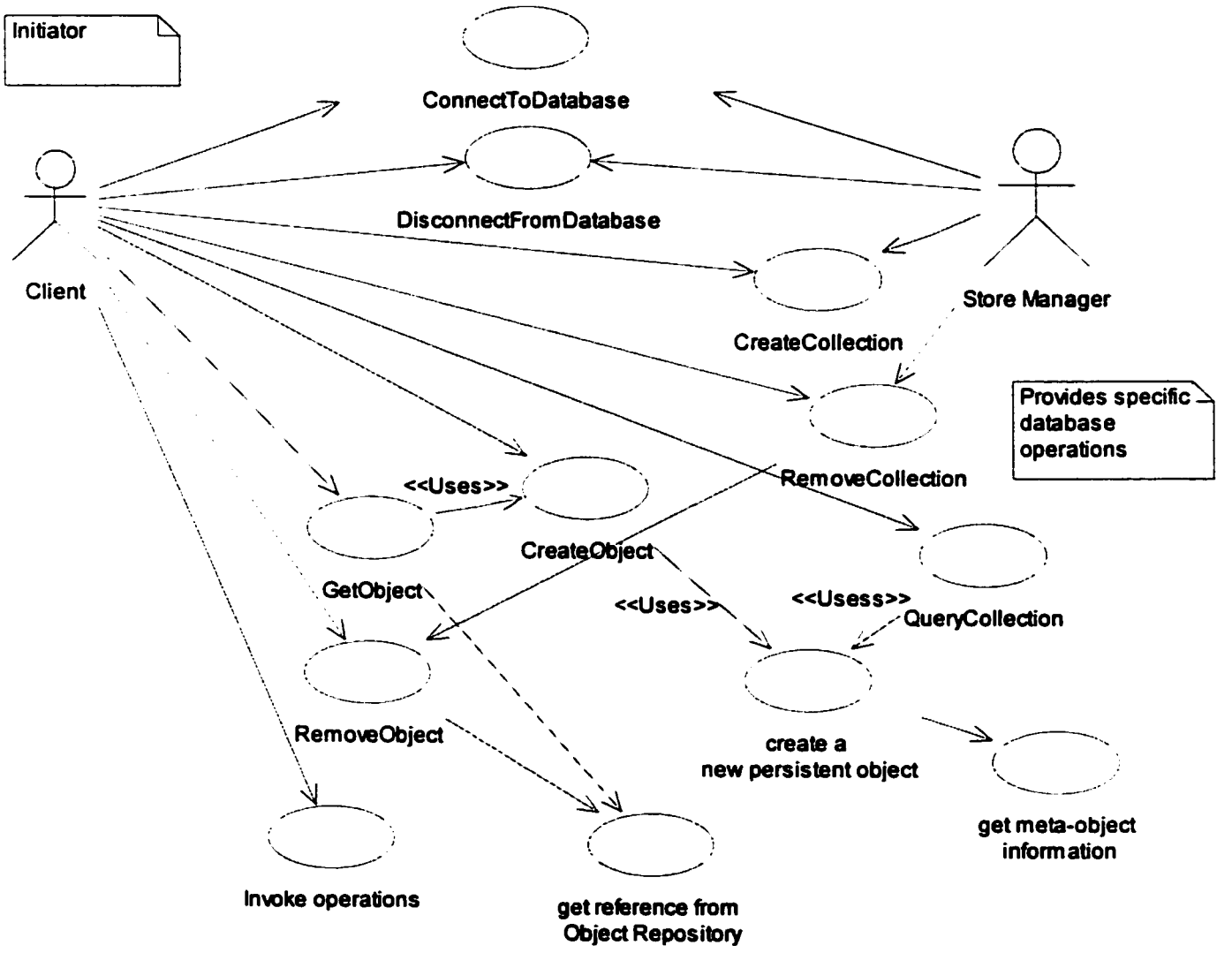

Figure 6 - 8 The persistent CORBA objects approach.

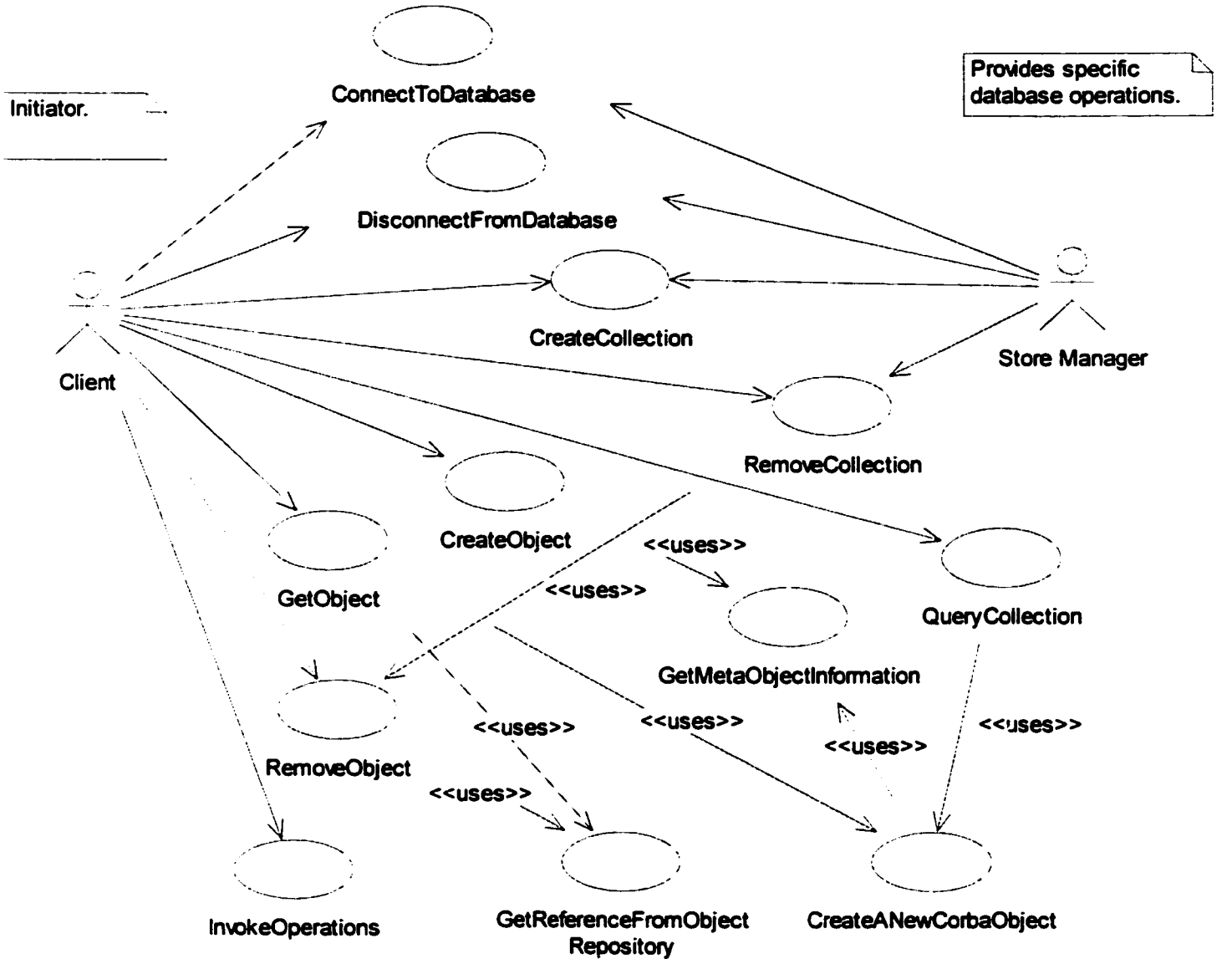

Figure 6 - 9 The front-end solution.

6.2.3.1 Real Use Case "Connect to Database"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests to be connected to a specific Object Store database. The Database Adapter opens the requested database, starts a database transaction corresponding to this request, and registers the new client.

Typical Course of Events

1. The Database Adapter client requests to be connected to a specific database.	2. The Database Adapter verifies if the client is not already connected, opens the requested database (done by the Store Manager) and registers this fact, opens a database transaction and registers the client as connected. Also a list of objects Ids that are to be used by this client is initialized. The Database Adapter informs the Database Adapter Client that it was successfully connected to the database.

Alternative courses:

step (2):

- If the database name provided by the Database Adapter Client does not exist, the Database Adapter returns immediately, informing the client.
- If the database corresponding to the database name provided by the client is already open, the Database Adapter will not open it again.
- If the client is already connected, the Database Adapter will not connect the client again, and will not start a new database transaction.

- If the Database Adapter cannot create a database transaction, or it cannot open the database, or it cannot register the client, the client will be informed about the failure of this operation.

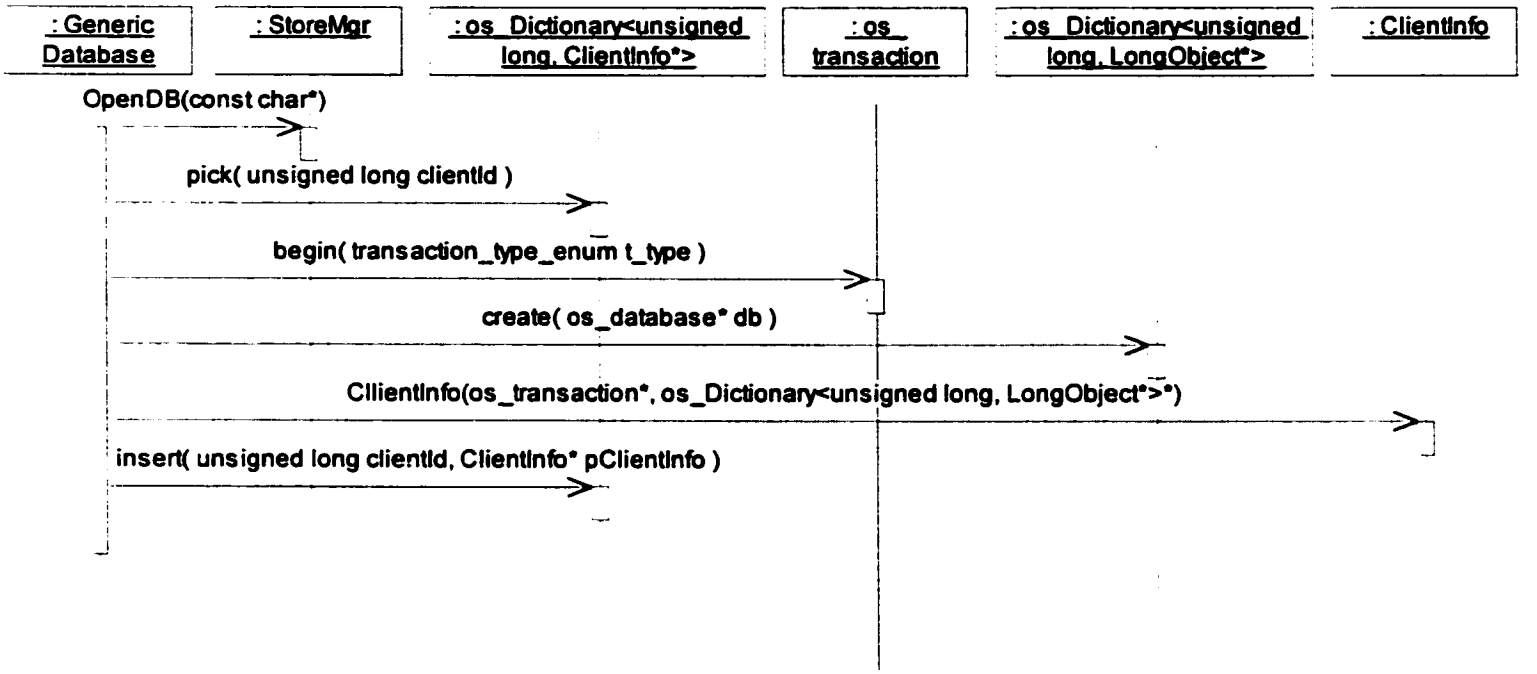

Figure 6 - 10 Connect to database sequence diagram.

6.2.3.2 Real Use Case "Create Collection"

Actors `Database Adapter Client` (initiator)

Overview This process is triggered when a specific Database Adapter Client requests the creation of a collection of objects in a specific database. The Database Adapter verifies if the specified database collection is not already created, and creates a new one.

Typical Course of Events

1. The Database Adapter client requests the creation of a new object collections.	2. The Database Adapter verifies if a collection with the same name is not already created, checks if the target database is already opened, and creates and names a new database collection. The Database Adapter informs the Database Adapter Client that the collection requested was successfully created.

Alternative courses:

step (2):

- If the database name provided by the Database Adapter Client does not exist, the Database Adapter returns immediately, informing the client.
- If the database collection corresponding to the database collection name provided by the client is already created, the Database Adapter will return immediately, informing the client.
- If the creation of the collection or the naming of the collection fails, the Database Adapter will return immediately, informing the client.

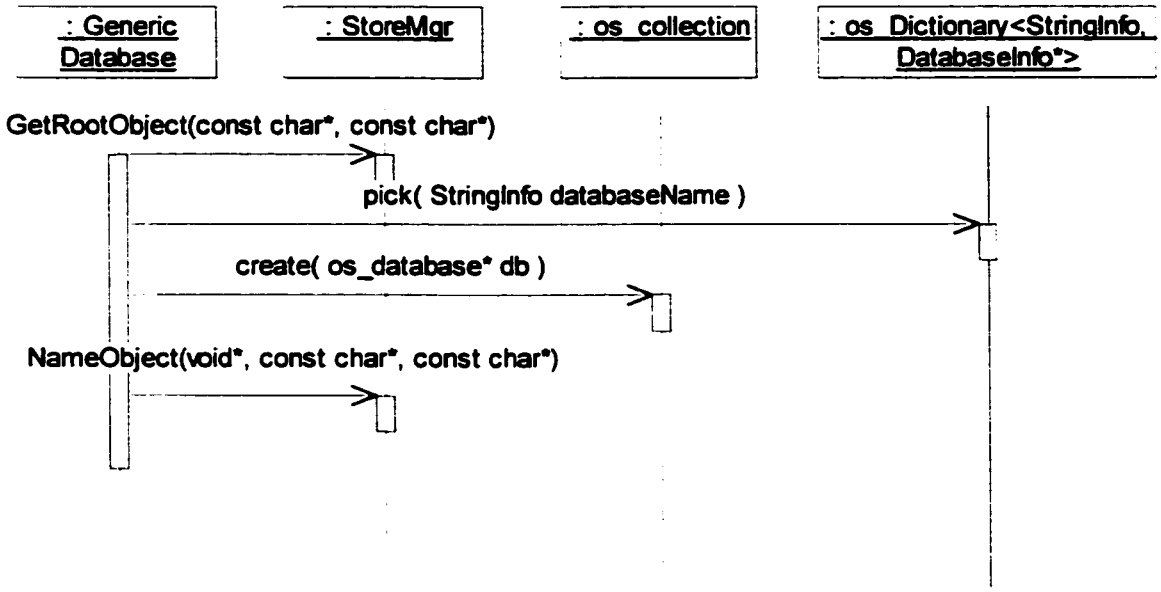

Figure 6 - 11 Create collection sequence diagram.

6.2.3.3 Real Use Case "Create Object"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests the creation of a new database object as part of a specific collection and a specific database. The Database Adapter opens the requested database, creates a new object and inserts it into the specified collection.

Typical Course of Events

1. The Database Adapter client requests the creation of a new object, by providing the name of the database, the collection name, the object name, the name of the object type, and the	2. The Database Adapter retrieves type specific information from the database schema based on the string identifying the type, and retrieves the specified database and collection objects based

identifier of self.	on the names provided by the client. Based on this information, the client creates a new object in the specified database and collection, registers the name of the object and the object reference in the object repository, and registers the client as a user of this object. Finally, it returns a reference to the newly created object back to the Database Adapter client

Alternative courses:

step (2):

- If the collection corresponding to the collection name provided by the Database Adapter Client doesn't exist, a new database collection is created. If specific object type information cannot be retrieved from the database schema, or the client is not connected to the specified database, or another object having the same name was previously created, the Database Adapter returns immediately informing the client.

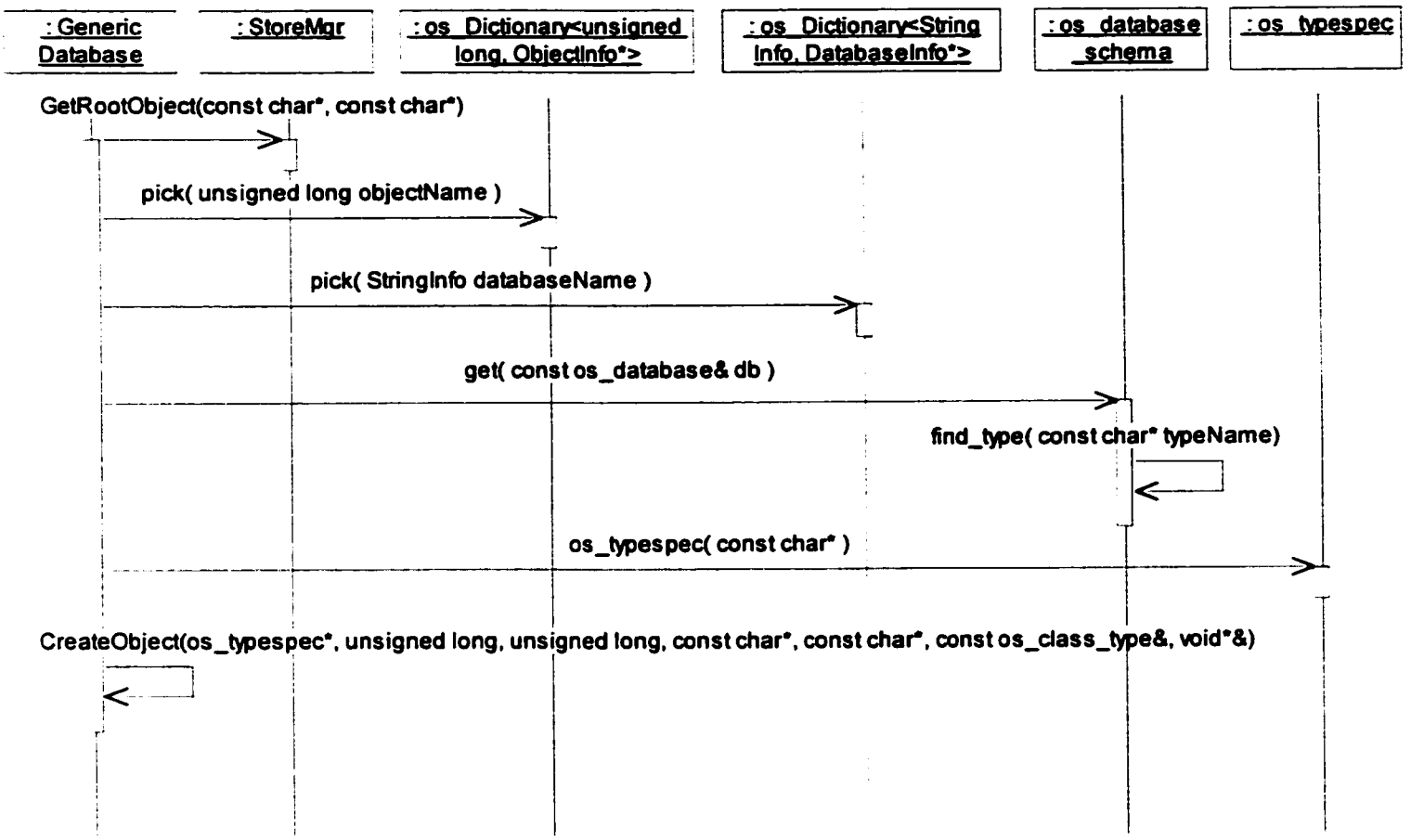

Figure 6 - 12 Create object sequence diagram.

6.2.3.4 Real Use Case "Get Object"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests the retrieval of an existing database object, stored in a specific collection and a specific database. The Database Adapter opens the requested database, retrieves the requested object and returns a reference to this object back to the requester.

Typical Course of Events

1. The Database Adapter client requests the retrieval of a specific database object, by providing the name of the database, the collection name, the object name, the name of the object type, and the identifier of self.	2. The Database Adapter verifies if a corresponding object having the same identifier (object name) is already stored in the database, retrieves the specified object, registers the client as a user of this object, and returns a reference to the newly retrieved object back to the Database Adapter client

Alternative courses:

step (2):

- If the database corresponding to the database name provided by the Database Adapter Client does not exist, or the object requested doesn't exist, or the client is not connected to the specified database, the Database Adapter returns immediately, informing the client. If the collection corresponding to the collection name provided by the Database Adapter Client doesn't exist, a new database collection is created.

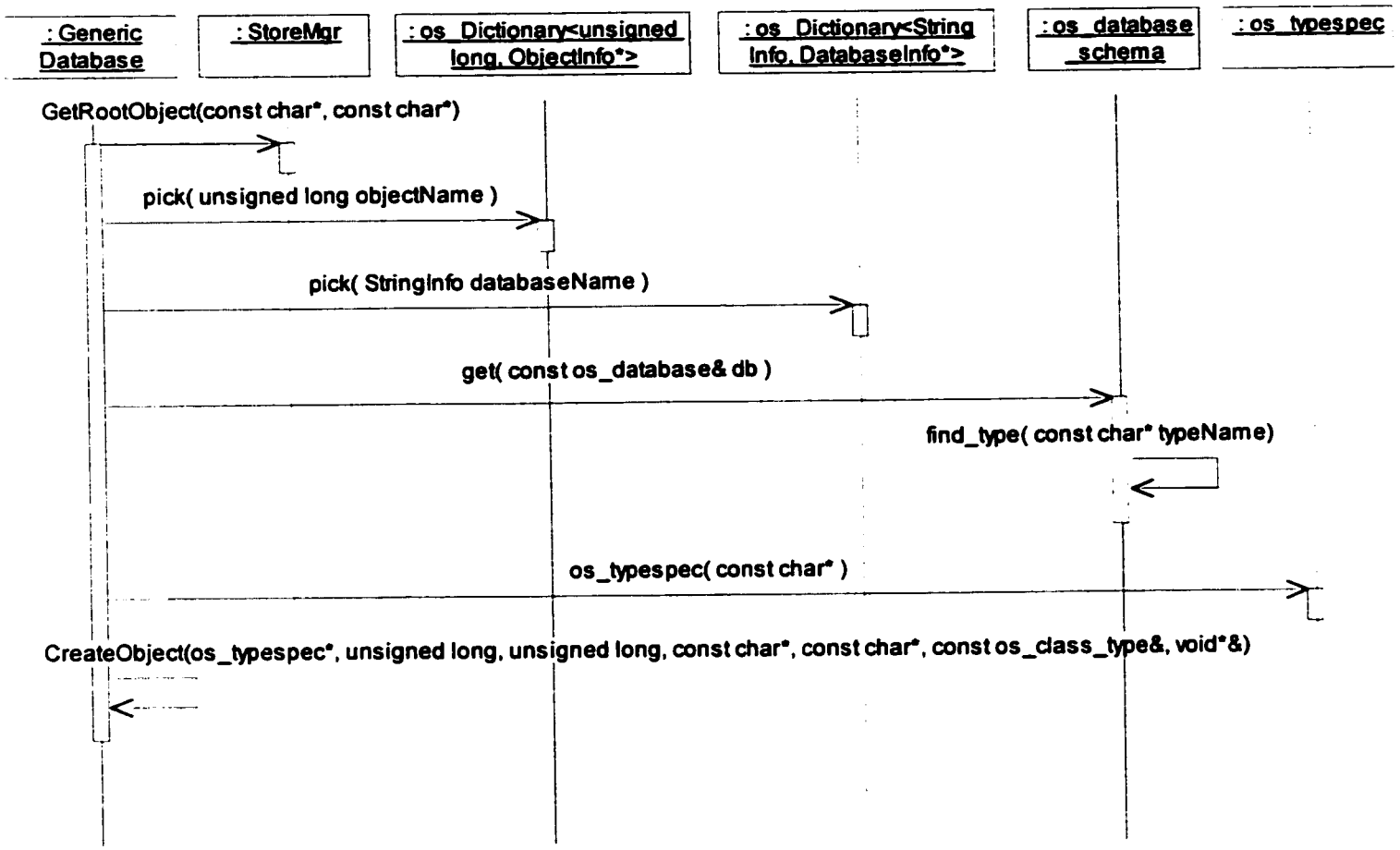

Figure 6 - 13 Get object sequence diagram.

6.2.3.5 Real Use Case "Remove Object"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests the removal of an existing database object, stored in a specific collection and a specific database. The Database Adapter opens the requested database, removes and deletes the requested object from the corresponding collection, and returns a confirmation back to the requester.

Typical Course of Events

1. The Database Adapter client requests the removal of a specific database object, by providing the name of the database, the collection name, and the object name.	2. The Database Adapter verifies if a corresponding object having the same identifier (object name) is already stored in the database, removes and deletes the specified object from the corresponding collection, removes the object information and reference information from the object repository, and returns a confirmation back to the Database Adapter client

Alternative courses:

step (2):

- If the database corresponding to the database name provided by the Database Adapter Client does not exist, or the collection corresponding to the collection name provided by the Database Adapter Client doesn't exist, or the object requested doesn't exist, or the client is not connected to the specified database, the Database Adapter returns immediately, informing the client.

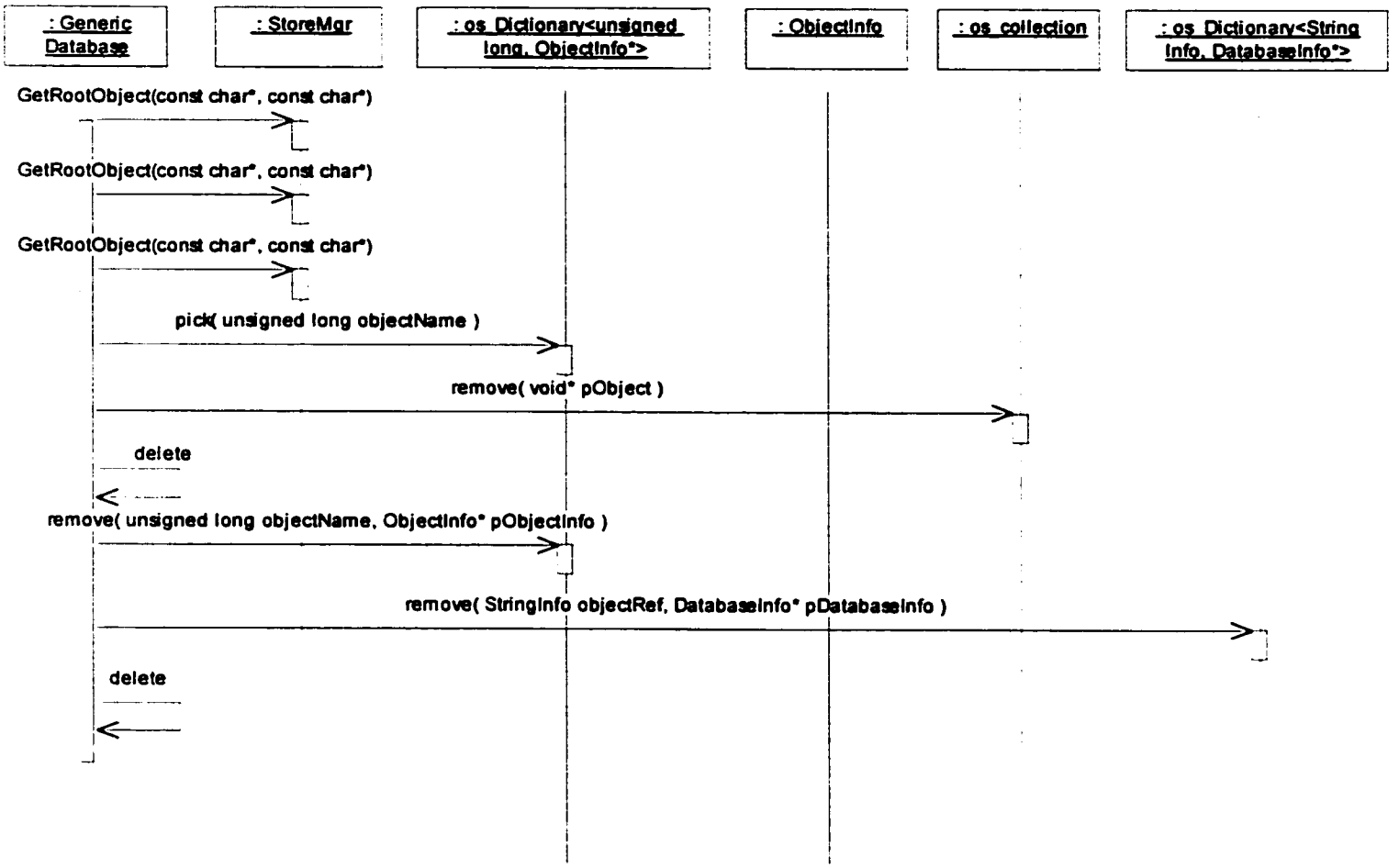

Figure 6 - 14 Remove object sequence diagram.

6.2.3.6 Real Use Case "Remove Collection"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests the removal of an entire database collection, stored in a specific database. The Database Adapter opens the requested database, removes and deletes all the objects from the corresponding collection, destroys the collection, and returns a confirmation back to the requester.

Typical Course of Events

1. The Database Adapter client requests the removal of a specific database collection, by providing the name of the database, and the collection name.	2. The Database Adapter verifies if a corresponding collection having the same identifier (object name) is already stored in the database, removes and deletes all the objects from the corresponding collection, removes the object information from the object repository for all the objects, destroys the collection, and returns a confirmation back to the Database Adapter client

Alternative courses:

step (2):

- If the database corresponding to the database name provided by the Database Adapter Client does not exist, or the collection corresponding to the database name provided by the Database Adapter Client doesn't exist, or the client is not connected to the specified database, the Database Adapter returns immediately, informing the client.

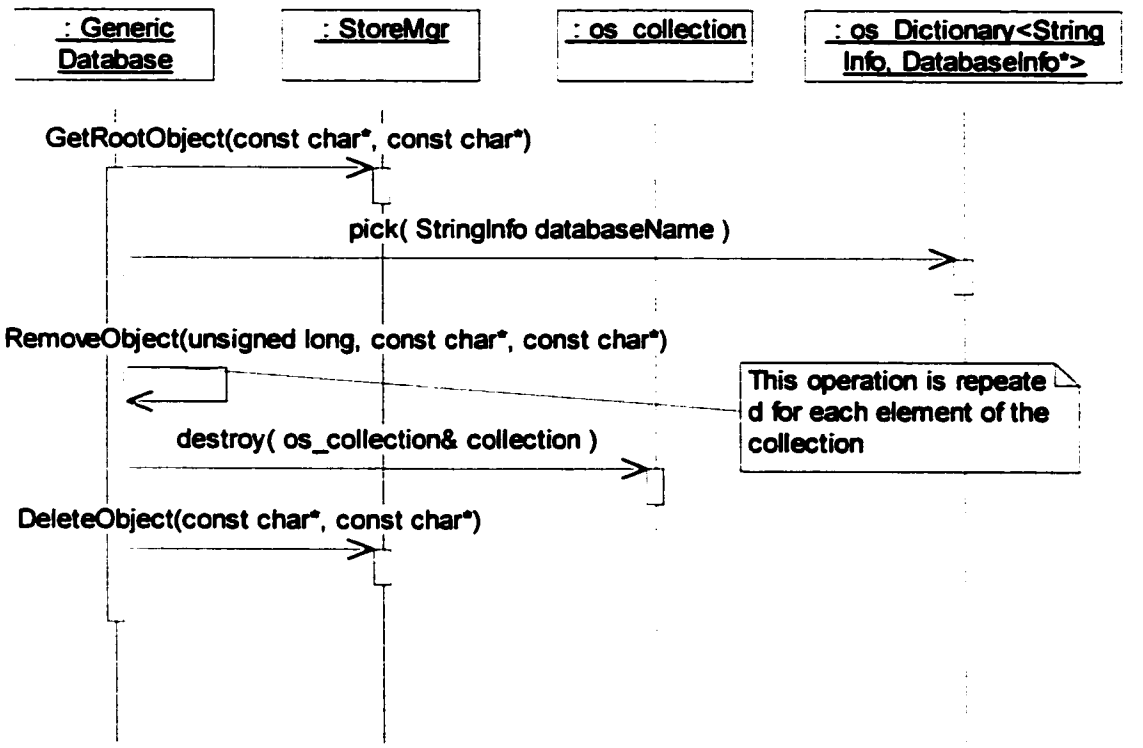

Figure 6 - 15 Remove collection sequence diagram.

6.2.3.7 Real Use Case "Query Collection"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests the retrieval of a set of objects meeting a certain criteria, from an existing database collection, stored in a specific database. The Database Adapter opens the requested database, queries the corresponding collection, wraps the result inside a CORBA object, and returns a reference to this object back to the requester. Further the requester can navigate through the result of the query and read or change the information contained within these objects.

Typical Course of Events

1. The Database Adapter client requests the retrieval of a set of objects stored in a specific database collection, by providing the name of the database, the collection name, a specific criteria based on the characteristics of the objects, the name of the object type, and the identifier of self.	2. The Database Adapter verifies if a corresponding collection having the same identifier (collection name) is already stored in the database, queries the collection, wraps the result of the collection in a CORBA object, registers all the objects as being used by the client, and returns a reference to this object back to the Database Adapter client
3. The Database Adapter client navigates through the result of the query (get first, last, next object) and reads or updates different objects.	4. The Database Adapter returns a reference back to the client for each of the objects requested, which are part of the result of the query.

Alternative courses:

step (2):

- If the database corresponding to the database name provided by the Database Adapter Client does not exist, or the collection corresponding to the collection name provided by the Database Adapter Client doesn't exist, or the object criteria is not correct, or the client is not connected to the specified database, the Database Adapter returns immediately, informing the client.

step(4):

- If the end of the collection is reached, the Database Adapter returns immediately, informing the client.

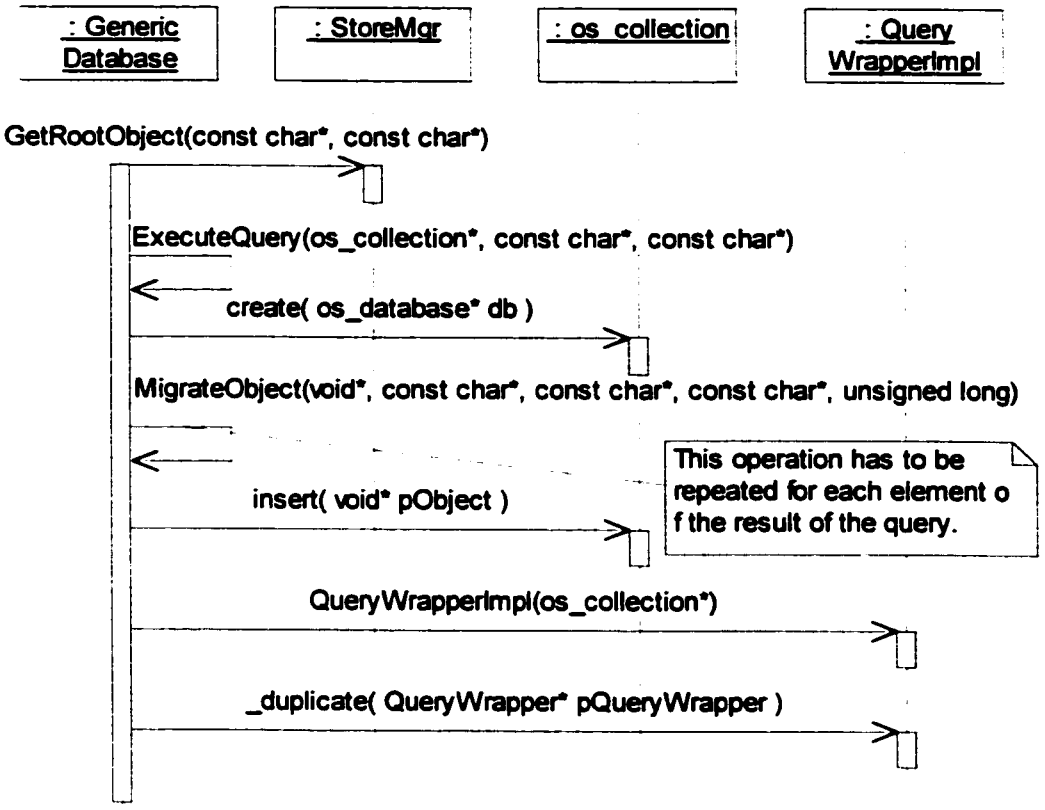

Figure 6 - 16 Query collection sequence diagram.

6.2.3.8 Real Use Case "Disconnect from Database"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered when a specific Database Adapter Client requests to have the database session started closed. The Database Adapter commits all the objects accessed by the client back to the persistent storage, and disconnects the client.

Typical Course of Events

1. The Database Adapter client requests to be disconnected from the Database Adapter (from all the databases accessed) by providing only the identifier of self.	2. The Database Adapter verifies if the client is already connected, and commits all the objects accessed by the client back to the persistent storage by visiting the list of objects accessed by that specific client, and by committing the database transaction started and removes the client information from the list of connected clients. The Database Adapter informs the Database Adapter client that it was successfully disconnected from all the objects and databases accessed.

Alternative courses:

step (2):

- If the client is not connected to the Database Adapter, the Database Adapter returns immediately, informing the client.
- If the objects accessed by the client cannot be committed back to the persistent storage, or the database transaction started cannot be committed, the Database Adapter will return immediately.

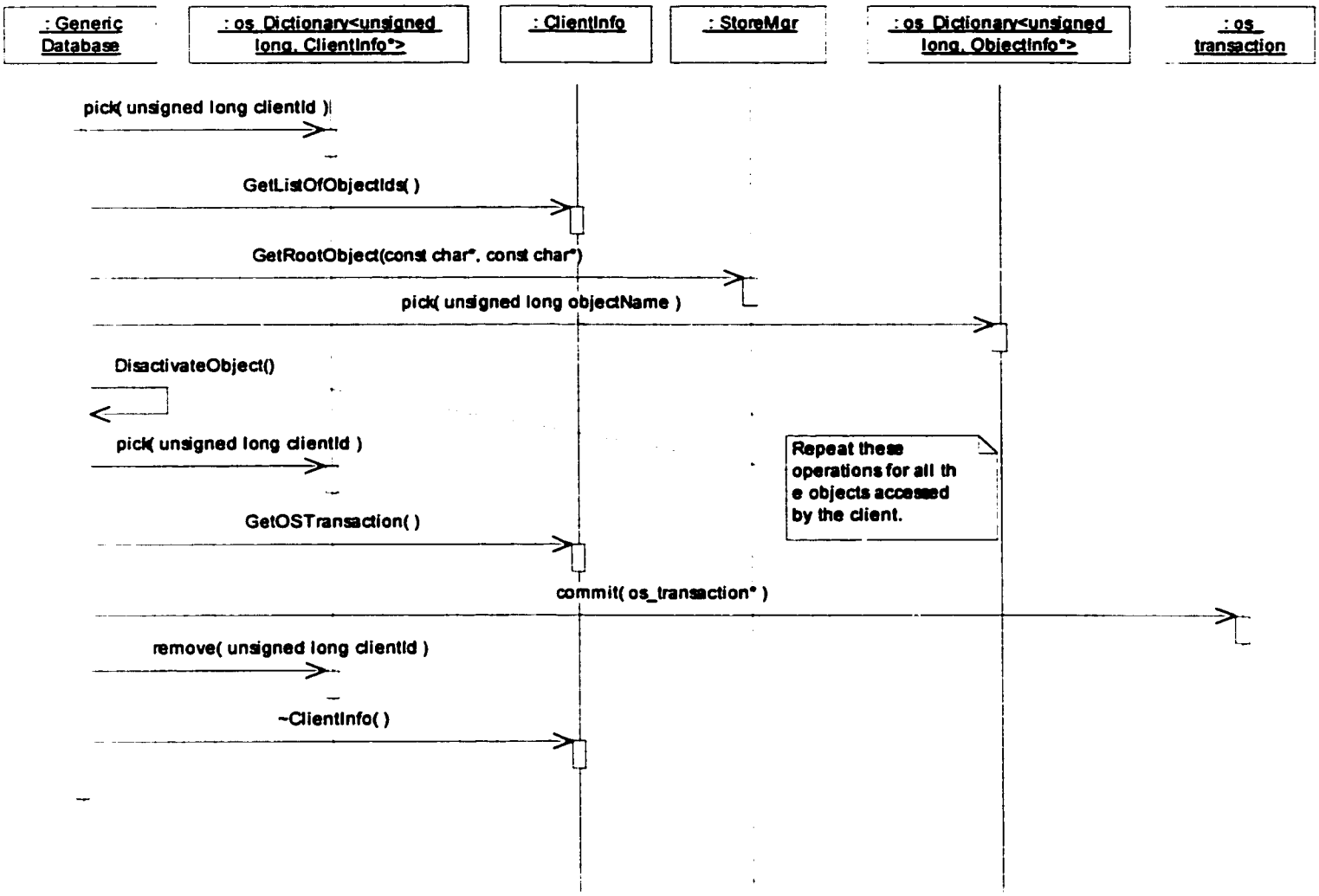

Figure 6 - 17 Disconnect from database sequence diagram.

6.2.3.9 Real Use Case "Get Meta-Object Information"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered indirectly when a specific Database Adapter Client requests the creation of a new object corresponding to the name of the type provided. The Database Adapter retrieves specific meta-object information corresponding to the type provided.

Typical Course of Events

1. The Database Adapter client requests the creation of a new object in a specific database and collection, name of the database, the collection name, the object name, the name of the object type, and the identifier of self.	2. For this specific use case the The Database Adapter retrieves the corresponding meta-data from the database schema and it uses it to create a new object.

Alternative courses:

step (2):

- If the database schema corresponding to the current application doesn't contain information about that specific type, the Database Adapter generates an exception and returns immediately, informing the client.

6.2.3.10 Real Use Case "Create New Persistent Object"

Actors `CORBA Client` (initiator), `Store Manager`

Overview This process is triggered indirectly when a specific Database Adapter Client requests the creation of a new object corresponding to the name of the type provided. The Database Adapter creates a new persistent object corresponding to the type provided.

Typical Course of Events

1. The Database Adapter client requests the creation of a new object in a specific database and collection, by providing the name of the database, the collection name, the object name, the name of the object type, and the identifier of self.	2. For this specific use case the Database Adapter creates a new persistent object corresponding to the type provided.

Alternative courses:

step (2):

- If the database schema corresponding to the current application doesn't contain information about that specific type, the Database Adapter generates an exception and returns immediately, informing the client.

6.2.3.11 Real Use Case "Get Reference from Object Repository"

Actors	`CORBA Client` (initiator), `Store Manager`
Overview	This process is triggered indirectly by the Database Adapter Client each time the retrieval of the information required involves the use of the persistent reference corresponding to a persistent object stored in the database. The Database Adapter retrieves the persistent object reference from the object information repository for a specific object name and uses it to access the persistently stored object.

Typical Course of Events

1. The Database Adapter client requests the retrieval, or removal of an object, from a specific database and	2. For this specific use case the The Database Adapter retrieves the corresponding persistent object

collection, by providing the name of the database, the collection name, the and the object name, the type name, and the identifier of self.

reference from the object repository and it uses it to retrieve that specific object.

Alternative courses:

step (2):

- If the object information repository does not contain information about the specified object name the Database Adapter generates an exception and returns immediately, informing the client.

6.2.4 Object Design

This section presents the software classes (i.e. detailed static model) and the IDL interfaces used to implement the use cases described in the previous chapter.

6.2.4.1 IDL interfaces

This chapter describes the IDL interfaces used in the implementation of the Generic Database Adapter.

```
// IDL for the Generic Database Adapter
interface GenericDatabaseAdapter
{
   exception AdapterException{ string reason; };

   boolean ConnectToDatabase( in string databaseName, in long clientId )
      raises (AdapterException);

   boolean DisconnectFromDatabase( in long clientId )
      raises (AdapterException);

   boolean CreateCollection( in string collectionName, in string databaseName )
      raises (AdapterException);

   boolean RemoveCollection( in string collectionName, in string databaseName )
      raises (AdapterException);

      boolean CreateObject( in string objectType, in long objectName, in long clientId,
                  in string collectionName, in string databaseName )
      raises (AdapterException);

   Object GetObject( in string objectType, in long objectName, in long clientId,
               in string collectionName, in string databaseName )
      raises (AdapterException);

   boolean RemoveObject( in long objectName, in string collectionName,
               in string databaseName )
      raises (AdapterException);

   QueryWrapper QueryCollection( in string collectionName, in string databaseName,
     in long clientId, in string sqlString, in string objectType, in string objectPointerType )
      raises (AdapterException);
};
```

Figure 6 - 18 IDL interface for the Generic Database Adapter.

```
// IDL for the Generic Database Adapter (Query interface)

interface QueryWrapper
{
   exception QueryWrapperException{ string reason; };

   Object GetFirst( ) raises (QueryWrapperException);
   Object GetLast( ) raises (QueryWrapperException);
   Object GetNext( ) raises (QueryWrapperException);
 };
```

Figure 6 - 19 The IDL query interface.

6.2.4.2 Object Model

This section presents the software classes used for implementing the Generic Database Adapter.

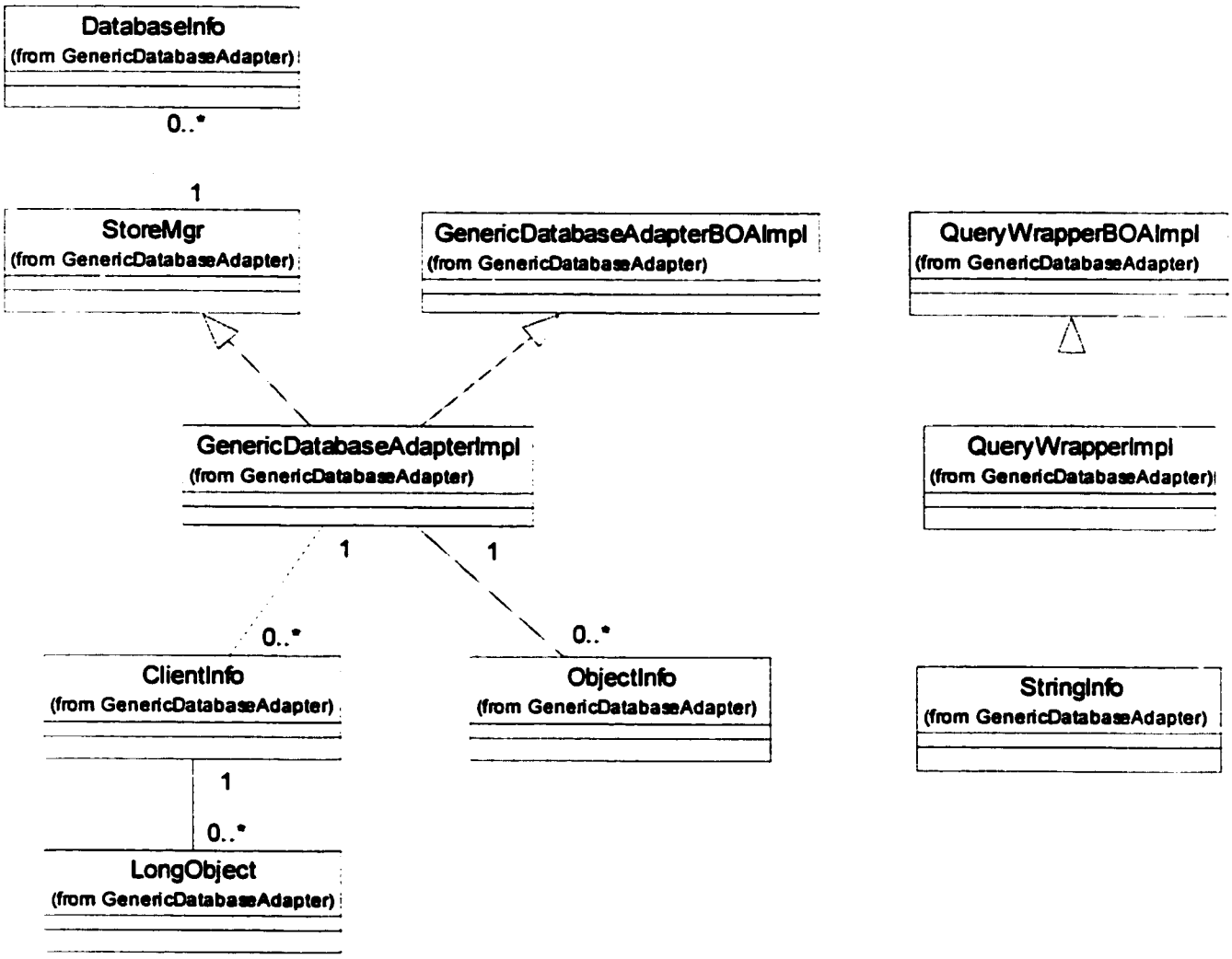

Figure 6 - 20 The Logical Object Model.

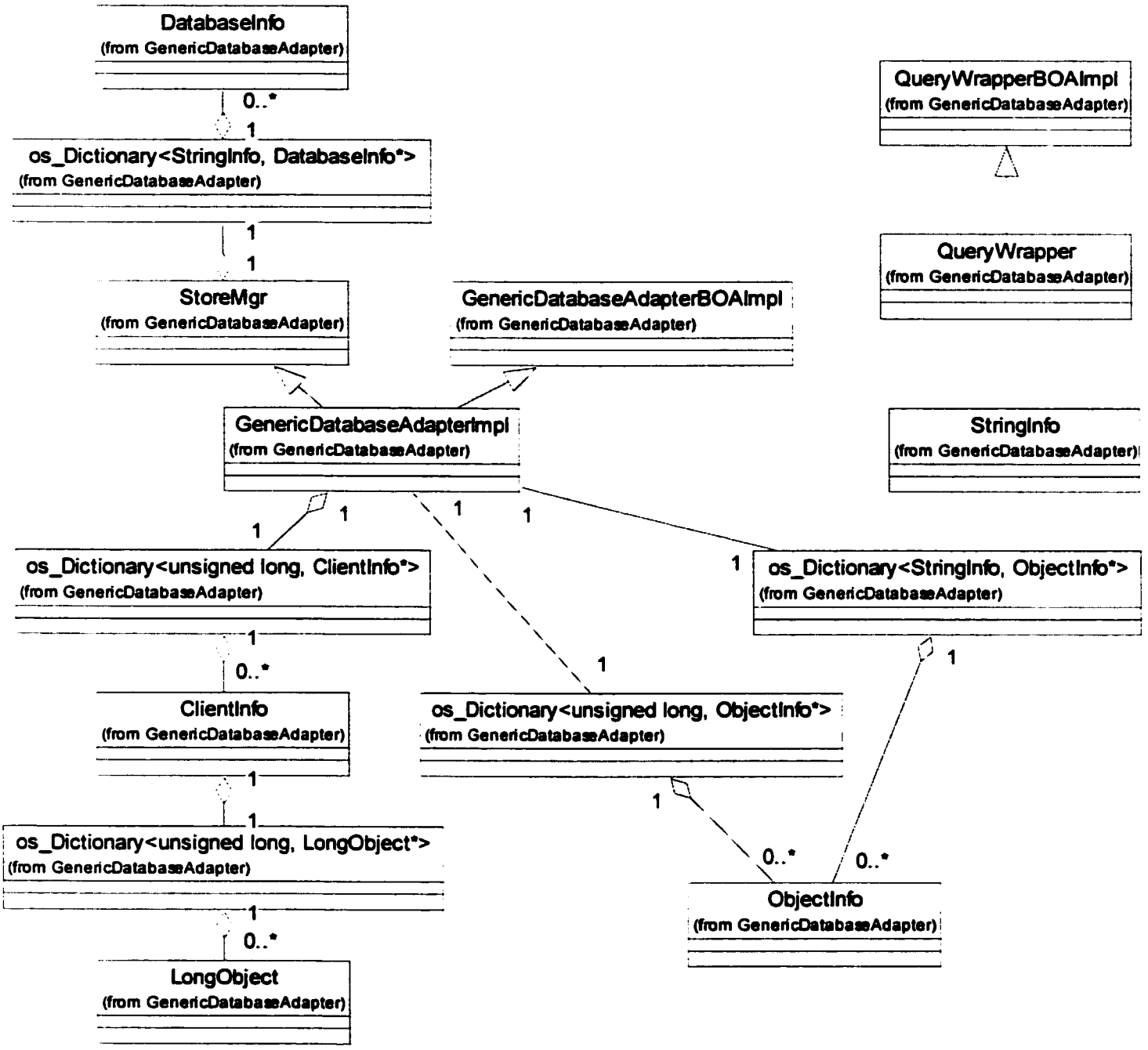

Figure 6 - 21 The Physical Object Model.

6.3 Classes.

This section describes all classes used to implement the Generic Database Adapter, together with all the services and data members corresponding to each class.

6.3.1 StringInfo

This class represents a string and is used as a key in the database dictionary that store pointers to ObjectInfo objects. Each key contains the string corresponding to the persistent reference of a specific object stored in an Object Store database.

6.3.1.1 Attributes

6.3.1.1.1 char* m_pszData

This data member represents a C++ string.

6.3.1.2 Methods

6.3.1.2.1 StringInfo(const char*)

This method creates a new instance of this class and initializes the object to the string provided as the argument.

6.3.1.2.2 ~StringInfo( )

This method destroys an object belonging to this class.

6.3.1.2.3 StringInfo(const StringInfo&)

This method represents the copy constructor corresponding to this class.

6.3.1.2.4 static int Rank(const void*, const void*)

This method is needed if the objects of this class are used as keys in a hash dictionary.

6.3.1.2.5 static os_unsigned_int32 Hash(const void*)

This method is needed if the objects of this class are used as keys in a hash dictionary.

6.3.1.2.6 static os_typespec* get_os_typespec()

This method is needed in order to store objects of this class persistently.

6.3.2 LongObject

This class represents an unsigned long and is used as values in the transient dictionary that stores information on the objects used by a specific Adapter client during a session. Each value contains the identifier of the object.

6.3.2.1 Attributes

6.3.2.1.1 unsigned long mValue

This data member represents a C++ unsigned long.

6.3.2.2 Methods

6.3.2.2.1 LongObject(unsigned long)

This method creates a new instance of this class and initializes the object to the unsigned long value provided as argument.

6.3.2.2.2 ~LongObject( )

This method destroys an object belonging to this class.

6.3.2.2.3 unsigned long getValue( )

This method returns the value contained within the data member.

6.3.2.2.4 void setValue(unsigned long)

This method sets the data member to the value provided as argument.

6.3.2.2.5 static os_typespec* get_os_typespec()

This method is needed in order to store objects of this class persistently.

6.3.3 ClientInfo

This class encapsulates specific information on every Generic Database Adapter Client. Each time a client connects to a specific database, the Generic Database Adapter starts a new session, a new Object Store transaction for that client, and stores the identifier of the transaction inside a ClientInfo object. Also identifiers for all the objects accessed by the client during a session are registered inside the ClientInfo object corresponding to the client.

6.3.3.1 Attributes

6.3.3.1.1 os_transaction* m_pTransactionId

This data member represents the identifier of the Object Store transaction.

6.3.3.1.2 os_Dictionary<unsigned long, LongObject*>* m_ListOfObjectIds

This data member represents a dictionary of object identifiers and object names used for keeping track of all the objects used by the Generic Database Adapter Client during a session.

6.3.3.2 Methods

6.3.3.2.1 ClientInfo(os_transaction*, os_Dictionary<unsigned long, LongObject*>*)

This method creates a new instance of this class by providing an identifier for the Object Store transaction started for the session and a pointer to a transient dictionary.

6.3.3.2.2 ~ClientInfo( )

This method destroys an object belonging to this class.

6.3.3.2.3 os_transaction* GetOSTransaction( )

This method returns the identifier of the Object Store transaction.

6.3.3.2.4 os_Dictionary<unsigned long, LongObject*>* GetListOfObjectIds();

This method returns a pointer to the list of object identifiers used during the current session.

6.3.3.2.5 static os_typespec* get_os_typespec()

This method is needed in order to store objects of this class persistently.

6.3.3.2.6 void SetOSTransaction(os_transaction*)

This method sets the Object Store transaction identifier to the value provided as an argument.

6.3.3.2.7 void AddObjectId(unsigned long, LongObject*)

This method adds another object identifier to the list of used objects.

6.3.4 ObjectInfo

This class encapsulates specific information on every persistent object created using the Generic Database Adapter for a specific application. Each time a new persistent object is created and stored in a specific Object Store database, the Generic Database Adapter registers information about that object into an ObjectInfo object and stores it persistently into a database dictionary for future use.

6.3.4.1 Attributes

6.3.4.1.1 os_reference_protected m_voidReference

This data member represents the persistent Object Store reference of an objects stored persistently in a database, and is being used for copying the current object to a new one, when a Generic Database Adapter Client accesses the object.

6.3.4.1.2 os_reference_protected m_corbaReference

This data member represents the persistent Object Store reference of an objects stored persistently in a database, and is being used to return a CORBA reference to the Generic Database Adapter Client that requests a specific object.

6.3.4.1.3 unsigned long m_objectName

This data member represents the identifier of self.

6.3.4.2 Methods

6.3.4.2.1 ObjectInfo(unsigned long, os_reference_protected, os_reference_protected)

This method creates a new instance of this class by providing an identifier for the Object Store object being created, an Object Store reference created based on a void* to the object, and another Object Store reference created based on a CORBA::Object*.

6.3.4.2.2 ~ObjectInfo( )

This method destroys an object belonging to this class.

6.3.4.2.3 os_reference_protected GetVoidReference( )

This method returns the reference created based on a void*.

6.3.4.2.4 os_reference_protected GetCorbaReference( )

This method returns the reference created based on a CORBA::Object*.

6.3.4.2.5 static os_typespec* get_os_typespec()

This method is needed in order to store objects of this class persistently.

6.3.4.2.6 unsigned long GetObjectName( )

This method returns the identifier of the object.

6.3.4.2.7 void SetVoidReference(os_reference_protected)

This method sets the void* based Object Store reference to the value provided.

6.3.4.2.8 void SetCorbaReference(os_reference_protected)

This method sets the CORBA::Object* based Object Store reference to the value provided.

6.3.4.2.9 void SetObjectName(unsigned long objectName)

This method sets the identifier of the object to the value provided.

6.3.5 DatabaseInfo

This class encapsulates specific information on every Object Store database accessed using the Generic Database Adapter during a specific run for a specific application. Each time an Object Store database is accessed, the Generic Database Adapter registers information about that database into a Database Info object and stores it into a transient dictionary for future use.

6.3.5.1 Attributes

6.3.5.1.1 os_database* m_DatabaseReference

This data member represents the identifier of an Object Store database, that is accessed by different Generic Database Adapter Clients.

6.3.5.2 Methods

6.3.5.2.1 DatabaseInfo(os_database*)

This method creates a new instance of this class by providing an identifier for the Object Store database being accessed.

6.3.5.2.2 ~DatabaseInfo( )

This method destroys an object belonging to this class.

6.3.5.2.3 os_database* GetDatabaseReference( )

This method returns the database identifier.

6.3.5.2.4 static os_typespec* get_os_typespec()

This method is needed in order to store objects of this class persistently.

6.3.5.2.5 void SetDatabaseReference(os_database*)

This method sets the database identifiers to the value provided as an argument.

6.3.6 QueryWrapperImpl

This class encapsulates specific information on the result of a query executed against a specific object collection contained within a specific database. The result of each query is wrapped in an instance object of this class. The user of an object of this class can browse through the collection of objects representing the result of the query by using the public services provided by this class.

6.3.6.1 Attributes

6.3.6.1.1 os_collection* mpCollection

This data member represents the identifier of an Object Store collection which contains the objects selected as a result of executing a query.

6.3.6.1.2 os_cursor* mpCursor

This data member represents the identifier of an Object Store cursor that is used internally by the class for navigating through the result of the query.

6.3.6.2 Methods

6.3.6.2.1 QueryWrapperImpl(os_collection*)

This method creates a new instance of this class by providing an identifier for the Object Store collection used for storing the result of the query.

6.3.6.2.2 ~QueryWrapperImpl( )

This method destroys an object belonging to this class.

6.3.6.2.3 CORBA::Object_ptr GetFirst( )

This method returns a CORBA reference to the first element of the collection representing the result of the query.

6.3.6.2.4 CORBA::Object_ptr GetNext( )

This method returns a CORBA reference to the next element of the collection representing the result of the query.

6.3.6.2.5 CORBA::Object_ptr GetLast( )

This method returns a CORBA reference to the last element of the collection representing the result of the query.

6.3.7 StoreMgr

This class encapsulates the Object Store API used by the Generic Database Adapter for opening, closing database, creating, retrieving database root objects, and for storing database identifiers for all the databases used by the Generic Database Adapter Clients.

6.3.7.1 Attributes

6.3.7.1.1 os_Dictionary<StringInfo, DatabaseInfo*>* m_databaseInfoDictionary

This data member represents the identifier of an Object Store dictionary that contains identifiers for all the databases used by the Generic Database Adapter Clients.

6.3.7.2 Methods

6.3.7.2.1 StoreMgr( )

This method creates a new instance of this class.

6.3.7.2.2 StoreMgr( )

This method destroys an object belonging to this class.

6.3.7.2.3 void OpenDB(const char* pszDBLogicalName)

This method opens the Object Store database having the name provided as an argument, and registers the name of the database together with the database identifier as part of the database collection information contained within the object.

6.3.7.2.4 void CloseDB(const char* pszDBLogicalName)

This method closes the Object Store database having the name provided as an argument, and de-registers the name of the database together with the database identifier from the database collection information contained within the object.

6.3.7.2.5 void CloseAllDatabases()

This method closes all the databases opened by the Generic Database Adapter clients.

6.3.7.2.6 void NameObject(void* pObj, const char * pszName, const char* pszDBLogicalName)

This method creates a database root in the database identified by the database name provided as an argument and associates the root with an identifier of an object or collection of objects provided as an argument.

6.3.7.2.7 void* GetRootObject(const char * pszName, const char* pszDBLogicalName)

This method retrieves the identifier of a database root located in a database identified by the database name provided as an argument and having the name identified by a string provided as an argument.

6.3.8 GenericDatabaseAdapterImpl

This class is a C++ singleton entity and encapsulates the entire functionality of the Generic Database Adapter. Thus only one object of this type will be created per each instance of the Generic Database Adapter. All the methods of this class are thread safe thus the adapter can be used concurrently by different clients. This class exports a set of API services that can be used by any users of the Generic Database Adapter such as: create new persistent Object Store objects, remove existing persistent objects, retrieve existing persistent Object Store objects, create and remove persistent database collections, query existing collections of persistent objects.

6.3.8.1 Attributes

6.3.8.1.1 os_Dictionary<unsigned long, ClientInfo*>* m_pClientInfoDictionary

This data member represents the identifier of an Object Store dictionary that contains identifiers for all the clients that are connected to the Generic Database Adapter at a moment of time.

6.3.8.1.2 static ObjectManagerUnivImpl* m_pObjectManagerUnivImpl

This data member represents the identifier of a static pointer to the singleton object of self.

6.3.8.1.3 static pthread_mutex_t m_Mutex

This data member represents the identifier of a mutex used to ensure that the Generic Database Adapter can be accessed concurrently by different clients at the same time.

6.3.8.2 Methods

6.3.8.2.1 GenericDatabaseAdapterImpl (const char* const objectName)

This method creates a new instance of this class.

6.3.8.2.2 ~GenericDatabaseAdapterImpl ( )

This method destroys an object belonging to this class.

6.3.8.2.3 static GenericDatabaseAdapterImpl* Access( )

This static method allows access to the singleton object by creating a new instance of the class and returning a pointer to it if the singleton object was not already created, or by returning a pointer to the singleton object if the object was already created.

6.3.8.2.4 static void CreateManagerOnce( )

This method ensures that an instance of this class is created only once per any instance of the Generic Database Adapter.

6.3.8.2.5 static void CreateMutexOnce( )

This method ensures that the mutex used by this class is initialized only once.

6.3.8.2.6 boolean InitializeDatabase()

This method initializes the internal list used by the Generic Database Adapter to keep track of all the clients using the adapter at any moment in time, and an internal database used to keep track of all the persistent objects created by the adapter until now. This database contains two different dictionaries referring the same set of ObjectInfo objects. The first dictionary uses the name of the objects as the key, and the other one uses the stringified equivalent of the Object Store reference as the key. The first dictionary is used when the clients try to retrieve existing persistent objects, and the second one is used when the query service is invoked.

6.3.8.2.7 CORBA::Object* ManageObject(os_typespec* objectTypespec, unsigned long objectName, long clientId, const char* collectionName, const char* databaseName, const os_class_type& class_type_of_object, void*& pVoidReference);

This method creates a new database object and returns a pointer to self. If the object already exists the method retrieves creates a new persistent object, copies the content of the old one into the new one, and returns a pointer to self. The copy of the existing object into the new one is done automatically by retrieving metadata about the internal structure of the object from the database schema and by copying the values corresponding to the identified data members from one object to the other. After the creation of the object

specific object information is stored into the ObjectInfo database for future references. There are two different pointers that get returned after invoking this method:

- The pointer to the newly created object gets converted to a CORBA::Object* and is returned to the requester.
- The pointer to the newly created object gets converted to a void* and gets stored into the ObjectInfo database to be used for future references (for copying the content of the existing object into a new one when the GetObject() method is invoked). The Object Store reference corresponding to this pointer is used as a key in the second dictionary used as part of the ObjectInfo database.

6.3.8.2.8 os_collection& ExecuteQuery(os_collection* pCollection, char* queryString, char* objectPointerType)

This method executes a specific query on a specified collection of objects, having the description of the type passed as a string.

6.3.8.2.9 void* MigrateObject(void* pVoidObject, const char* objectType, const char* collectionName, const char* databaseName, long clientId)

This method is used for the implementation of the query service. It migrates a specific object from the transient collection representing the result of a query, into a collection which is going to be wrapped into a QueryWrapper object whose reference will be returned to the client. Using the Object Store reference corresponding to the pVoidObject pointer passed in as an argument, the information about the corresponding object is retrieved from the ObjectInfo database and the ManageObject() method described above is invoked.

6.3.8.2.10 CORBA::Boolean ConnectToDatabase(const char* databaseName, CORBA::Long clientId)

This method has to be invoked by any Generic Database Adapter Clients before starting accessing objects stored in the database identified by the database name passed as an argument. The method opens the database and checks if the client is already registered with the adapter. If the client is not registered with the adapter, it registers the client, opens an ObjectStore transaction and stores the identifier of the transaction in the together with an empty list of objects that are to be accessed by the client, into a

ClientInfo object, and inserts the object into the ClientInfo transient database. If the client is already registered with the adapter, the method only opens the database identified by the database name passed as an argument.

6.3.8.2.11 CORBA::Boolean DisconnectFromAdapter(CORBA::Long clientId)

This method has to be invoked by any Generic Database Adapter Clients when the client wants to terminate the interaction with the Generic Database Adapter. The method retrieves the list of objects accessed by the client, nulls out two transient pointers contained within each object, commits the Object Store transaction started when the client connected to the database, and removes and deletes the ClientInfo object corresponding to this client from the ClientInfo transient database..

6.3.8.2.12 CORBA::Boolean CreateCollection(const char* collectionName, const char* databaseName)

This method creates a new database persistent collection in a specific database whose name is passed as an argument. The name of the collection will be set to the value passed as an argument.

6.3.8.2.13 CORBA::Boolean RemoveCollection(const char* collectionName, const char* databaseName)

This method removes an existing database persistent collection from a specific database whose name is passed as an argument. The name of the collection that is to be removed is passed as an argument. All persistent objects contained within the specified collection will be deleted completely from the database.

6.3.8.2.14 CORBA::Boolean CreateObject(const char* objectType, CORBA::Long objectName, CORBA::Long clientId, const char* collectionName, const char* databaseName)

This method creates a new database persistent object in a specific database and collection whose names get passed in as arguments. A string representing the type of the object that is to be created is passed also as an argument. In order to create a completely generic

database adapter the actual creation of the object is included into a dynamic library which is to be accessed by the adapter at run time. The string identifying the type is also used for retrieving Object Store schema information required for creating persistent objects. After determining the metadata corresponding to the object type, the ManageObject() method is invoked.

6.3.8.2.15 CORBA::Object_ptr GetObject(const char* objectType, CORBA::Long objectName, CORBA::Long clientId, const char* collectionName, const char* databaseName

This method verifies if an object with the name identified by the value passed as an argument exists in the database and collection whose names are passed as arguments. If the object exists, the ManageObject() method is invoked, otherwise a null pointer is returned.

6.3.8.2.16 CORBA::Boolean RemoveObject(CORBA::Long objectName, const char* collectionName, const char* databaseName)

This method verifies if an object with the name identified by the value passed as an argument exists in the database and collection whose names are passed as arguments. If the object exists, the corresponding ObjectInfo object is being removed and deleted from the ObjectInfo database, and the object itself gets deleted and removed from the database and collection specified.

6.3.8.2.17 QueryWrapper_ptr QueryCollection (const char * collectionName, const char* databaseName, CORBA::Long clientId, const char * sqlString, const char* objectType, const char* objectPointerType)

This method executes a query on the database and collection whose names are passed as arguments, using the query string passed in as an argument. After the query is executed it traverses the result of the query, calls MigrateObject() on each object, and inserts the returned object into a transient collections. A new QueryWrapper object is created and initialized using the transient collection afore created, and a pointer to self is returned to the requester.

7 Evaluation of the Generic Database Adapter

Two different sets of tests were performed in order to demonstrate the functional and performance capabilities of the Generic Database Adapter designed and developed as part of this thesis. The results and the interpretation of these results are presented as part of this chapter. The first set of tests is the functional tests, which verify the operation of the Generic Database Adapter and the second set of tests is the performance tests which determine the effect of the Generic Database Adapter on performing database related operations. In order to perform all the test sets described above, a simple Account class has been chosen as the source of objects to be managed by the Generic Database Adapter. The structure of the Account class is very simple, containing only an account number and a balance, as data members. A corresponding IDL interface is defined and implemented in order to provide a means to invoke operations on the corresponding Account objects.

7.1 Functional Testing of the Generic Database Adapter

The set of functional tests relies on the fact that an Object Store server is installed on each of the Unix machines involved in the tests and that they are interconnected in a network (access is possible from one of them to all the others). The following set of test case scenarios have been identified as suitable for the functional tests:

- Test Case 1: store Account objects in different collections, which are created in different databases on different machines.
- Test Case 2: retrieve and modify the Account objects created as part of the previous test case.
- Test Case 3: execute queries on the collection of objects created as part of the previous test cases, and modify the objects which are part of the result of the queries.
- Test Case 4: remove objects from different collections of objects and verify that the object doesn't exist anymore.

The process for verifying correct operation for each test case was as follows.

1. Create two databases on each of the Unix machines involved in this test. Create one collection, and then create objects, in each of the databases created above. List all databases created as Unix files in order to prove their creation. Each object created is retrieved and the account balance corresponding to each of them is retrieved and displayed.
2. All objects created as part of the previous test case are retrieved, the balance corresponding to each account is displayed, for some of them the balance is increased with a specific value, the balance corresponding to each object is displayed again proving that the balances have changed.
3. Perform a query on one of the collections using a specific criterion, display the balance corresponding to each of the objects which are part of the result of the query (demonstrate that the retrieval of first, next, last works properly), modify all objects that are part of the result of each query by increasing the balance with a specific value, and display the balance corresponding to each of the objects again.
4. Remove objects from one of the collections created, then try to retrieve those objects to prove that they do not exist anymore.

Before starting performing the functional test cases, the application schema has to be generated and linked into the executable corresponding to the Generic Database Adapter. Thus, all the classes whose objects are to be stored persistently, have to be included in the application schema, which in turn will be used by the Object Store engine to generate the database schema as part of each of the databases being created by the Generic Database Adapter.

The IDL interface for the Account has to be defined, compiled, and implemented in order to have the possibility to invoke requests on objects corresponding to this interface.

The dynamic library containing the code that facilitates the creation of persistent objects, has to be updated for the classes whose objects are to be stored persistently, and re-compiled. The Generic Database Adapter code requires no changes and it does not have to be re-compiled.

All the test cases described in this chapter have been validated successfully by checking the information displayed on the screen by the Generic Database Adapter clients, and by checking the content of the object databases created and changed during the tests. The results obtained at the end of each test case are presented in the following diagrams.

```
Wed May 30 19:26:49 2001
Functional test started
Connect to all databases.
Connect to database: arnie:/Schema/databases/Test1.db
Connect to database: arnie:/Schema/databases/Test2.db
Connect to database: hyperion:/Schema/databases/Test3.db
Connect to database: hyperion:/Schema/databases/Test4.db
Create collections.
Collection DB1Root1 created in database:
arnie:/Schema/databases/Test1.db
Collection DB2Root1 created in database:
arnie:/Schema/databases/Test2.db
Collection DB3Root1 created in database:
hyperion:/Schema/databases/Test3.db
Collection DB4Root1 created in database:
hyperion:/Schema/databases/Test4.db
Create objects.
Object 10 created in collection DB1Root1 and database:
arnie:/Schema/databases/Test1.db
Object 11 created in collection DB1Root1 and database:
arnie:/Schema/databases/Test1.db
Object 12 created in collection DB1Root1 and database:
arnie:/Schema/databases/Test1.db
Object 13 created in collection DB1Root1 and database:
arnie:/Schema/databases/Test1.db
Object 14 created in collection DB2Root1 and database:
arnie:/Schema/databases/Test2.db
Object 15 created in collection DB3Root1 and database:
hyperion:/Schema/databases/Test3.db
Object 16 created in collection DB3Root1 and database:
hyperion:/Schema/databases/Test3.db
Object 17 created in collection DB4Root1 and database:
hyperion:/Schema/databases/Test4.db
Object 18 created in collection DB4Root1 and database:
hyperion:/Schema/databases/Test4.db
Object 19 created in collection DB4Root1 and database:
hyperion:/Schema/databases/Test4.db
Object 20 created in collection DB4Root1 and database:
hyperion:/Schema/databases/Test4.db
```

Figure 7- 1 Functional test results - part1.

```
Retrieve all objects.
Retrieve object 11 from database: arnie:/Schema/databases/Test1.db
Object 11 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 5
The current balance is: 15
The balance is: 15
The amount deposited is: 20
The amount withdrawn is: 5
The current balance is: 30
The balance is: 30
The amount deposited is: 20
The amount withdrawn is: 5
The current balance is: 45
The balance is: 45
The amount deposited is: 20
The amount withdrawn is: 5
The current balance is: 60
Retrieve object 12 from database: arnie:/Schema/databases/Test1.db
Object 12 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 5
The current balance is: 15
Retrieve object 13 from database: arnie:/Schema/databases/Test1.db
Object 13 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 10
Retrieve object 14 from database: arnie:/Schema/databases/Test2.db
Object 14 retrieved.
The balance is: 0
Retrieve object 15 from database: hyperion:/Schema/databases/Test3.db
Object 15 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 10
Retrieve object 16 from database: hyperion:/Schema/databases/Test3.db
Object 16 retrieved.
The balance is: 0
```

Figure 7- 2 Functional test results - part 2.

```
Retrieve object 17 from database: hyperion:/Schema/databases/Test4.db
Object 17 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 10
Retrieve object 18 from database: hyperion:/Schema/databases/Test4.db
Object 18 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 10
Retrieve object 19 from database: hyperion:/Schema/databases/Test4.db
Object 19 retrieved.
The balance is: 0
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 10
Retrieve object 20 from database: hyperion:/Schema/databases/Test4.db
Object 20 retrieved.
The balance is: 0
Disconnect from all databases.
Disconnected from all databases.
Connect to database: arnie:/Schema/databases/Test1.db
Query the collection of objects found in database Test1.db. //
balance > 0
Retrieve the first object from the query result.
The account number is: 11
The balance is: 60
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 70
Retrieve the next object from the query result.
The account number is: 12
The balance is: 15
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 25
Retrieve the last object from the query result.
The account number is: 13
The balance is: 10
The amount deposited is: 20
The amount withdrawn is: 10
The current balance is: 20
```

Figure 7- 3 Functional test results - part 3.

```
Remove object 20 from database Test4.db.
Object removed from database hyperion:/Schema/databases/Test4.db
Retrieve object 20 from database hyperion:/Schema/databases/Test4.db
There is no object 20 in this database.

Disconnect from all databases.
Disconnected from all databases.

Wed May 30 19:27:06 2001
Time: 17049793 us


Test completed!
```

Figure 7- 4 Functional test results - part 4.

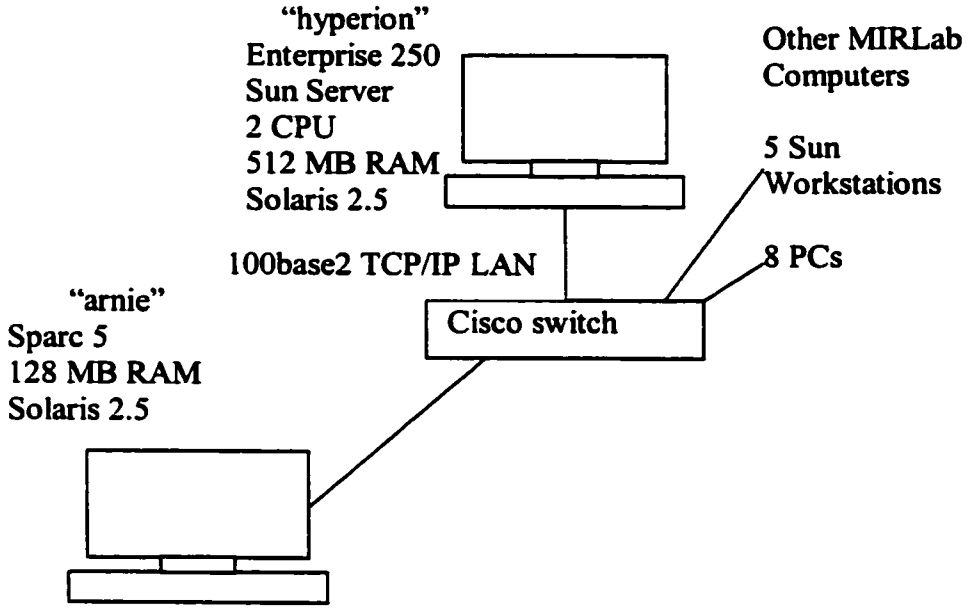

Figure 7- 5 Generic Database Adapter Test setup in the MIR laboratory.

7.2 Performance Testing of the Generic Database Adapter.

In order to determine the performance of the Generic Database Adapter two different sets of test cases were performed and the results obtained were compared. The first set of test cases chosen to determine the performance of the Generic Database Adapter is:

Using the Generic Database Adapter the following operations were performed and the amount of time needed for each independent operation was determined:

- Create 100 accounts as part of the same collection in the same database
- Retrieve the 100 accounts created as part of the previous test case
- Increase the balances of the first 10 accounts with 10 dollars, of the next 10 accounts with 20 dollars, etc. and the balance of the last 10 accounts with 100 dollars
- Query all accounts having the balance equal to 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
- Query all accounts having the balance less than 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
- Remove all the accounts created as part of this test

The second set of test cases is composed of the same test cases as the set described above, with the difference that the tests were performed using direct access to database objects through the Object Store APIs called from another C++ CORBA server.

Further pairs of graphs are presented in order to have the possibility to compare the results of the two sets of test cases and present the final conclusions.

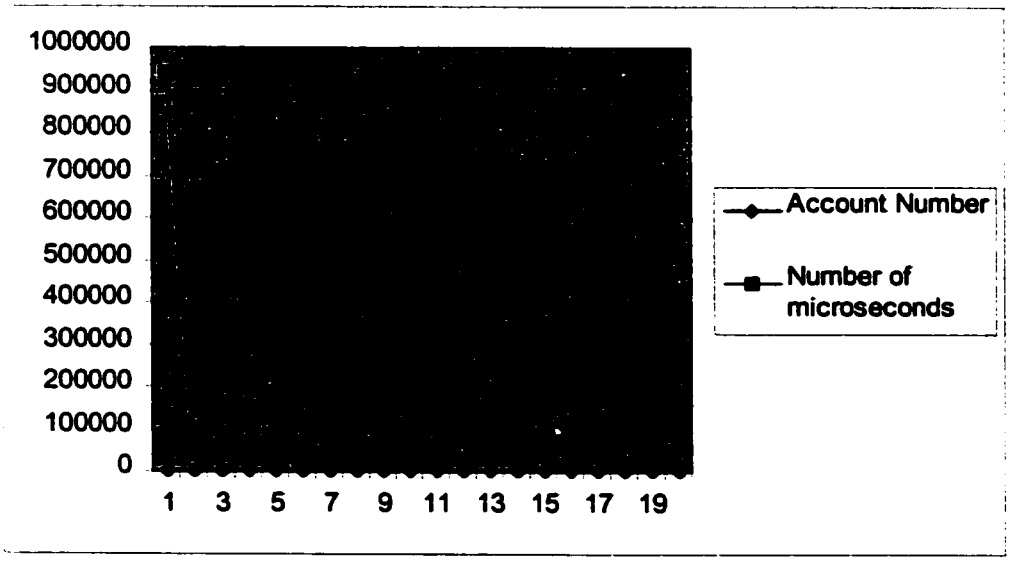

Figure 7- 6 Time required to create accounts using the GDA.

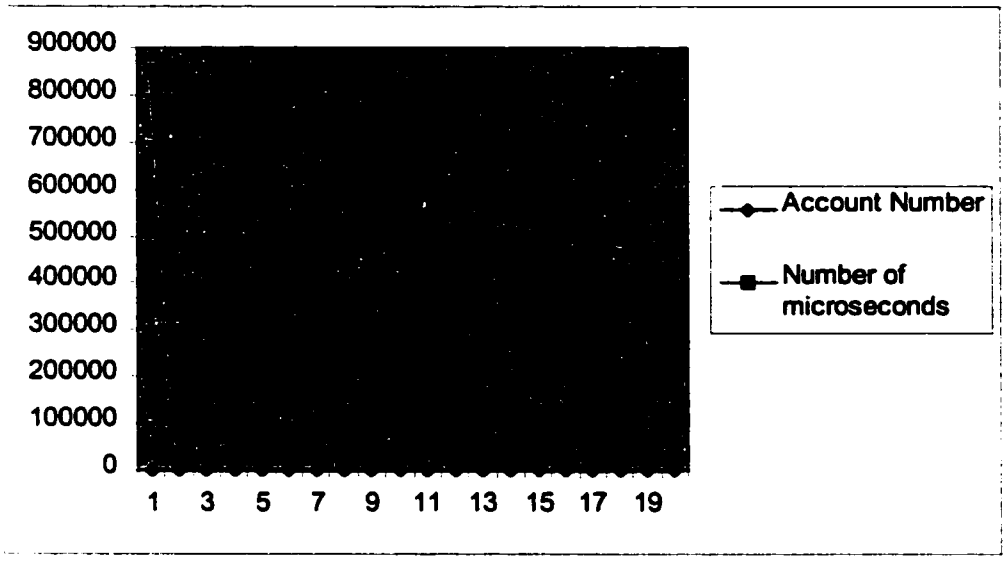

Figure 7- 7 Time required to create accounts without the GDA.

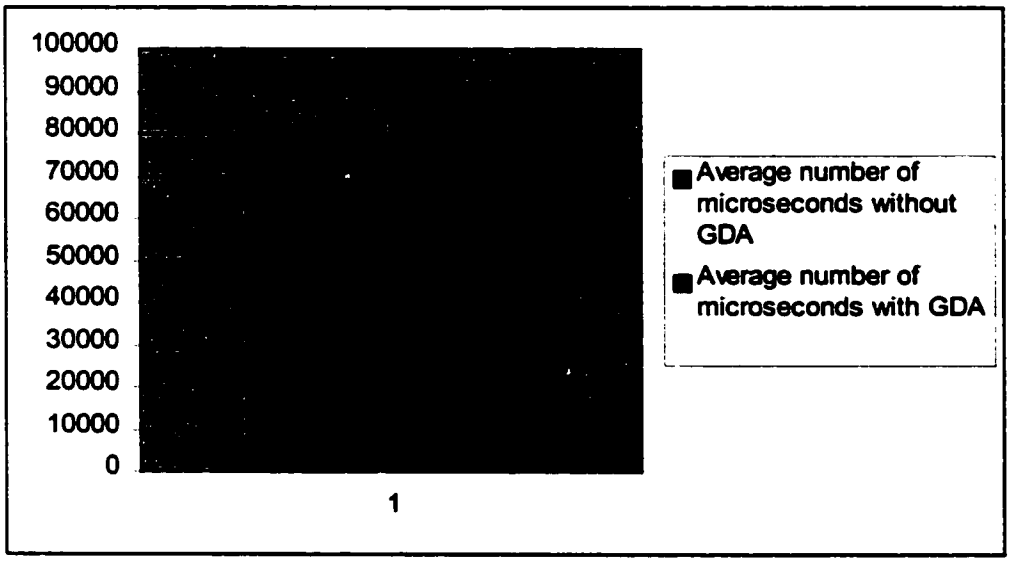

Figure 7- 8 Average time required for creation with and without the GDA.

As it results from the graphs above the amount of time required for the creation of an Account object is greater if the GDA is used because of the extra work the adapter has to do for registering the references to the new objects.

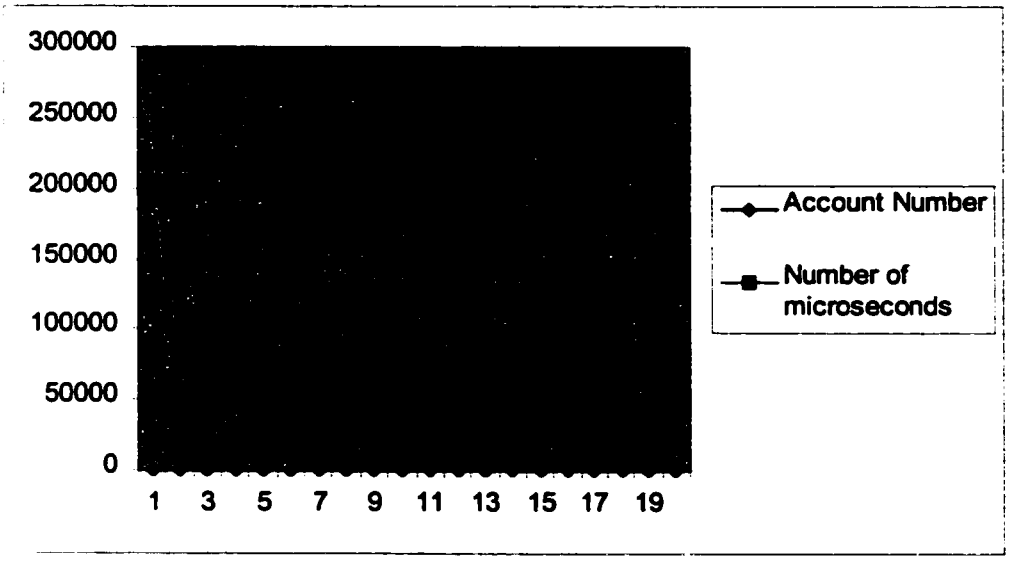

Figure 7- 9 Time required to retrieve accounts with GDA.

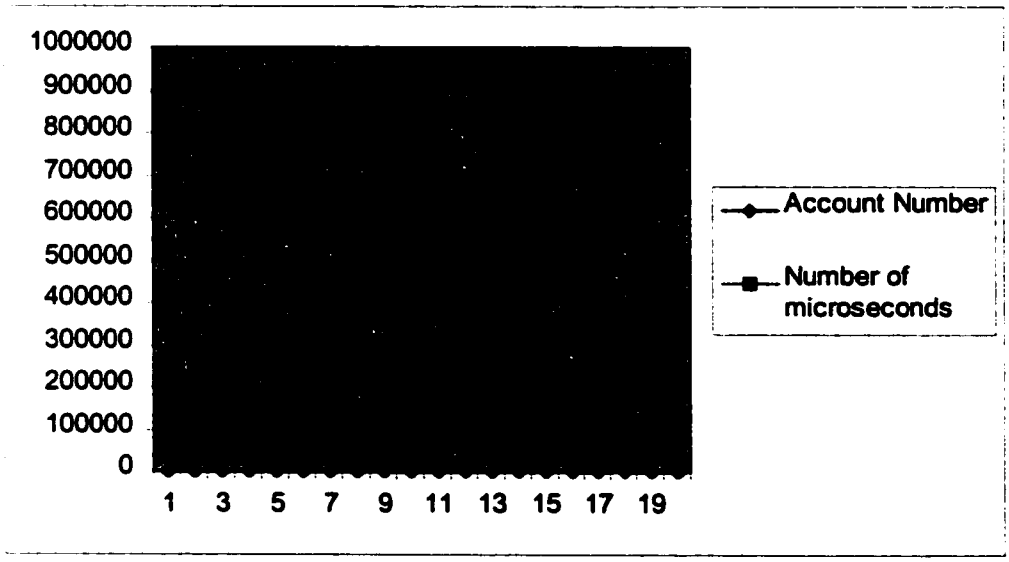

Figure 7- 10 Time required to retrieve accounts without GDA.

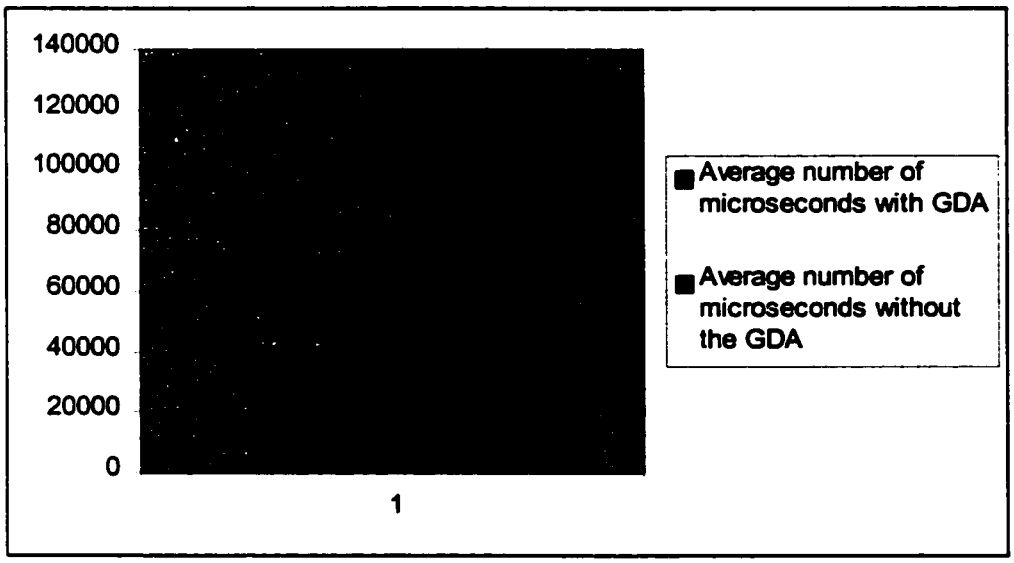

Figure 7- 11 Average time required for retrieval of accounts with and without GDA.

It results that a greater amount of time is required for the retrieval of Account objects without the use of the GDA because the user has to navigate from the database point of entry through the collection containing the objects to find a specific object. If the GDA is used, references to all objects handled by the adapter are already stored and ready to be used.

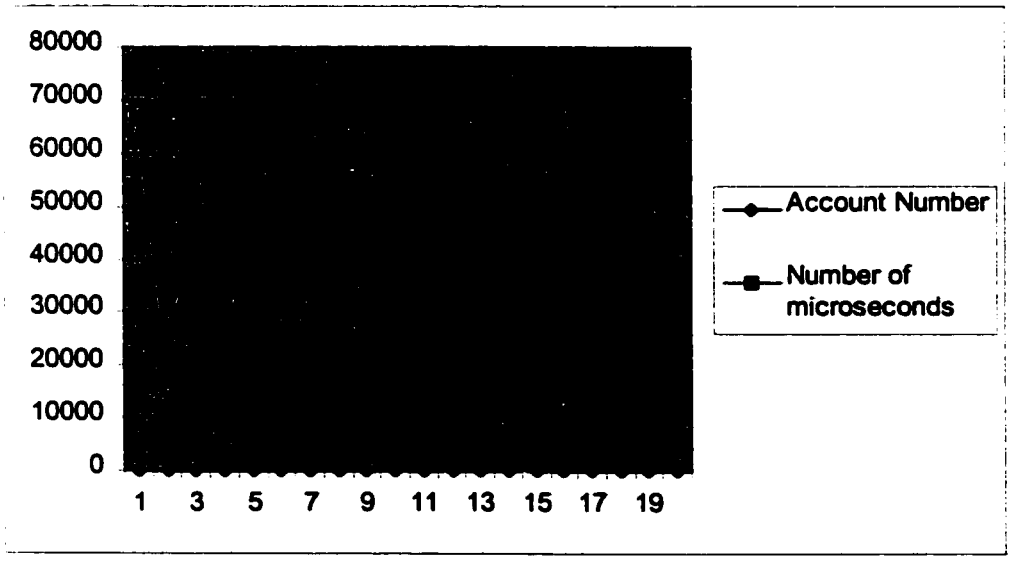

Figure 7- 12 Time required to remove accounts with the use of GDA.

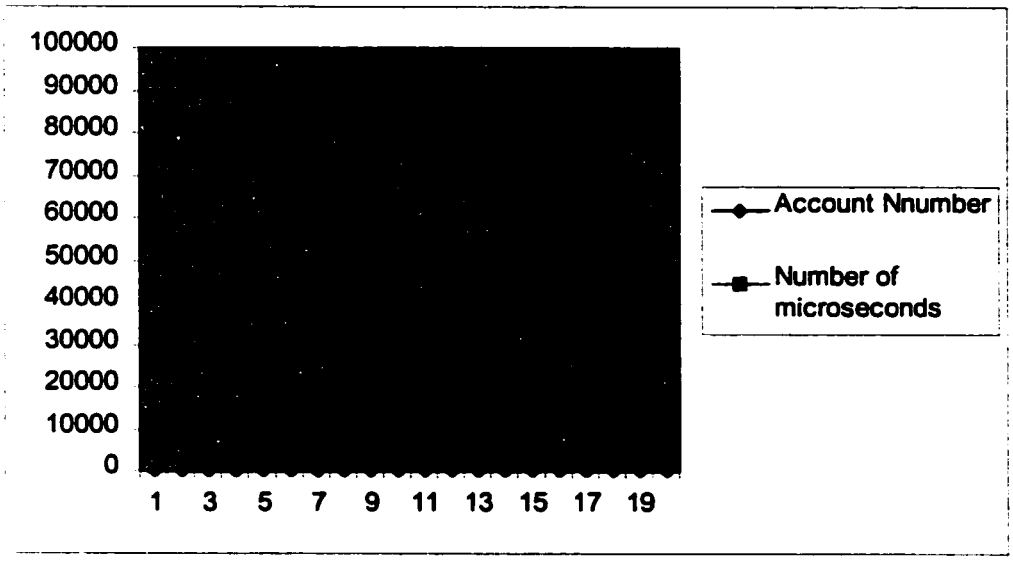

Figure 7- 13 Time required to remove accounts without the use of GDA.

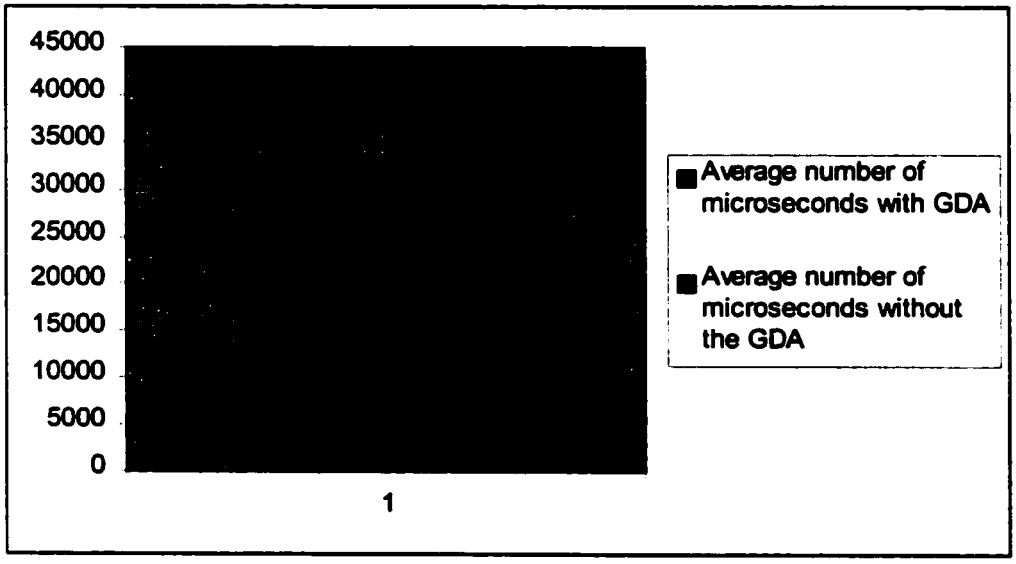

Figure 7- 14 Average time required to remove accounts with and without the GDA.

It results that a greater amount of time is required for the removal of Account objects without the use of the GDA because the user has to navigate from the database point of entry through the collection containing the objects to find a specific object. If the GDA is

used, references to all objects handled by the adapter are already stored and ready to be used.

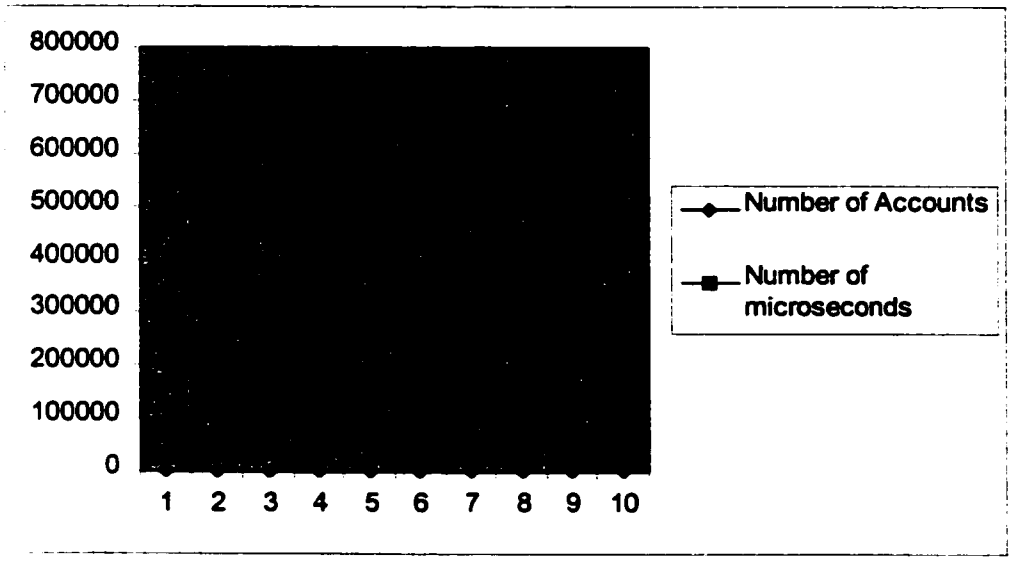

Figure 7- 15 Time required to query groups of 10 accounts with GDA.

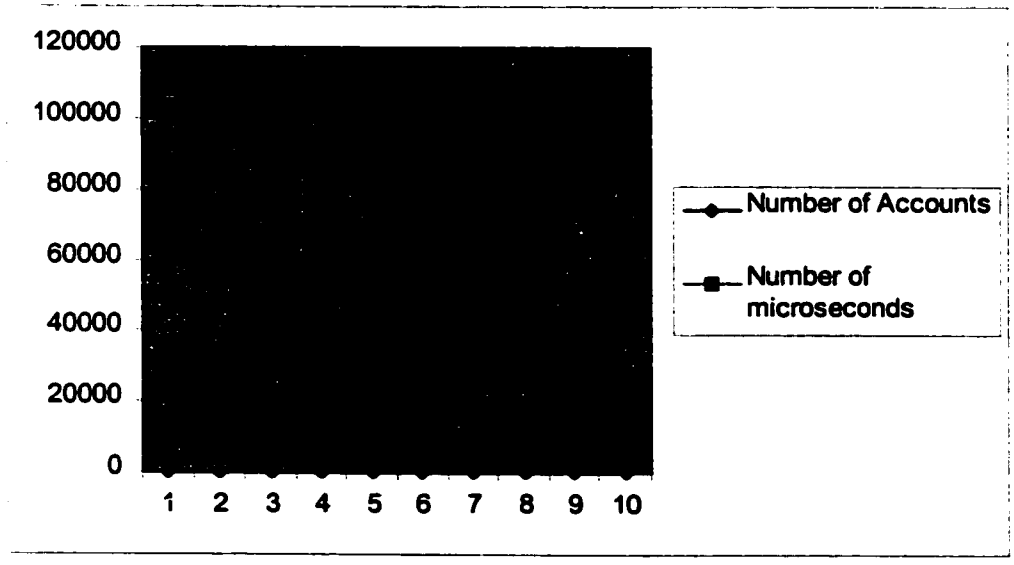

Figure 7- 16 Time required to query groups of 10 accounts without GDA.

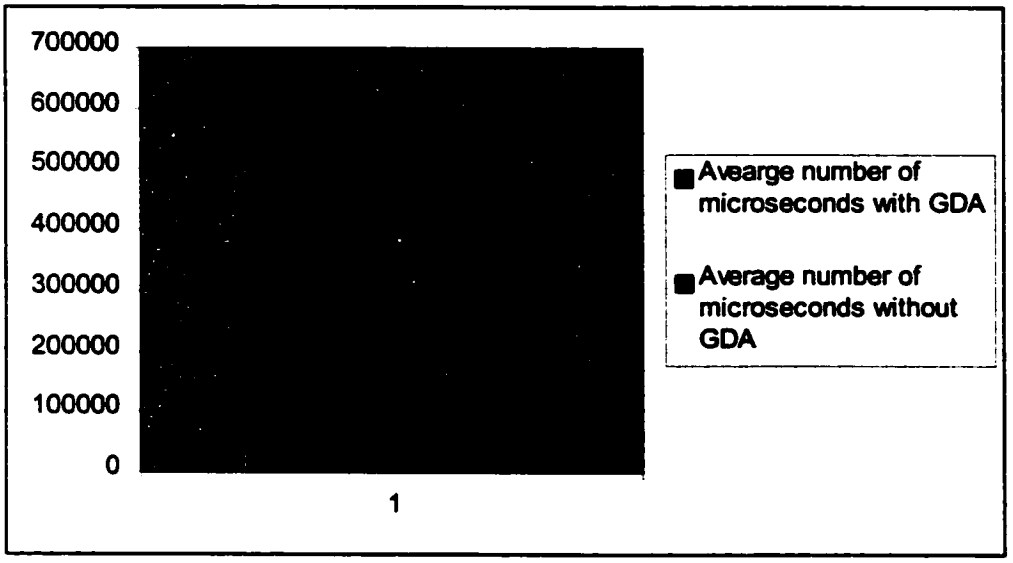

Figure 7- 17 Average time required to query groups of 10 accounts with and without GDA.

It results that a greater amount of time is required for querying Account objects with the use of the GDA, because the GDA has to do extra work in managing the result of the query for a specific GDA client. If the GDA is not used a query operation is taking less time because the system does not manage the objects accessed by a specific client.

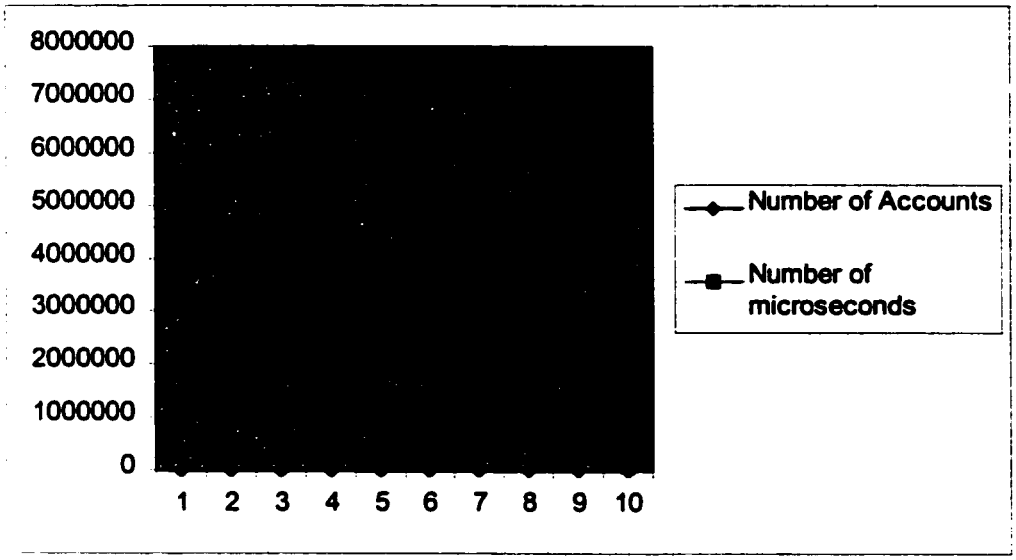

Figure 7- 18 Time required to query accounts in groups increasing by 10 accounts with GDA.

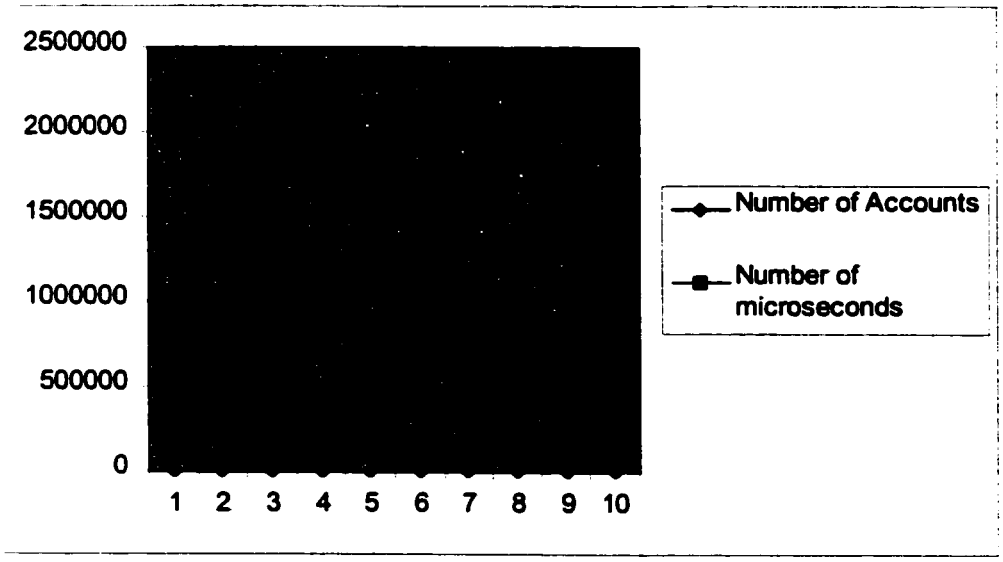

Figure 7- 19 Time required to query accounts in groups increasing by 10 accounts without GDA.

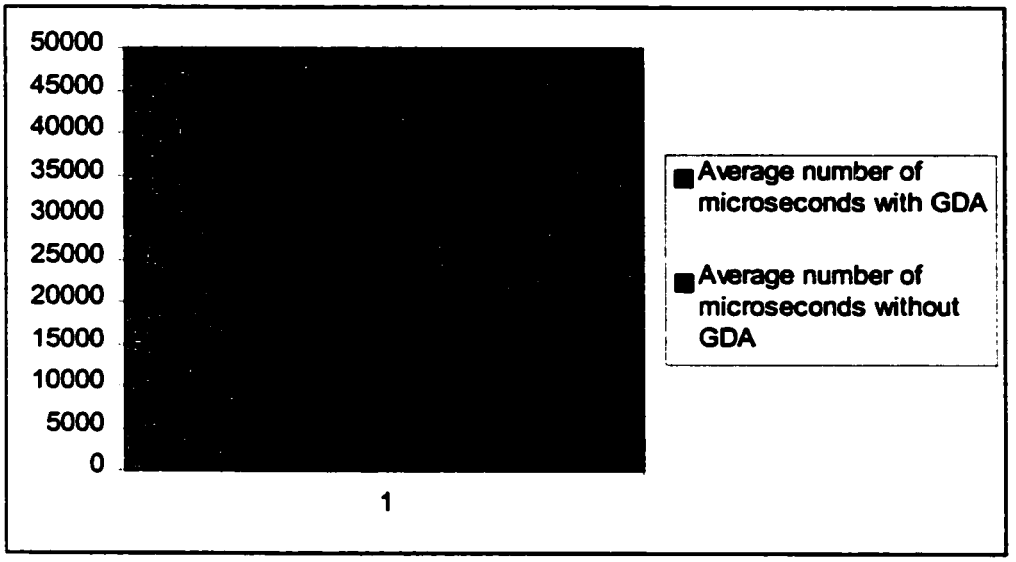

Figure 7- 20 Average time required to query accounts in groups increasing by 10 accounts with and without GDA.

It results that a greater amount of time is required for querying Account objects with the use of the GDA, because the GDA has to do extra work in managing the result of the query for a specific GDA client. If the GDA is not used a query operation is taking less time because the system does not manage the objects accessed by a specific client. It is noticed that the same performance is being obtained when a bigger number of objects is queried.

It results that the use of the GDA increases amount of time required for creating and querying (a small number of objects) operations, but decreases the time required for retrieving and removing operations. Taking into consideration all the other advantages offered by the GDA it results a definite advantage in using the Generic Database Adapter.

8 Conclusions and Future Research

Distributed computing applications using distributed databases (RDBMS or OODBMS) have been playing an important role in all segments of the industry. Both, the existing standard for distributed applications (Common Object Request Broker Architecture defined by the Object Management Group) and the database technologies are evolving rapidly in order to accommodate the ongoing demands of the economy. Many services and facilities are defined for CORBA to aid in the design and development of large, enterprise-scaled applications. At the same time new features are added to the existing DBMSs in order to improve their performance and versatility. However, no existing CORBA mechanism is specified to support a generic adaptation to relational or object-oriented databases. The need to incrementally modify the functionality and implementation of large distributed applications to add new persistent information, or change existing persistent information without having to change in any way the database programming interface used to achieve this, has led to the research in this thesis.

8.1 Concepts Addressed in this Thesis

The concept of adding new information to persistent storage or changing existing one, in a multi-tier distributed CORBA-based application, without the need of modifying the way in which the persistent storage is accessed, has led to the idea of having a Generic Database Adapter (GDA). The concept of generic database adaptability is introduced in this thesis as an extension to the already existing solutions for particular database adapters. The result is the novel solution of a Generic Database Adapter that can manage the access to different distributed databases without the need to change its interface, functionality, and implementation. Thus, the use of a Generic Database Adapter would facilitate the access to different types of databases without having to change the implementation or functionality of the adapter when new information has to be added to existing or new databases of the same types or different types.

This thesis presents a prototype solution of a Generic Database Adapter for Object Store. The approach provides a means of managing the creation, retrieval, removal, grouping, querying of objects whose states have to be stored persistently, or of objects whose state is already stored persistently. Different solutions for implementing a Generic Database Adapter were presented, but only one solution is fully developed. Functional and performance tests were performed and the results were presented, in order to demonstrate the utility and accuracy of the adapter.

8.2 Contributions of this Thesis

Overall, this thesis examines the problem of database adapters in a multi-tier CORBA based environment, resulting in the new concept of a generic database adapter. Two different solutions are proposed but only one solution is fully implemented and presented. The current solution was developed only for object-oriented databases, specifically for Object Store.

More specifically, the need for and the problems of generic database adapters in a multi-tier CORBA based environment were addressed. This includes a review of related research in the field of database adapters used for distributed CORBA based systems. A complete description of Relational and Object-Oriented database management systems is presented, together with a full description of the internal mechanisms used for implementing them. The advantages and disadvantages corresponding to each implementation are presented. A new concept in distributed programming, a Generic Database Adapter, is presented to address these requirements. Complete detailed design specifications of the core component are presented using the Unified Modeling Language and Rational Rose. Functional and performance tests are performed using the Generic Database Adapter and results are presented on the impact to performance caused by the use of the generic database adapter.

This thesis made several significant research contributions, which can be summarized as follows:

1. A new concept in database programming, a Generic Database Adapter, is presented.

2. A full description of two solutions for implementing a Generic Database Adapter is provided in chapter 6.

3. A complete implementation solution, together with the detailed design describing this solution, is presented using UML. This solution is fully compliant with the CORBA 2.0 specification and is implemented at the application level, without modification to the ORB or IDL compiler. The Object Store 5.1 is used as the OODBMS. The impact of using a Generic Database Adapter on performance is measured. The performance overhead was found small and acceptable to most applications.

8.3 Future Research

The implementation of a complete Generic Database Adapter, which would manage database information stored either in relational databases or object-oriented databases, is beyond the scope of this thesis. Work in this field should be continued in order to validate the concept of generic database adapters for all types of database management systems. This would include the following.

- Implement the second solution described in this document and compare the performance results against the solution implemented as part of this thesis.
- Increase the Generic Database Adapter's interface, in order to include other services which would be useful for developing large scale multi-tier CORBA based applications.
- Extend the implementation of the existing generic database adapter in order to include the use of Relational Database Management Systems. As part of this

solution investigate how the database schema information can be queried and how this information can be used in order to maintain the generic feature of the database adapter. Investigate also the use of object-relational mapping, where relational DBMSs can be viewed as object-oriented DBMSs.

- Investigate the implementation of the Generic Database Adapter using different programming languages, including Java, and operating systems, including MS-WindowsNT.

9 Bibliography

[1] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology", C++ *Report*, vol. 11, November/December 1997.

[2] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects", C++ *Report*, vol. 12, April 1998.

[3] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter", C++ *Report*, vol. 13, June 1998.

[4] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter", C++ *Report*, vol. 14, September 1998.

[5] J. Klefndfenst, F. Plasfl, and P. Tuma, "Lessons Learned from Implementing the CORBA Persistence Object Service", in Proceedings of OOPSLA'96, (San Jose, CA), ACM, October 1996.

[6] S. Vinoski and M. Henning, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1998.

[7] S. Baker, *CORBA Distributed Objects Using Orbix*, Addison-Wesley Longman, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[9] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[10] T. Quatrani, *Visual Modeling With Rational Rose And UML*, Addison-Wesley Longman, 1998.

[11] D. Ionescu, V. Groza, R. Ocica, V. Cimpu, M. Trifan, M. Cimpu, V. Vieru, "A Distributed Network Management Environment", Proceedings of CONTI'98, The International Conference on Technical Informatics, third edition, Timisoara, Romania, vol. 3, pp 49-60, 1998.

[12] V. Cimpu, D. Ionescu, M. Cimpu, "Dynamic Managed Objects for Network Management", Machine Intelligence Laboratory, University of Ottawa, Canada, 1998.

[13] D. Ionescu, V. Groza, M. Trifan, R. Ocica, C. Lambiri, "Report No: 1 BaseLayer Functional Specification", Machine Intelligence Laboratory, University of Ottawa, Canada, 1997.

[14] D. Ionescu, V. Groza, M. Trifan, R. Ocica, C. Lambiri, "Report No: 2 BaseLayer General Architecture", Machine Intelligence Laboratory, University of Ottawa, Canada, 1997.

[15] D. Ionescu, R. Ocica, "Report No: 1 Interaction Translation", Machine Intelligence Laboratory, University of Ottawa, Canada, 1999.

[16] Derek Shawn Elsaesser, "Metamorphic Objects for Dynamic Reconfiguration of CORBA-based Applications", Master's Thesis, Department of Electrical Engineering, University of Ottawa, Canada, 2000.

[17] "Life Cycle Service Specification", Object Management Group, November 1996.

[18] "Persistent Object Service Specification" Object Management Group, March 1995.

[19] "The Common Object Request Broker: Architecture and Specification (Revision 2.0)", Object Management Group, July 1995.

[20] Lambiri, C., "Temporal Logic Model for Distributed Systems", Master's Thesis, Department of Electrical Engineering, University of Ottawa, Canada, 1995.

[21] Lerner, B., and Habermann, N., "Beyond Schema Evolution to Database Reorganization", in Proceedings of OOPLSA, ECOOP '90, ACM press, pp. 67-88, Reading, MA, USA, 1990.

[22] Mowbray, T., "CORBA Design Patterns", in Proceedings of Patterns '98, Object Management Group, London, U.K., February 1998.

[23] Mowbray, T. and Malveau, R., *CORBA Design Patterns*, John Wiley & Sons, Inc., New York, 1997.

[24] Blaha, M., Premerlani, W., and Shen, H., "Converting OO Models into RDBMS Schema", IEEE Software, pp. 28-39, May 1994.

[25] Orbix Programmer's Guide (Version 2.3), IONA Technologies PLC, Dublin, Ireland, 1997.

[26] Orbix Programmer's Reference (Version 2.3), IONA Technologies PLC, Dublin, Ireland, 1997.

[27] Seigel, J. *CORBA Fundamentals and Programming*, John Wiley & Sons, New York, 1996.

[28] *ObjectStore C++ API User Guide (Release 5.1)*, Object Design, Inc., Burlington, MA, April 1998.

[29] *ObjectStore Advanced C++ API User Guide (Release 5.1)*, Object Design, Inc., Burlington, MA, April 1998.

[30] *ObjectStore API Reference Manual (Release 5.1)*, Object Design, Inc., Burlington, MA, April 1998.

[31] *C++ Programming with Object Store*, Object Design, Inc., San Francisco, CA, February 1997.

[32] *Object Store Database Design*, Object Design, Inc., San Francisco, CA, February 1997.

[33] *Object Store Performance and Tuning*, Object Design, Inc., San Francisco, CA, February 1997.

[34] *The Common Object Request Broker: Architecture and Specifications Revision 2.2.* Object Management Group, 1998.

[35] Shirley, J., Hu, W., and Magid, D., *Guide to Writing DCE Applications, 2nd Edtion*, O'Reilly & Associates, 1994.

[36] Stroustrup, B., C++ Programming Language, Third Edition, Addison-Wesley, 1997.

[37] Stroustrup B., The C++ Programming Language, Addison Wesley, 1991.

[38] Bertino E., Martino L., Object-Oriented Database Systems (Concepts and Architectures), Addison Wesley, 1994.

[39] Lewis B., Berg J.D., A Guide to Multithreaded Programming, Prentice Hall, 1997.

[40] Larman C., Applying UML and Patterns (An Introduction to Object-Oriented Analysis and Design), Prentice Hall, 1997.

[41] Rankins R., Garbus J. R., Solomon D., Mcewan B. W., Sybase SQL Server 11, SAMS Publishing, 1999.

[42] Date C. J., An Introduction to Database Systems, Addison-Wesley, 2000.