# A New Architecture for Integration of CORBA and OODB

Ruey-Kai Sheu, Kai-Chih Liang, Shyan-Ming Yuan, *Member*, *IEEE*, and Win-tsung Lo

**Abstract**—Object-oriented database system (OODB) supports an object-oriented data model with the functionality of persistency and transaction semantics. In order to facilitate the use of OODB, the Object Database Management Group (ODMG) defined a standard for object database management system. On the other hand, the Object Management Group (OMG) defined the Common Object Request Broker Architecture (CORBA), which is an emerging standard of distributed object technology providing the interconnection network between distributed objects. For the sake of matching these two object models, taking the advantages of merging both of them, and building a more sophisticated infrastructure, the integration of CORBA and OODB is currently an urgent and important issue in distributed object systems. Instead of using Object Database Adapter (ODA) suggested by the ODMG, in this paper, we provide a novel way of reusing the Object Transaction Service (OTS) and wrapping techniques to introduce OODB into CORBA automatically. Through our design, CORBA clients or OODB object implementers do not need to learn any knowledge of each other. In addition, error recovery is also provided to guarantee the consistency of object states. The whole task for integrating CORBA and OODB is done transparently by our proposed preprocessor, which plays an important role in solving problems encountered by ORB and OODB vendors easily.

**Index Terms**—OMG, CORBA, ODMG, object-oriented database, integration, transaction.

## 1 Introduction

### 1.1 Data Management Concepts

The need for data management originates from the complexity of using application data. There are three elementary features that we have to pay attention to. They are *data definition*, *data manipulation*, and *data control*. Data definition means how the data, which will be used later, looks. Its major concern is the structure of data. In OODB, data definition means the schema definition of objects. On the other hand, in CORBA [1], [2], it is the definition of object interfaces through Interface Definition Language (IDL). Data manipulation is mainly composed of operations which have something to do with the defined data. These operations include data creation, modification, deletion, and reference. Data control usually means the mechanism that is actually used to store the data for later usage. The semantics of objects cannot be changed no matter what mechanism is used to control the storage of objects.

We focus on the definition and manipulation of data objects because tasks of data control have been done well by OODB storage managers. We must take care of the mapping of data definition in the two systems and maintain the semantics of data manipulation like transaction, concurrent accesses, and recovery. CORBA's OTS is the key component in our architecture to deal with the problems of data manipulations [3], [4], [5], [6]. While integrating CORBA and OODB using OTS, it not only retains the useful OODB transactional functions, but also provides a transactional management environment for distributed business applications.

### 1.2 Motivations

There are three major reasons that motivate the integration of CORBA and OODB. They are the *problems* of object persistency and heterogeneity while using CORBA or OODB individually, the emerged *advantages* for the integration, and the *experience of past research* [8].

#### 1.2.1 The Problems

With growing Internet and object technology, how to make distributed objects convenient to use and manage is the crucial issue for business applications. Persistence Object Service (POS), proposed by OMG, is used to keep critical data of corporations stored in permanent storage [3]. However, there are weaknesses in POS implementation and integration with other key object services [7]. In addition, the unspecified semantics of operations and the unspecified functionality of POM (Persistent Object Manager) in POS specification mismatch OODB transaction semantics. It is not enough to provide a distributed object management environment just through POS. No OODB vendors would rather discard the OODB proprietary transaction functions to match the POS specification than lose the potentials to beat their competitors. On the other hand, although the ODMG attempts to propose a source code portable standard, the most current Object-Oriented Database Management Systems (ODBMS) on the market are still vendor-specific. Providing data access to applications in today's heterogeneous environment is

• R.-K. Sheu, K.-C. Liang, and S.-M. Yuan are with the Department of Computer and Information Science, National Chiao-Tung University, Taiwan, Republic of China.
E-mail: {gis87802, gis85804, smyuan}@cis.nctu.edu.tw.
• W.-t. Lo is with the Department of Computer and Information Science, Tung Hai University, Taiwan, Republic of China.
E-mail: winston@pine.iecs.fcu.edu.tw.

*Manuscript received 15 May 1998; revised 18 Oct. 1998.*
*For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 109205.*

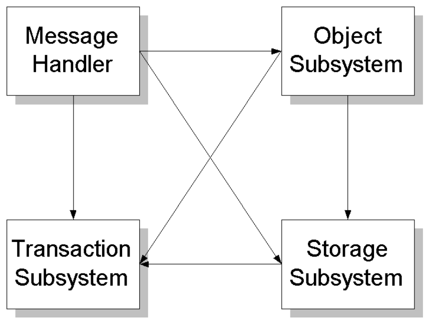

Fig. 1. Functional components of the ORION system.

very complex for software vendors as well as corporate application developers. Therefore, we propose an open architecture to integrate ODMG-compliant OODB with CORBA. It is necessary to extend the OODB architecture to distributed heterogeneous environment in order to promote the functionality and usability of OODB.

### *1.2.2 The Advantages*

CORBA and OODB can be complementary to each other—CORBA serves as the underlying distributed infrastructure for OODB and OODB provides persistency to CORBA. A better model can be constructed by integrating the two parts. From the viewpoint of CORBA, there is no good and straightforward methodology to transparently provide object persistency to meet the requirements of applications. It is an obvious benefit to make object persistency easy by integrating CORBA and OODB. On the other hand, for the reason of the low-level store model proposed by POS, the CORBA client programmers have to take care of the dynamic state and persistent state of server objects. The single-level store model proposed by the integration architecture would alleviate the penalty of the programmer. In the perspective of OODB, CORBA helps OODB across the boundaries of different computer machines, operating systems, and programming languages and it indeed promotes OODB to distributed heterogeneous environment. In addition to distributed deployment of OODB, CORBA allows OODB client objects to be lightweight and communicate with each other in a low-bandwidth network [8].

### *1.2.3 The Experience*

Although problems of using CORBA or OODB and the advantages for the combination of CORBA and OODB motivate the integration, the main idea for integrating CORBA and OODB comes from the experience of the ORION project [9]. This project is in the Advanced Computer Technology (ACT) program at Microelectronics and Computer Technology Corporation (MCC) in Austin, Texas. The major functional components of the ORION system, as shown in Fig. 1, consists of Message Handler Subsystem, Object Subsystem, Transaction Subsystem, and Storage Subsystem.

The message handler receives all messages sent to the ORION system. The object subsystem provides high-level data management functions. The transaction management subsystem coordinates concurrent object accesses and provides recovery capabilities. The storage subsystem manages persistent storage of objects and controls the flow of objects between the secondary storage device and main memory buffers. Likewise, from the experience of ORION system, we construct a three-tier CORBA/OODB integration architecture as shown in Fig. 2. We can use the Object Request Broker (ORB) to act as the message handler to transfer messages between distributed objects. We use the OTS to provide transaction semantic. Finally, in the experience of ORION system model, the OODB is treated as a storage subsystem to support object persistency intuitively. Besides the successful experience of ORION, the standard interfaces of CORBA and OODB will facilitate the integration architecture. All the requirements of distributed object management will be satisfied in the CORBA/OODB integration architecture.

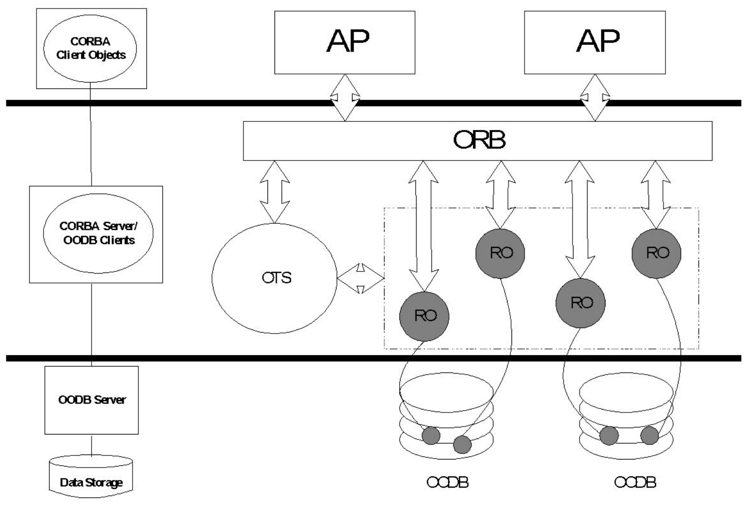

Fig. 2. The three-tier CORBA/OODB integration architecture.

### 1.3 The Goal

As the demand for information continues to grow, so do the software vendors make their applications more open and accessible. An open architecture is indeed needed for future systems. The goal of this paper is to introduce a general and easy-to-implement model that allows software vendors to make the time-to-market of their products shorter with lower development cost.

The ODA model suggested by ODMG is now implemented by some vendors on the market [10], but those products must be ORB vendor proprietary with the specific object activation mechanisms as well as transactional operations [17]. We are dedicated to providing an open and pluggable architecture to adopt any ODMG 2.0 compliant OODB and a spontaneous code generation model to reduce the burdens for the integration of CORBA and OODB. The proposed architecture follows the CORBA and ODMG standards and can co-exist with the two stand-alone systems.

## 2 Backgrounds

Some fundamentals must be introduced before describing the CORBA/OODB integration architecture. These backgrounds include the CORBA architecture, Object Transaction Service, and the ODMG standard. We will briefly describe them in the following subsections.

### 2.1 The CORBA Architecture

Providing data access to business applications in heterogeneous environments is very difficult. Those applications might want to access data through different programming languages, diverse operating systems, and distinct computing machines. Environment heterogeneity leads to a lot of problems, which motivated the birth of CORBA proposed by the OMG.

Fig. 3 shows the CORBA Architecture [1], [2]. The IDL Stubs/Skeletons define how clients invoke corresponding services and perform marshal/unmarshal which encode/decode the operations and parameters into flattened message formats. The Dynamic Invocation Interface lets clients discover the method to be invoked at run time. The run-time meta information of objects is stored in the Interface Repository. The Object Adapter provides the run-time environment for instantiating server objects, passing requests to them, assigning them object references, and registering them to the Implementation Repository. CORBA specifies that each ORB must support a standard adapter called the Basic Object Adapter (BOA) [3].

### 2.2 Object Transaction Service

OTS is one of the Common Object Services of CORBA. The concepts of transactions are important, especially when we need to develop reliable applications. Here, we use OTS to help programs perform the administrative functions by accessing shared objects in OODB. The OTS supports the concept of a transaction, which is a single atomic unit of work, and processes the ACID properties, which characterize a transaction in a number of ways.

Fig. 4 shows the CORBA OTS functional diagram [6]. The detail interactions between transaction originator, recoverable objects, and other components of OTS are described as follows, step by step:

1. The transaction originator issues a request to the *Factory* to begin a transaction and a unique *Control* object is returned.

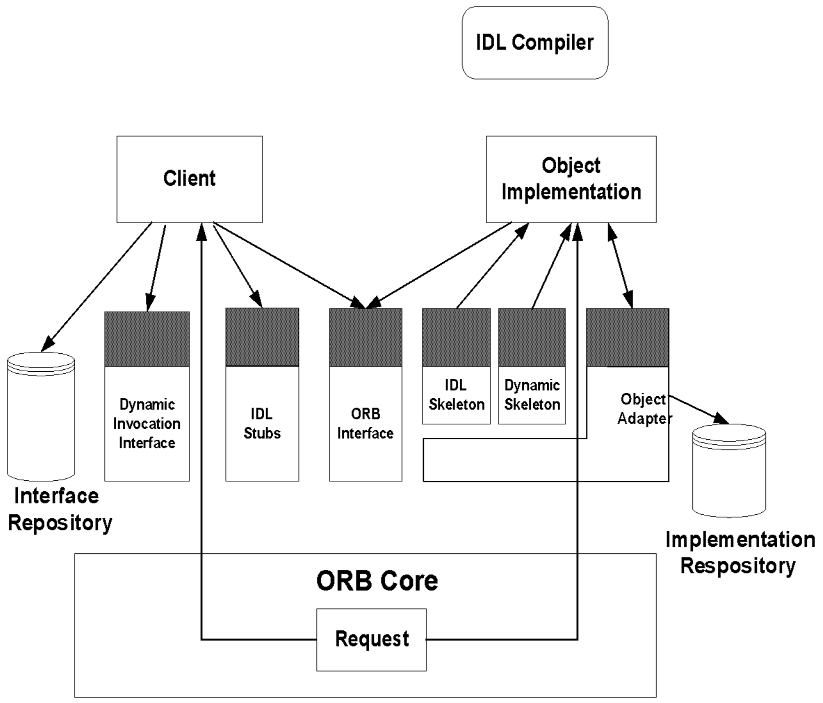

Fig. 3. The CORBA architecture.

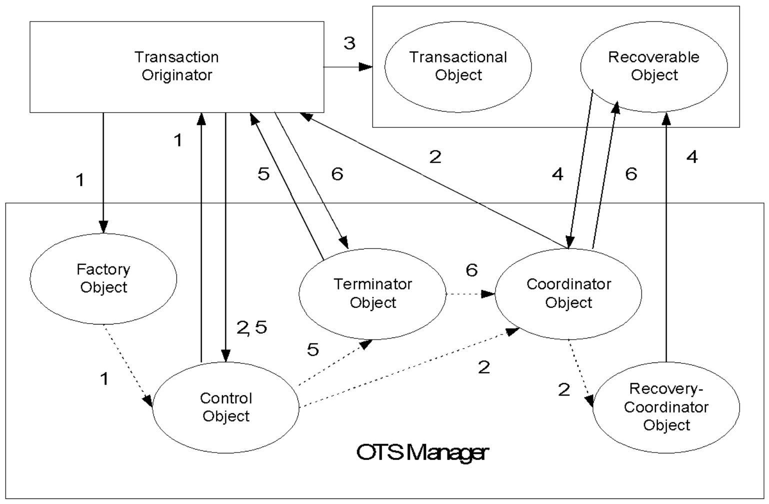

Fig. 4. Functional diagram of CORBA OTS.

2. Through the *Control* object, transaction originator can get a *Coordinator* object to coordinate objects, which join to the transaction.
3. The transaction originator invokes operations on the recoverable objects with Coordinator as an input parameter. Here, we assume that the transaction context is propagated explicitly by passing objects defined in OTS as explicit parameters.
4. Recoverable objects use *register_resource* method invocations to register themselves as *Resource* objects to the *Coordinator* object. A *RecoveryCoordinator* object will be returned after the registration.
5. The transaction originator gets a *Terminator* object through the *Coordinator* object and prepares to terminate the transaction.
6. The transaction originator uses the *Terminator* object to complete the transaction. The *Coordinator* coordinates the termination process among *Resource* objects using the two-phase commit protocol.

### 2.3 The ODMG

ODMG is an organization dedicated to promoting open OODB architectures. The ODMG defines three essential issues to be standardized, which are Object Definition Language (ODL), Object Manipulation Language (OML), and Object Query Language (OQL). ODL is the standard declarative language, which is used to define the interfaces of objects in OODB. OML is used to manipulate OODB objects. For instance, one would like to create a new object, delete another object, and the like. OQL is the query interface used to browse the objects in OODB with special criteria. Through these unified standards, all the OODB objects could communicate with each other. In our design, we will use the interfaces defined in ODMG Standard: ODMG 2.0 allows CORBA applications to incorporate with proprietary object-oriented databases.

Fig. 5 shows the ODMG architecture. ODMG compliant OODB objects are defined through ODL. The ODL preprocessor analyzes the ODL object definition and generates the registry program to register the schema to OODB. In addition, the preprocessor outputs the necessary information for object persistency as well as the related header files for programming language binding.

## 3 CORBA/OODB Integration Models

For the reason of distributed environment characteristics such as shared database, shared business logic or message handling system, and the individual front-ends, the legacy system models have progressed to a new paradigm. The integration of CORBA and OODB should match those characteristics to satisfy the requirements of the present distributed applications. We will illustrate the integration through the viewpoint of models, which meet the requirements of user applications.

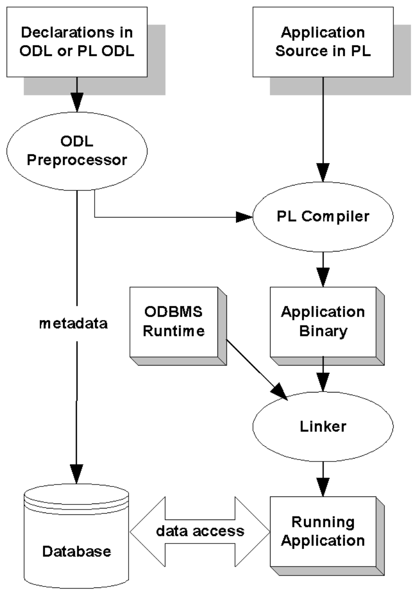

Fig. 5. The ODMG architecture.

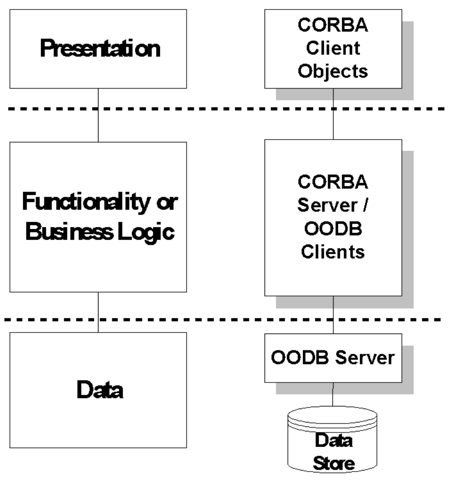

Fig. 6. The three-tier CORBA/OODB integration model.

### 3.1 The Three-Tier Model

The key characteristic of a three-tier model is the separation of distributed computing environment into layers of *presentation*, *functionality*, and *data component* [11]. This is needed for building flexible, extensible architecture with a manageable environment and facilitating rapid development of robust applications. Experiences over the past several years illustrate that CORBA is well-suited to best-effort, client-server applications running over networks [12]. In our design of CORBA/OODB integration, we propose a three-tier model that separates CORBA clients, OODB, and other CORBA objects into three different layers, as shown in Fig. 6.

### 3.2 Data Model

A data model is a logical organization of the real-world objects, constraints on them, and relationships among objects. Object-oriented applications use existing object-oriented programming languages like C++ and Smalltalk to express their semantics. Most current object-oriented database systems also support OO programming language binding. However, CORBA does not support mechanisms to bind OODB directly. In the integration of CORBA and OODB, we face the problems of impedance mismatch emerging from different data models of CORBA and OODB [13]. The main problem for integrating these two data models is that any data retrieved from OODB server to CORBA servers have to be translated from their database representation to the in-memory CORBA programming language specific representation. The translation between these two data models includes the object type, operation, and relationship mapping.

#### *3.2.1 Type Mapping*

Here, we use C++ programming language as the target language to show the type mapping. Table 1 shows the mapping of fixed-length primitive types, whose length is known at compile time, between CORBA and OODB. There are also other types defined by ODMG such as String, Date, Time, TimeStamp, Collection, Set, Bag, List, and Varray. In our design, we define the IDL object interfaces corresponding to those OODB object types, respectively, and implement the codes for them through wrapper techniques. We show some nonprimitive type mapping in Table 2.

TABLE 1
Primitive Type Mapping for CORBA and OODB

| OODB ODL | CORBA IDL | C++ |
|---|---|---|
| int, short int, unsigned int | short | CORBA::Short |
| long,unsigned long | long | CORBA::Long |
| Float | float | CORBA::Float |
| Double | double | CORBA::Double |
| Char | char | CORBA::char |

The last kind of type mapping is the mapping of user-defined types, which is the classical key issue of object persistency. In OODB object model, the construction time of a user-defined type attribute depends on the characteristics of programming languages. For example, while constructing an object, the object constructors do not construct the nonprimitive type as well as user-defined type attributes in Java, but they do in C++. In the integration of CORBA and OODB, the CORBA developers leak just the object interfaces to clients instead of exposing the database schema, that is, the clients do not have the knowledge of the memory layout of a persistent CORBA object. We map user-defined type attributes into *get* and *set* methods. While constructing a CORBA object, the embedded user-defined type attributes (interobject reference) will be constructed recursively in their constructors. But, the attributes of the user-defined type object will be initialized only when the *get* method is called. In Table 3, we show the constructor and *get* method for the object with embedded user-defined type attribute in it. While constructing the CORBA_B object, the CORBA_A object will be constructed recursively in the CORBA_B object constructor. Only when the *get* method is called will CORBA_A object states be initialized from the associated OODB object by invoking the *fetch* method.

#### *3.2.2 Operation Mapping*

All operations on OODB objects can be divided into two parts, relationship related and nonrelationship related. For nonrelationship operations, the CORBA server objects are implemented as an agent for OODB object operations. On the other hand, the relationship related operations are implemented as methods of CORBA objects. ODMG defines three kinds of relationships between objects. They are 1-1, 1-m, and m-m relationship, which denote the cardinality of the target object in one relationship. We use three methods for the relationship mapping. They are *Get_<related object>_relation*, *Set_<related object>_relation*, and *Clear_<related object>* operations. The major difference between those methods in 1-1 and 1-m or m-m relationships is that the *<related object>* and return values of methods should be nonfixed length types such as Collection, Set, List, and Varray. Whether two CORBA objects have relationship or not depends on the relationship of their associated OODB objects.

TABLE 2
Nonprimitive Types of Mapping for CORBA and OODB

| OODB ODL | CORBA IDL |
|---|---|
| interface **Date** {<br>typedef unsigned short ushort;<br>enum Weekday {Sun, Mon, Tue, Wed, Thu, Fri, Sat};<br>enum Month { Jan, Feb, Mar, Apr, May, Jun,<br>Jul, Aug, Sep, Oct, Nov, Dec};<br><br>ushort year();<br>ushort month();<br>ushort day();<br>ushort day_of_year();<br>Month month_of_year();<br>Weekday day_of_week();<br>boolean is_leap_year();<br>boolean is_equal(in Date a_date);<br>boolean is_greater(in Date a_date);<br>boolean is_greater_or_equal(in Date a_date);<br>boolean is_less(in Date a_date);<br>boolean is_less_or_equal(in Date a_date);<br>boolean is_between(in Date a_date, in Date b_date);<br>Date next(in Weekday day);<br>Date previous(in Weekday day);<br>Date add_days(in short days);<br>Date substract_days(in short days);<br>long substract_date(in Date a_date);<br>}; | interface **Date** {<br>enum Weekday {Sun, Mon, Tue, Wed, Thu, Fri, Sat<br>};<br>enum Month {Jan, Feb, Mar, Apr, May, Jun, Jul,<br>Aug, Sep, Oct, Nov, Dec };<br><br>attribute short year;<br>attribute Month a_month;<br>attribute Weekday day;<br>short day_of_year();<br>boolean is_leap_year();<br>boolean is_equal(in Date a_date);<br>boolean is_greater(in Date a_date);<br>boolean is_greater_or_equal(in Date a_date);<br>boolean is_less(in Date a_date);<br>boolean is_less_or_equal(in Date a_date);<br>boolean is_between(in Date a_date, in Date b_date);<br>Date next(in Weekday day);<br>Date previous(in Weekday day);<br>Date add_days(in short days);<br>Date substract_days(in short days);<br>long substract_date(in Date a_date);<br>}; |
| interface **Collection** {<br>unsign long cardinality();<br>boolean is_empty();<br>boolean is_ordered();<br>boolean allows_duplicates();<br>void insert_elements(in any element);<br>void remove_element(in any element);<br>Iterator create_iterator(in boolean stable);<br>BidirectionalIterator create_bidirection_iterator(<br>in boolean stable); | interface **Collection** {<br>readonly attribute long cardinality;<br>readonly attribute boolean empty;<br>readonly attribute boolean ordered;<br>void insert_element(in any element);<br>void remove_element(in ayn element);<br>Iterator create_iterator(in boolean stable);<br>BidirectionalIterator create_bidirection_iterator{<br>in boolean stable); |

TABLE 3
User-Defined Type Object Construction

| | CORBA_A | CORBA_B |
|---|---|---|
| IDL | interface CORBA_A {<br>short short_value;<br>}; | interface CORBA_B{<br>CORBA_A a;<br>}; |
| Constructor | Constructor of CORBA_A{<br>}; | Constructor of CORBA_B{<br>a = new CORBA_A;<br>}; |
| *get* method | CORBA_short CORBA_A::get_short_value(){<br>return this->OODB_dirty->short_value;<br>}; | CORBA_A CORBA_B::get_a(){<br>a.fetch(this->OODB_clean);<br>_duplicate(a);<br>return a;<br>}; |
| *fetch* method | Void CORBA_A::fetch(Ref<Object> handle){<br>this.OODB_clean = handle;<br>...//associate OODB obj with CORBA obj<br>this.short_value = handle->short_value;<br>}; | Void CORBA_B::fetch(Ref<Object> handle){<br>this.OODB_clean = handle;<br>...//associate OODB obj with CORBA obj<br>}; |

## 3.3 Transaction Model

Transaction management is a critical issue in OODB system as well as any distributed system. It defines the scope of persistent operations. Only changes within the transaction scope can be made persistent. In the current ODMG Object Model, transient objects are not subject to transaction semantics. This means that the states of modified transient object need not be restored when a transaction makes the abort decision. In the CORBA/OODB integration architecture, data objects are collection of persistent objects, which are defined through the ODMG ODL and provided by the RO Server, which is a key component in the integration architecture. Transient CORBA objects can co-exist with the integrated CORBA/OODB persistent objects in business applications.

The ODMG standard does not require that OODB vendors support distributed transactions that span multiple processes and/or span more than one database. However, the OMG OTS allows transactions to span multiple threads, multiple address spaces, or more than one logic database, which may be implemented as one or more physical databases. To integrate the two transaction models in OODB and OTS, there should be a mechanism to manage the relationship between the CORBA and OODB transactions. In our design, we use the RO server to begin OODB transactions as logic transactions for CORBA. The collection of logic transactions managed by the OTS coordinator forms a CORBA transaction. Through the design of RO server, the distributed transactions are supported in the model of CORBA/OODB integration. The management of transactions is shown in Fig. 7. There could be multiple resource objects in a CORBA transaction spanned in several threads to represent several persistent objects in multiple databases. The RO server starts corresponding processes to serve the operations for the persistent objects and control the transaction semantics through the help of OODB transaction manager. In addition, concurrent accesses to CORBA objects are denied by the RO Factory in the RO server through the recognition of the unique OODB object handle, which is managed by OODB. The well-controlled transaction model in our integration framework not only guarantees the integrity of business objects but also solves the unspecified concurrent access semantics in POS.

In the transformation of transactions, whether or not OODB vendors support distributed transactions, the distributed transactions are supported in the CORBA/OODB integration architecture. It not only keeps the transaction semantics for CORBA applications but also promotes the functionality for OODB. It is the major advantage to integrating CORBA and OODB through OTS than POS and ODA, which just treats OODB as a data store and wastes the original functionality supported by OODBs.

## 3.4 Fault Model

We assume the optimistic fault model that the failures seldom occur. When a failure occurs, we must keep the consistency of the database. All the operations on objects must be *REDO* or *UNDO*, depending on the time at which the failure occurs. In OTS transaction scenario, the recovery point is the time while OTS issues the *REQUEST-TO-PREPARE* operation to all its participants [6]. Before the recovery point, all operations should be *UNDO*, i.e., all operations are invalid in the transaction. On the contrary, after the recovery point, all the operations on the objects must be persistent and the values of object states are invariant. There are two possible failures in our design:

- ***RO Server Crash*:** This failure could occur due to the media failure and it will affect all the CORBA clients. We will *REDO* or *UNDO* all the operations on all objects when the server is restarted.

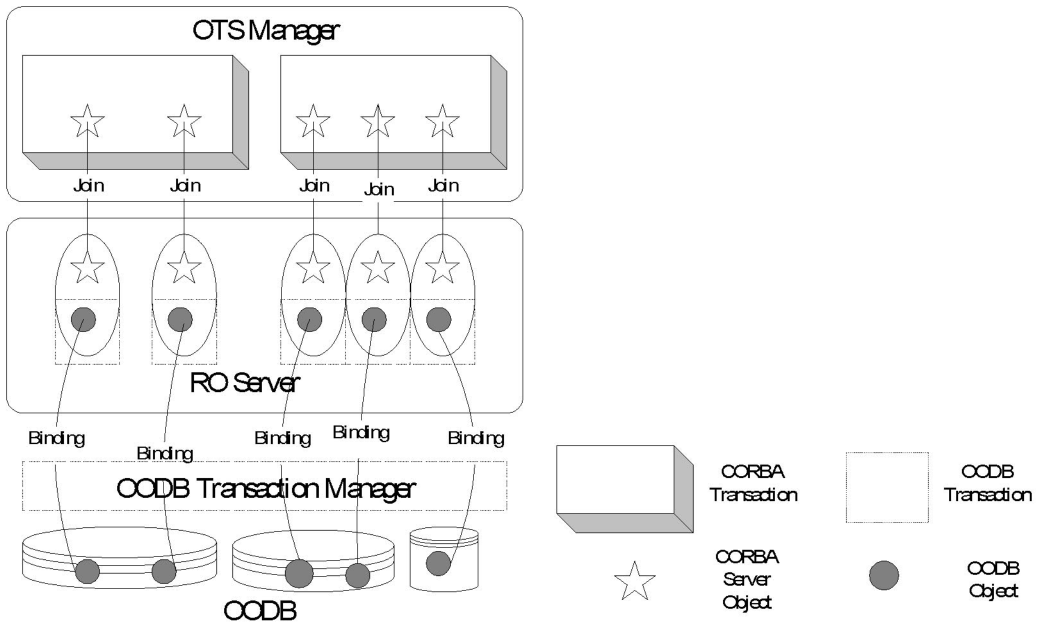

Fig. 7. The transaction model for CORBA/OODB integration.

- ***Object Crash*:** The failure affects the object itself. It will automatically recover the states of persistent objects while any subsequent bind operations to the RO Factory of the same type are issued by any CORBA client.

In the two-phase commit protocol, the participants of a transaction should be responsible for the recovery after failures, which might occur in the uncertainty period [14]. When failures occur, all we are concerned about is how to recover from modified object states to the consistency states. If failures occur before the *REQUEST-TO-PREPARE* operation issued by OTS Coordinator, the transaction will be aborted and undo all the operations. While the *COMMIT* operation is issued, the changes will be persistent and the integrity of states will be assumed after the recovery routines. If failures occur for the reason of object crash or RO server crash, we use an automatic mechanism to recover the consistent states by asking the OTS Coordinator what the decision is. In the recovery routine of our design, we keep the necessary information in the *Recovery Object* to prevent inconsistency caused by failures. The necessary information includes the *RecoveryCoordinator*, original state, and new state of the modified object. When recovery routine starts, all modified objects will be consistent through the help of *RecoveryCoordinator*.

## 4 The Integrated Architecture

### 4.1 System Overview

Fig. 8 illustrates the functional components in the CORBA/OODB integration architecture. The components consist of CORBA Client, OTS, RO Servers, and the pluggable ODMG compliant OODB. Interoperations between these components are based on the standard CORBA and ODMG interfaces. To provide CORBA clients with a well-suited transaction environment, it is vital to support a transparent mapping of the data models and transaction scopes between CORBA and OODB. For the data model mapping, we use wrapping techniques to combine the CORBA transactional object with OODB object into the RO server object. As for the transaction semantics, we use RO Server to serve as the dispatcher for the distributed CORBA transaction, which spans multiple OODB transactions.

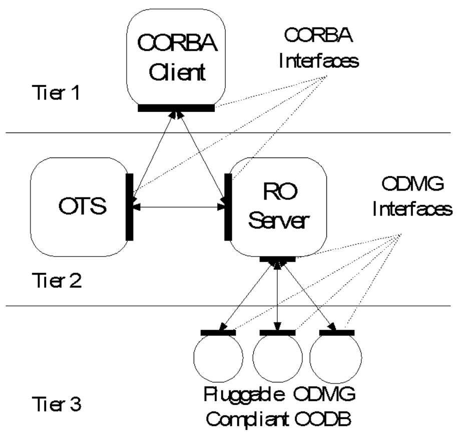

Fig. 8. The functional components of CORBA/OODB integration.

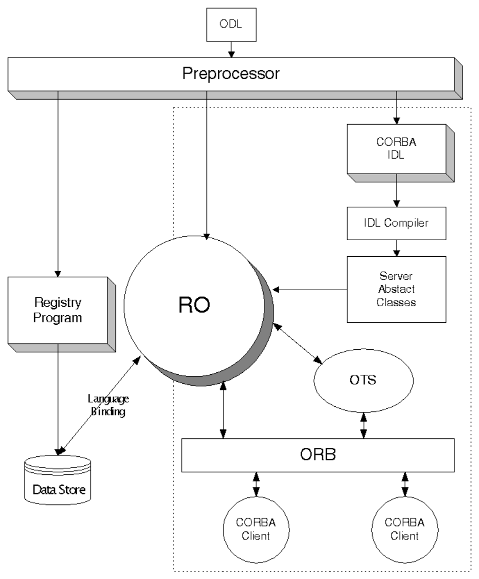

Fig. 9. The CORBA/OODB integration architecture.

### 4.2 System Architecture

In Fig. 9, we illustrate the system architecture of our design. This architecture can be treated as the integration of CORBA programming model and OODB programming model. For an OODB developer, in a general scenario, he/she must define the object schema by ODL. After the preprocessor processes the ODL definition, the meta information will be registered to the data store. The preprocessor will also generate the necessary header files and glue information. All that a developer needs is to implement the object in OODB way without caring about the knowledge of CORBA. For a CORBA client application programmer, object interface is the only thing he/she need to know, i.e., all he/she needs to care about is the IDL object definition which is generated from the ODL defined by the OODB developer.

#### 4.2.1 Preprocessor

The preprocessor in Fig. 9 is a key component in the CORBA/OODB integration architecture to support the spontaneous code generation model. It deals with all the tasks for the integration and generates the necessary codes automatically. The generated components are *Registry Program, Header and glue files, CORBA IDL, Partial RO object implementation, CORBA server implementation,* and *Recovery Object.*

To promote the portability of generated codes, it is inevitable that the preprocessor should know the difference between specific operations played in these two stand-alone systems. For example, different OODB systems will not have the same database system entry point. ORB vendors will have their own methods to bind server objects, too. There are problems encountered, usually by ORB and OODB vendors in integration processes. Now, it is no

TABLE 4
Null Interface Definition

| |
|---|
| //ODL<br>class A {}; |
| //IDL<br>Interface CORBA_A{}; |

TABLE 5
Private Information for Object Implementation

```
class CORBA_A {
        char *recovery_coordinator;
        Ref<A> dirty_object;
        Ref<A> clean_obj;
        Ref<A_Recovery> recovery_obj;
        void fetch(Ref<A> aA);
        Ref<A> get_handle();
        void set_reference(Ref<A> aA);
};
```

longer the problem of OODB or ORB vendors. It is the responsibility of preprocessor tool implementors to be conscious of the differences of codes between vendor proprietary functionalities. That is, this new approach solves problems encountered by ORB and OODB vendors through a third party—the proposed preprocessor. Through preprocessor tools, it not only reduces the time needed in the integration processes but also helps OODB and ORB vendors to plug their products into this architecture easily and quickly.

### 4.2.2 RO Component

The RO component in the integration architecture can be treated as a black box to integrate CORBA and OODB smoothly. There are several entities in the black box, including the RO server process, RO Factories, and RO objects. The RO server process provides RO Factories to the CORBA clients. RO objects are created by the RO Factories within the RO Server for server object implementation and support the binding with the OODB in the integration architecture. The "R" of RO means that the component owns the capabilities, which include inheritance from OTS Resource interface, to register itself to CORBA transaction and recovery ability to maintain the consistency of a database. The RO object deals with all the tasks to tie CORBA and OODB tightly with the functionality of persistency and recoverability. The major tasks of the RO component are: *binding OODB objects*, *implementing OTS resource interface*, *transactional operations*, *recovery functionality*, and *mapping transaction scopes.*We detail these tasks in the following.

- ***Binding OODB objects*****:** The RO object plays two roles in the integration architecture: CORBA server object and OODB client object. Through the object implementation framework, the CORBA server object can treat the implementation of an OODB object as the servant of its own [2]. The key point to bind the CORBA object and OODB object is the mapping between OODB object handle and CORBA object reference. RO Factory must provide a mechanism to associate them. In general, RO objects keep some private information, which includes the object reference of *Recovery_Coordinator* and object handles of clean and dirty OODB objects. RO objects also need additional private methods to fetch OODB object states to CORBA object. Those private methods are *get_handle()*, *set_reference()*, and *fetch()*. The *get_handle()* returns the OODB object handle to CORBA object. The *set_reference()* method sets the OODB object handles to the CORBA object. The *fetch()* method associates the CORBA object states with the OODB object state. This private information and methods are generated automatically by the preprocessor. Table 4 shows a null ODL and null IDL interface definition and Table 5 is the definition of the corresponding object implementation in C++.
- ***Implementation of OTS Resource interface*****:** To participate in the 2PC of OTS, the RO object must have the implementation of *Resource* interface which includes the *commit*, *prepare*, *rollback*, *commit_one_phase*, and *forget* methods. The object of *Resource* interface is registered to an OTS transaction and used in the recovery phase to ask the decision of the transaction or to replay the two phase commit protocol. In our proposed transaction model, the RO Factory delays the commit votes to the *Coordinator* to enhance the performance and to guarantee the transaction semantics. This is the critical point that the ODA misses. Matching the transaction semantics through the standard transaction scenario will integrate CORBA and OODB seamlessly. It makes the RO object not only a persistent object but also a manageable object. On the other hand, reusing OTS to integrate CORBA and OODB indeed provides the functional logic and meets the requirements of business applications, which is missing when POS are used.
- ***Transactional operations*****:** There are three major transactional operations that should be concerned in the RO server: *Create*, *Query*, and *Delete* for manipulating objects. The RO Factory, which manages specific-type objects, provides these operations to CORBA clients. We implement the *Query* method as an agent to OODB through the ODMG OQL. The *Delete* method checks if the reference count is zero as possible and then determines whether to kill an object or not. Object reference will be invalid after the Delete method makes the OK decision. When creating an object, the RO Factory first creates an OODB object and a CORBA object, and then calls the *set_reference* and *fetch* methods to bind them together. Finally, it returns an RO object to clients. In Table 6, we present some pseudocodes for the *Create* transactional methods.
- ***Recovery Functionality:*** Table 7 shows the ODL schema of the *Recovery Object*. Each RO object keeps the information of RecoveryCoordinator, the dirty object, and clean object to ask OTS what the transaction decision is to maintain the integrity of itself. Every time an RO Factory is

TABLE 6
Transactional Operations

```
RO_object   RO_Factory_A::Create(CosTransactions::Coordinator TxCO){
   Ref<A> OODB_obj =  new (&OODB) A;
   // new operation is overloaded. ODMG does not define how to create objects.
   //It is the preprocessor implementor   s responsibility to write the correct codes for OODB vendors.
   RO_object  tmp = new RO_object;
   tmp->set_reference(OODB_obj);
   tmp->fetch(OODB_obj);

   // register the resource object to OTS

   _duplicate(tmp);
   return tmp;
}
```

TABLE 7
The Schema of the Recovery Object in ODL

```
// Recovery Object ODL Schema
class Recovery_Object {
   String Recory_Coordinator;
   Ref<Object_type> original_object;
   Ref<Object_type> dirty_object;
};
```

bound, it checks if any *Recovery Object* with the same type exists in the database. If yes, it means that some failures have occurred and there must be *REDO* or *UNDO* in the transaction. In the optimistic fault model, it seldom happens in normal condition.

- ***Mapping transaction scopes*:** The RO Factory consists of several objects with the same data type and it coordinates all the objects in one of the spanned OODB transactions of an OTS transaction. The RO Factory dominates the relationship between OTS and RO objects in the integration architecture and keeps the integrity of transaction semantics between the two transaction contexts. The mapping of transaction contexts can be divided into the following three types in Fig. 10, where Tx means transaction(s).

The key point for managing the transaction semantics is the well-controlled relationships among the client threads, server threads, and the OTS Coordinators. One Coordinator object manages a distributed transaction context, which may span many OODB transaction contexts. RO Factories manage concurrent accesses to OODB objects through the recognition of OODB object handles. Object consistency is also guaranteed through the cooperation of RO Factories and OTS, which supports the atomicity and isolation properties. The OODB provides the durability property. That is, through the cooperation among RO Factories, OTS, and OODB, the ACID properties are satisfied to guarantee the integrity of transaction semantics and to integrate CORBA and OODB seamlessly.

## 4.3 The System Development Scenario

We insist that the integration of CORBA and OODB is spontaneous. That is, the developers in different domain do not need to understand any knowledge about the other part. Fig. 11 shows the system develop scenario, which is divided into three gray parts—the OODB developer, the CORBA client AP, and the spontaneous code generation part—for the CORBA/OODB integration architecture. From the viewpoint of an OODB developer, as shown in Part A of Fig. 11, there are only two tasks that they have to do. The OODB developers should define the object schema through ODL and then implement the OODB object in OODB programming style. With the OODB object implementation, the ODL preprocessor will bring the OODB object to CORBA environment automatically. In the beginning, the preprocessor takes the ODL as inputs to generate the corresponding IDL object definition, registry program and the RO implementations, which are based on the class definition and server skeleton generated by IDL compiler and rely on the OODB header files as well as the glue information. Part B of Fig. 11 is the main part for the spontaneous code-generation model. It not only separates the developers into two parts but also simplifies the application development for the CORBA/OODB integration system. In the perspective of CORBA client AP developers, as shown in Part C of Fig. 11, the IDL compiler takes the IDL object definition as inputs and generates the necessary information for the client application development. The clients use the class definition to develop applications through the help of client stub without the knowledge of the OODB objects.

## 4.4 Application Scenario

Fig. 12 shows the general scenario for client applications in the CORBA and OODB integration architecture. When a CORBA client binds an RO Factory, the RO server forks a process to deal with the transactional operations on objects of that type. After getting the RO Factory object reference, the client can use *Create* or *Query* operations to construct an RO server object to deal with further requests, which include the methods defined in IDL and manipulations on relationships between objects. After constructing an RO server object, which consists of the body of the Transactional Object and Resource object for OTS transaction style,

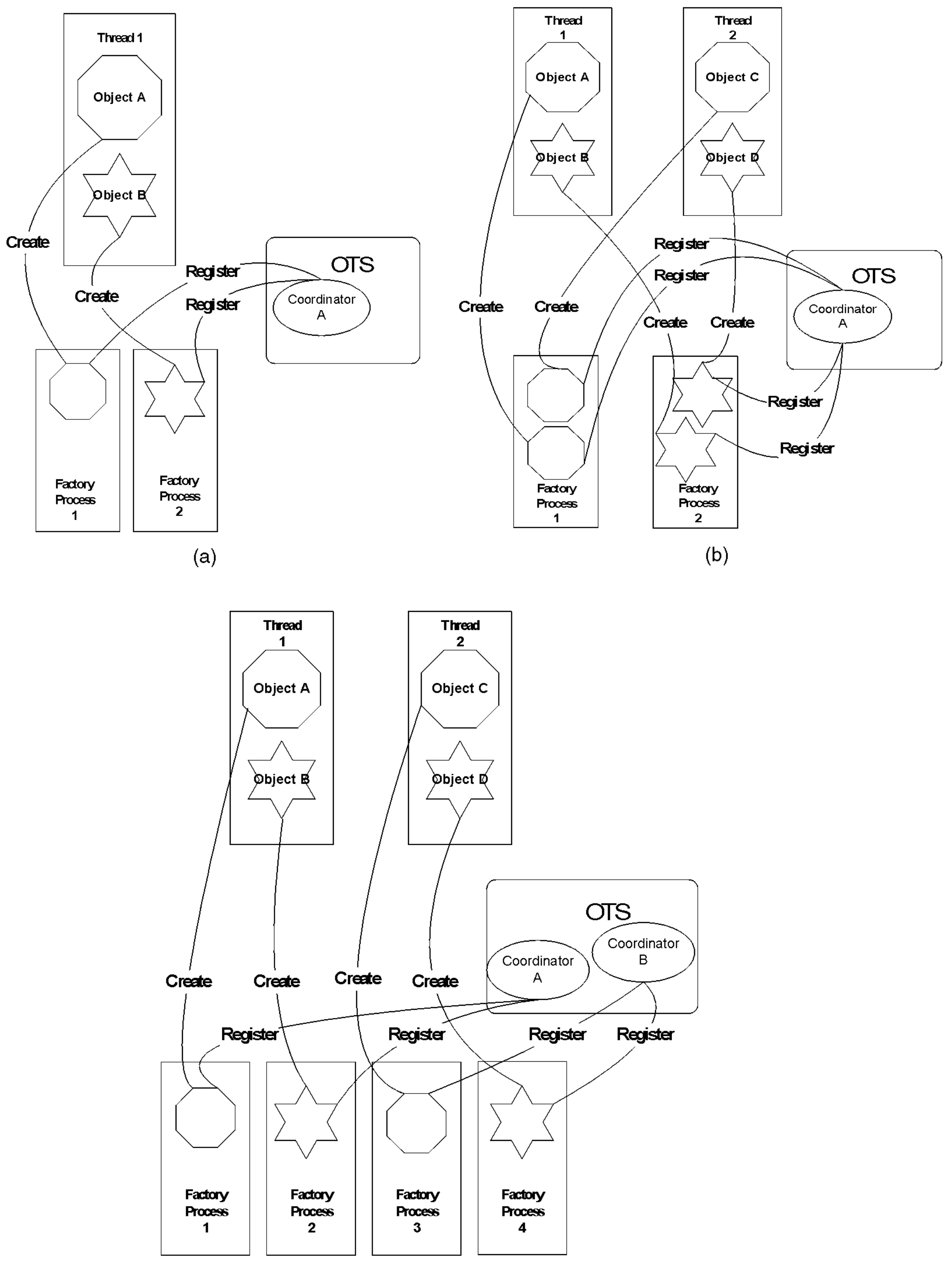

Fig. 10: (a) Single thread joins single Tx; (b) multiple threads join single Tx; (c) multiple threads join multiple Tx.

the RO Factory registers the RO server object to OTS. The Resource object body is used to join the OTS with two-phase commit protocol and used in the recovery phase. The TO, which consists of the OODB data object and the interfaces defined in IDL, receives the transactional operations from client side and updates the OODB object states through OODB language binding.

Separating the server objects into processes by types simplifies the tasks for object management and speeds up the recovery routines. Through the RO Factories in different processes, as the mapping of transaction contexts mentioned in the previous section, the CORBA transaction context can span several OODB transactions controlled by RO Factories and the OTS coordinator. The concurrent access of the OODB objects between CORBA transaction contexts is controlled according to the OTS isolation property. And, the RO Factory controls the concurrent access to the CORBA in-memory object.

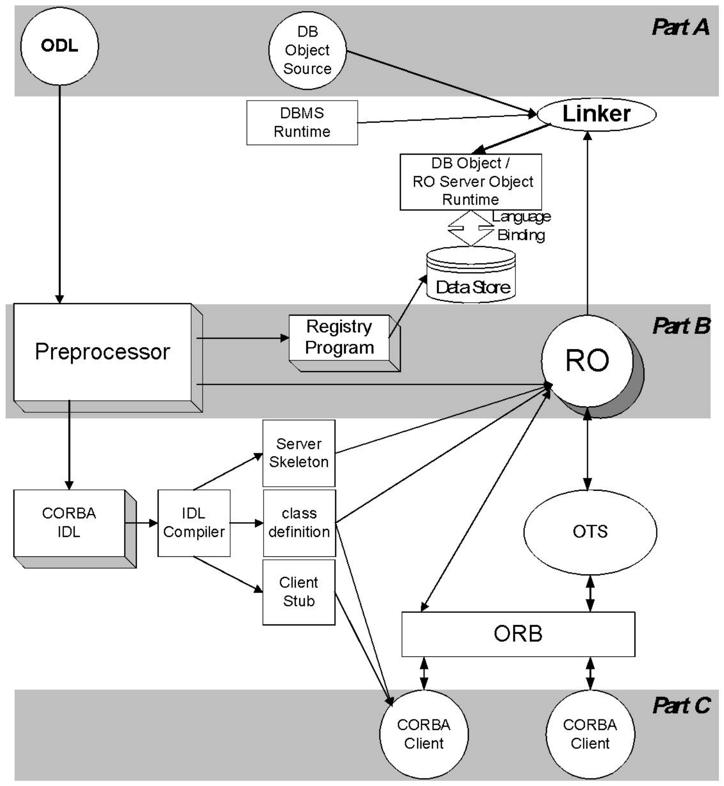

Fig. 11. The system development scenario.

## 5 Discussion

Several issues must be handled with care while integrating CORBA and OODB. These important issues include the object reference representation, object activation, object deactivation, transaction management, and system performance. In our design, we use the RO Factory and the wrapping techniques to support the activation and deactivation of the RO objects. As for the transaction management, we use the OTS to serve as the transaction manager to support distributed transaction. There are other approaches [15] to integrate CORBA and OODB, as well as several integration tools on the market. Notably, the ObjectStore, Versant, and Objectivity/DB are integrated with OODB via adapters, while $O_2$ and Persistence

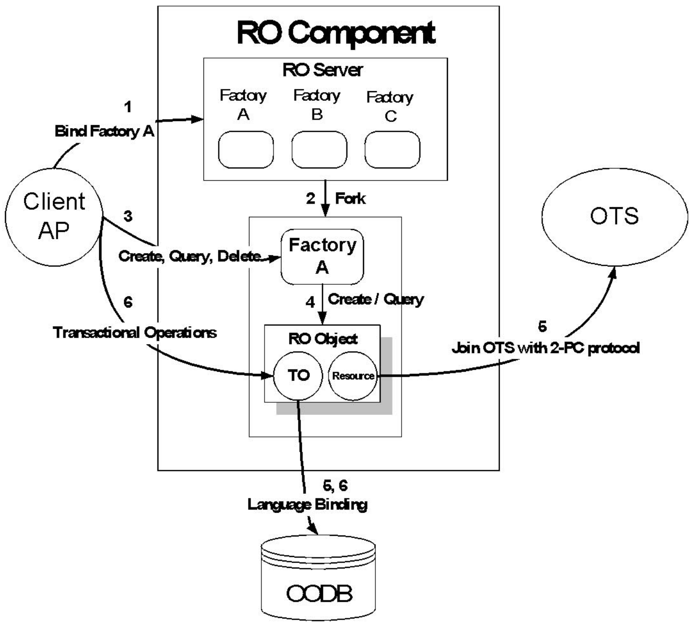

Fig. 12. The application scenario.

choose the front-end (wrapper) objects generating approach [16], [17], [18]. Both the techniques concern the object reference representation, object activation, object deactivation, and transaction management for the vendor-specific OODB in their specific manners. In this section, we will discuss the issues for integrating CORBA and OODB by comparing the two techniques with our model.

## 5.1 Critical Issues

### 5.1.1 Object Reference Representation Issue

The difference between CORBA references to transient and persistent objects is that the persistent object contains the information to identify the corresponding OODB object. In the ODA approach, an adapter is responsible for the generation of object references. However, the programmer should take care of the construction of the wrapped object references while using the front-end tools on the market. In our model, take the Orbix BOA for example, the CORBA object references are generated through the BOA, which is supported by CORBA standard. The relationship between the CORBA object reference and OODB object handle can be kept in an association table in the RO server process. Formally, the wrapped object can be treated in two abstraction levels—OODB abstraction level (the persistent part) and CORBA abstraction level (the nonpersistent part). For the CORBA abstraction level, the nonpersistent part is the same as general CORBA objects without persistent states. For the OODB abstraction level, the persistent part of the CORBA object is registered to the OODB by the preprocessor. That is, when clients get the CORBA object reference, it is bound with the proper OODB object handle and they can manipulate the object with other COSS services as other normal CORBA objects do. Most importantly, all the necessary codes for the association table are generated automatically by the ODL preprocessor. No vendor specific libraries need to be added to the adapter and the programmers do not need to write any extra codes at all.

### 5.1.2 Object Activation and Deactivation Issues

The object activation and deactivation issues are trivial for the adapter technique because it controls the generation of the object reference. On the contrary, when wrapping technique is used, the programmer must take the responsibility for activating the OODB object through programming language binding and controlling the relationship between the CORBA wrapper object and the activated OODB object. In the case of object deactivation, the programmer should control the deactivation of OODB objects, as well as wrapper objects, through the management of reference count. In our model, the RO Factory controls the relationship between the CORBA abstraction level as well as the OODB abstraction level in the wrapper object. The RO Factory manages the object reference counts to control the object lifetime. It activates the wrapper object while clients call *Create* or *Query* method and deactivates the object when *Delete* method is called. To alleviate the burden of the programmer, the ODL preprocessor generates the necessary codes for the responsibility of taking care of the reference count of the objects.

### 5.1.3 Transaction Management Issue

No matter which technique on the market is used to integrate CORBA and OODB, the model for transaction management is vendor specific. For example, the adapter uses per operation, per transaction, or the phased mechanism to model the transaction semantics [17]. The phased mechanism violates the CORBA principles of separating client from server's memory as well as making clients aware of persistence of the server-side object. The wrapping techniques for the existing tools leave the transaction management tasks for the programmer. It provides much more space for maneuvering, but also increases the burden of the programmers. In our CORBA/OODB integration architecture, we reuse the OTS to manage the transactions for applications [6]. The OTS not only provides the management of distributed transactions but also promotes the capability and usability for the restricted OODB transaction model. From the client's point of view, the reuse of OTS provides the standard CORBA transaction model. In addition, ACID properties, the recovery capability, and the single-level store model are transparently supported to their business applications.

## 5.2 Predominance of the CORBA/OODB Integration Architecture

There are several advantages to using the architecture proposed in this paper to build CORBA/OODB integration applications. These advantages include:

- ***The pluggable open architecture.*** There are popular CORBA/OODB integration tools on the market like the Orbix+Versant Object Adapter and Orbix+ObjectStore Object Adapter. All the products have the same characteristic that they develop an Object Adapter for each vendor-specific OODB. When a new OODB is introduced, the developers must rebuild an Object Adapter for it. On the contrary, we provide standard interfaces for each ODMG compatible OODB to plug into easily. If there are M ORB vendors and N OODB vendors, there will be $M * N$ adaptors and it will take almost $M * N$ times to build those adaptors. Through the integrated architecture, we just need to change a few of codes for the preprocessor implementation to adapt all vendors and their proprietary functions. It reuses the codes for each ORB and OODB integration through the standard interfaces and indeed reduces the development cost and shortens the time-to-market for their products. The integrated architecture is indeed much more open than the ODA approach.
- ***The standard CORBA transaction scenario.*** We reuse the OTS to help CORBA clients manage their business applications in transaction style. Instead of the usage of nonstandard phases for the adapter technique, which changes the original CORBA transaction programming model and brings extra burdens to CORBA client applications, the standard transaction scenario which reuses OTS will be kept no matter what kind of OODB are integrated.

TABLE 8
Average Times of Method Invocations Using Wrapping and Adapter Techniques (in Milliseconds) [18]

| | ORB | DB-only | Hand-written Wrapper | Adapter |
|---|---|---|---|---|
| *get* attribute | 2.3 | 13 | 78 | 52 |
| *create* an object | 7 | 35.3 | 193 | 245 |
| *delete* an object | 5.8 | 62.7 | 140 | 155 |

TABLE 9
Average Times of Method Invocations in Our Proposed Architecture (in Milliseconds)

| | OTS | DB-only | Preprocessor Wrapper |
|---|---|---|---|
| *get/set* attribute | 18.5 | 14.4 | 256.2 |
| *create* an object | 19.4 | 39.7 | 275.4 |
| *delete* an object | Non-sense | 30.5 | 263.5 |
| *query* an object | N/A | 59.1 | 313.1 |

- ***Multidatabase supported.*** A CORBA client process can manipulate many objects in various OODB simultaneously without knowledge of any vendor-specific programming model, especially the transaction model. The ODA and wrapper techniques are designed for a specific OODB, which is not enough for the multidatabase architecture. Through the reuse of OTS, the distributed transactions are well controlled. In addition, through the mapping of transactions automatically managed by each RO Factory, it will help to span the CORBA transactions into multiple database transactions.
- ***Solving POS problems.*** POS treats OODBs as data stores and defines interfaces for general-purpose data stores despite data management semantics so the interfaces will not match the OODB interfaces smoothly. If someone uses ODMG protocol to integrate CORBA and OODB using POS, he/she will find that there is no standard way to decide the transaction scopes. There is no way to guarantee the integrity of concurrent data access without the help of other components. It is not enough to integrate CORBA and OODB by POS. Instead of POS, the integrated architecture uses standard interfaces to integrate CORBA and OODB seamlessly and reuses OTS to match the OODB transaction scopes. It also defines three relationship methods to support relationship to CORBA objects, which is an important issue in business application and is not defined in POS. Especially through the user-defined type mapping, it provides the "Compound Persistent Object Service," which is lacking in POS for CORBA [7]. The proposed architecture provides not only the persistent data store for CORBA objects, it provides object management semantics for ORB and OODB vendors. It goes without saying that the integrated architecture solves the problems of unspecified semantics of operations and supports the management functionality to persistent objects, which is weakly defined in POS.
- ***Integrate with other CORBA services.*** As we mentioned above, the wrapper object could be treated as two abstraction levels, which are the CORBA abstraction level and the OODB abstraction level. We use the RO_Factory to manage transaction and concurrent access semantics for CORBA abstraction level through the help of OTS and use OODB to serve as storage manager to provide the persistency to CORBA objects in the OODB abstraction level. Likewise, we can reuse other services to manage other semantics for the wrapper object in the CORBA abstraction level easily. For example, we can reuse the Object Security Service to provide these CORBA objects different levels of security.

## 5.3 System Performance

Amirbekyan and Zielinski's paper [18] shows that the performance of handwritten wrapper objects is as good as adapter ones. Table 8 shows the times of each method invocation using wrapper and adapter techniques. Their experiments were performed on a Sun SPARC station 514 running Solaris 2.5, ObjectStore4, Orbix 2.1c ST, and OOSA 201. Times of method invocation were measured from the client side for pure ORB, pure OODB, and wrapper objects. In our architecture, preprocessor wrapping mechanisms, OODB and OTS will dominate the execution time of user applications. Table 9 shows the performance of our proposed architecture. Our experiments are performed on Pentium II 400, Windows NT workstation 4.0, Orbix 2.02, VC++ 5.0, and WOO-DB 2.0 Beta [21]. We do not measure the execution time for *delete* operation because it is nonsense to delete transient objects in a transaction context. And, it is another of COSS's issues, such as Naming Service, to provide a large number of CORBA objects to be queried, so we do not measure the *query* operation execution here also.

TABLE 10
The Overhead Ratio of the Handwritten Wrapper and the Preprocessor Wrapper

| | | ORB + DB | Overhead of Wrapper | Ratio (Wrapper/(ORB+DB)) |
|---|---|---|---|---|
| Hand-written Wrapper | *get* attribute | 15.3 | 62.7 | 4.10 |
| | *create* an object | 42.3 | 150.7 | 3.56 |
| Preprocessor Wrapper | *get/set* attribute | 32.9 | 223.3 | 6.79 |
| | *create* an object | 59.1 | 216.3 | 3.66 |

For the sake of different test environments, the measured execution time does not mean the precise overhead of integration techniques. The accurate and meaningful overhead for integration systems should be measured relative to pure systems. In this paper, the overhead of wrapper objects is measured by comparing the operation execution time with pure CORBA and OODB systems. The overhead is represented as ratio, which is shown in Table 10.

In the current version of our design, each invocation on objects is treated as an update operation, so there are the same execution times for *get* and *set* operations. The differences between *create* and other operations are that we do not need to consider the concurrent access and recovery control for *create* operation, but we do for other operations. From Table 10, it is obvious that the efficiency of *create* operation for nonoptimized preprocessor wrapper is almost the same as the optimized handwritten wrapper one. While invocation *get/set* operation on objects, we have to create *Recovery Objects* for them, so we have a little higher ratio of overheads than handwritten wrapper ones. It is usually a trade-off between providing concurrent accesses and recovery capabilities to each update operation or just arguing the operation execution time. In our design, each operation execution time includes the operation itself, the OTS 2PC, concurrent accesses control, and recovery procedures. It is reasonable that the nonoptimized preprocessor generated codes take more time than the specific, handwritten optimized wrapper ones. According to the results of our experiments, it is feasible to integrate these CORBA and OODB through this new proposed-automatic model.

## 6 Conclusion

CORBA is a software bus, which provides a middle layer for the interoperations between distributed object-oriented components. Through the standard interfaces, it is easy to plug new application components into the CORBA environment. In this paper, we propose an integrated architecture to plug OODB into CORBA. To integrate CORBA and OODB, the first task to deal with is to map the ODL to the standard IDL interfaces. We use a preprocessor to solve the mapping problems. In addition, to reduce the development cost and shorten the time to market, we use the wrapping technique to reuse the existing legacy codes. For regularity of the transactional programming models, the preprocessor can generate all the necessary codes for the integration.

In this paper, we propose a new architecture to integrate CORBA and OODB. We consider the mapping of data types, the management of transaction semantics, concurrent object access, and functionality of recovery while failures occur, especially the design of automatically code generation model. The most important feature of our design is that the integrated OODB is ODMG compatible. The usage of ODMG standard interfaces makes vendor specific ODMG compliant OODB pluggable. In an open system, it is significant to have standard interfaces to reduce the burden for announcing the vendor specific application interfaces and speed-up the time to market, as well as reduce the development cost.

## 7 Future Works

In the current version, we treat the OODB as the storage manager for distributed objects. From a global viewpoint, we need general meta information to represent the object hierarchy and the relationship between distributed objects. To build a distributed database management system, the global view of schema is the most important issue. Integrating schemas in different OODBs, even RDBs, is vital for building a distributed database system in the future, i.e., we might need another component like Naming Service for schemas. There are many tasks for us to do in the future to provide a truly open and complete architecture, for example, the management of event, security, and version control. We especially have to optimize the preprocessor to get better performance. We can take advantage of the integration architecture and reuse the services defined in COSS [3], like the OQS, Relationship Service, Security Service, as well as the Portable Object Adapter, to build a distributed object management system on CORBA.

## Appendix A

It is indeed helpful to understand the integration architecture and application development scenarios clearly through a simple example. The following example is written by C++ programming language and following the OTS specification and ODMG standards (see Table 11). The development environment is MSVC++ 5.0, Orbix 2.02, and WOO-DB 2.0 Beta version [19], [20], [21]. The registry program is not shown here because of its vendor specific characteristics.

Because we use the explicit transaction propagation mechanism, all methods in Factory must have a parameter

TABLE 11
The ODL Class Definition

```
//ODMG ODL
class Person
{
        public:
           Person();
           ~Person();
           String  name;
           short   short_value;
           float   float_value;
           Ref<Employee> coupleEmp inverse couple; // relationship definition
};

class Employee : public Person
{
          public:
           Employee();
           ~Employee();
           Ref<Person> couple inverse coupleEmp; // relationship definition

           void Set_Salary(double Salary);
           omDouble Get_Salary(); //typedef omDouble double for the reason of source code portability
};
```

TABLE 12
The Corresponding IDL Generated by the ODL Preprocessor

```
interface CORBA_Person {
              attribute string  name;
              attribute short   short_value;
              attribute float   float_value;

              CORBA_Employee get_coupleEmp_relation();
              void set_coupleEmpl_relation(inout CORBA_Employee aemployee);
              void clear_coupleEmpl_relation();
 };

interface PersonFactory{
        CORBA_Person Create(inout CosTransactions::Coordinator TxCO);
    void Delete(inout CORBA_Person aperson,
            in CosTransactions::Coordinator TxCO);
    void Query(inout CORBA_Person result, in string querystring,
            in CosTransactions::Coordinator TxCO);
};


interface CORBA_Employee : CosTransactions :: Resource , CORBA_Person{

              double Get_Salary();
              void Set_Salary(in double salary);
              CORBA_Person get_couple_relation();
              void set_couple_relation(inout CORBA_Person aperson);
              void clear_couple_relation();
};

interface EmployeeFactory{

        CORBA_Employee Create(inout CosTransactions::Coordinator TxCO);
        void Delete(inout CORBA_Employee aemployee,
                in CosTransactions::Coordinator TxCO);
        void Query(out CORBA_Employee result, in string querystring,
                in CosTransactions::Coordinator TxCO);
} ;
```

TABLE 13
Sample of a Client Program in the CORBA Transaction Style

```
try{
//Bind a OTS server
  TxF_var = CosTransactions_TransactionFactory::_bind(":OTS",    hostname   );
//Bind Factories on RO server
  PF_Var = PersonFactory::_bind("PersonFactory:RO_Server","hostname");
  EF_Var = EmployeeFactory::_bind("EmployeeFactory:RO_Server","hostname");
//Create a Transaction Context
   Ctrl_var = TxF_var->create(0);
//Get a Transaction Coordinator by a given transaction context
  TxCO_var = Ctrl_var->get_coordinator();

// operations in transaction scope
aPerson = PF_Var->Create(TxCO_var);
aEmployee = EF_Var->Create(TxCO_var);
aEmployee->SetSalary(100);
...
// Get Transaction Terminator
   TxT_var = Ctrl_var->get_terminator();

// Commit
  TxT_var->commit(0);

}
catch(CORBA_SystemException &SysEx){
...
}
}
```

TABLE 14
The RO_Server Main Program

```
main(){
  try{
        PersonFactory_ptr PF = new PersonFactory_i;
        EmployeeFactory_ptr EF = new EmployeeFactory_i;
        CORBA_Orbix.impl_is_ready(   RO_Server   );
  }
  catch(CORBA_SystemException &SysEx){
   ...
  }
}
```

to specify the transaction context that the client wants to join in. The generated IDL (see Table 12) must support the essential features—the interfaces for object creation, deletion, and query as mentioned in the previous section. The client can use the *Create* and *Query* (see Table 16) methods to get an RO object. Through the concepts of OML, which concern the creation, deletion, modification, reference, and relationship, the definition of RO Factory interface is enough for CORBA server objects. The CORBA server object serves as an agent to serve the transactional operations from clients. The main difference from the ODMG OML is the relationship of objects. We provide three methods to get/set/clear the related objects for each object when needed.

In the next step of application development in the integration architecture, the developers use the IDL compiler to generate codes for the CORBA programming model. After compiling the IDL file, CORBA clients could write their applications in CORBA environment based on the generated stub and header files. We show a simple CORBA client program in transaction style in Table 13.

The client first binds the necessary servers to serve the further requests. For example, the client binds the OTS and RO_Server (see Table 14) in the beginning and then calls the *Create* operation to get a server object reference. The RO Factory for the returned server object will automatically register the object to OTS. After getting the server object reference, the following transactional operations will affect the state of the server object. In the client's point of view, the operations will directly effect the state of the OODB objects in the single-level store model. Finally, the client commits the transaction. The detail interactions among the client, RO_Factories, RO objects, OTS, and OODB are shown in Fig. 13.

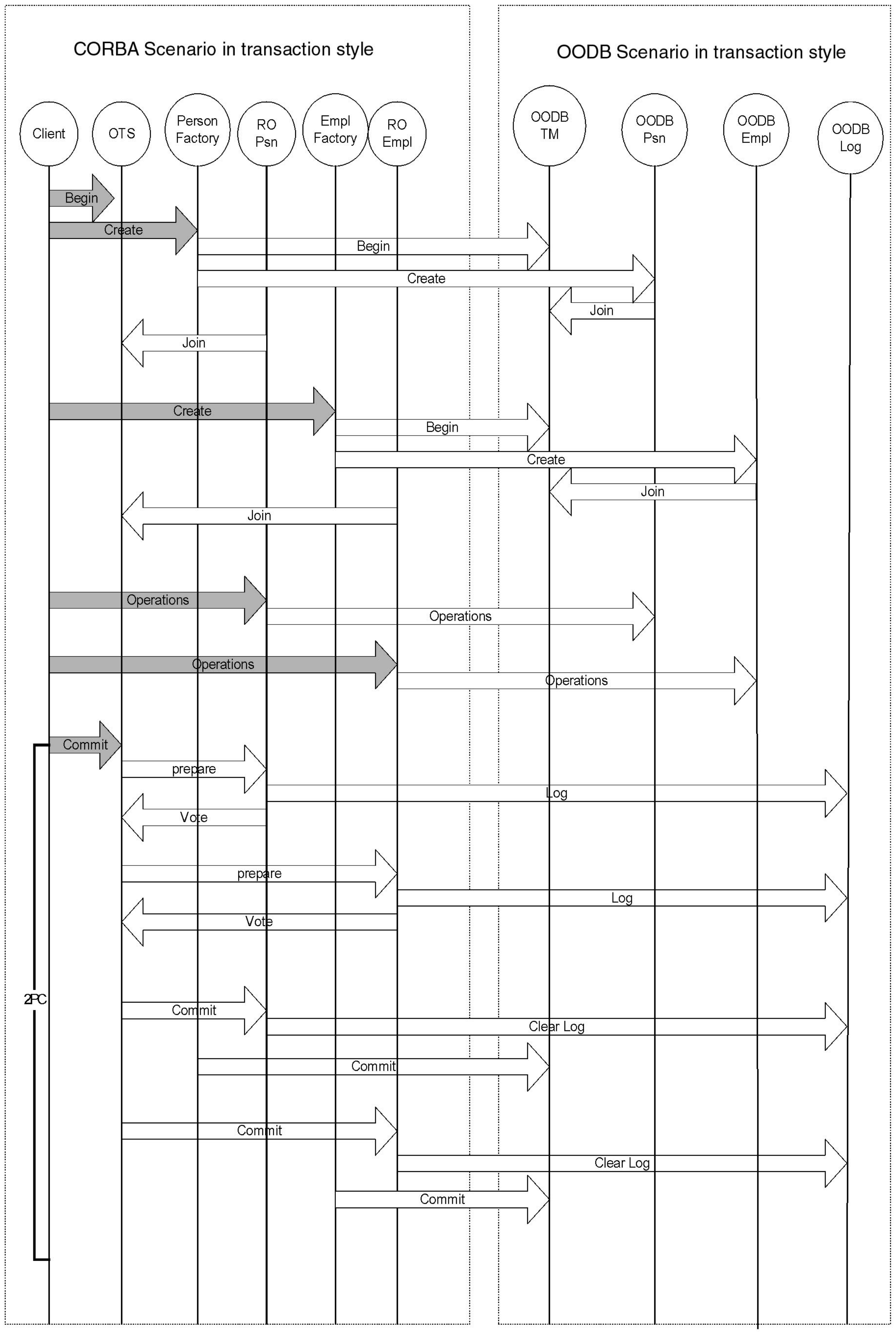

Fig. 13. The detail interactions in a transactional program.

TABLE 15
The Prototype Implementation of the *Create* Method for the RO Factory

```
//Constructor of PersonFactory
PersonFactory::PersonFactory
{
 Count == 0 ; // the unmber of its created objects
 WOOSystem::Initialization(); // startup the connection to OODB
 Transaction.begin();  // begin a DB transaction
 Database.open(database_name);
}

CORBA_Account::commit()
{
  if((Association_table_count --)==0){
     Transaction.commit();  // commit a DB transaction
    Association_table_count = num; // num is the total number of objects created by this factory.
  }
}

CORBA_Personptr
PersonFactory::Create(CosTRansaction::Coordinator Tx_CO)
{
  public:
  Ref<Account> account; //OODB object reference
  PersonResource AR = new PersonResource;
  account = new (&OB) Account; // create a new Account object in database OB
  CORBA_Person_ptr corba_account = new CORBA_Account;
  corba_account->set_reference(account);
  corba_account->fetch(account);
 Tx_CO->register_resource(AR);
 _duplicate(corbaaccount);
  return  corba_account;
}
```

TABLE 16
The Prototype Implementation of the *Query* Method for the RO Factory

```
Collection
PersonFactory::Query(QueryString, CosTransaction::Coordinator Tx_CO)
{
  Collection tmp = new Collection;
  PersonResource AR;
  oql(tmp,QueryString); // query interface for OODB.
  while((tmp.is_empty())!=0){
     if( Account == (tmp)->lookup())
     {
       // check if the object has been queried to keep the consistency
     }
    else{
       tmp.insert_element(Account);
       Tx_CO->register_resource(AR); // register the resource object to OTS
     }
  }

  Collection::_duplicate(tmp) // duplicate the reference count
  return tmp;
}
```

From the scenario in Fig. 13, we can see explicitly that Factory maps the CORBA transaction to the proper OODB transactions managed by OODB transaction manager. In 2PC protocol, the logged object in OODB will help the recovery routines in the recovery phase. In the beginning, the client starts a CORBA transaction and then invokes the *Create* method (see Table 15) for the bound RO Factory reference. When RO Factory receives the *Create* request, it starts an OODB transaction for its first request and creates a corresponding OODB object, as well as registers the created object to OTS to join the CORBA transaction context. At the same time, the created OODB object will automatically join the OODB transaction through the OODB programming language binding. In 2PC protocol, the recovery point is at

TABLE 17
The Implementation of the CORBA_Person Server Object

```
// Implementation of CORBA_Person class defined in IDL
// method to set the value of attribute name
void CORBA_Person_i ::name(const  char* Person_name){
/* … */
}
// get the value of attribute name.
char * CORBA_Person_i::  name(){
/* … */
}
…
…
// get the relationship
CORBA_Employee_ptr CORBA_Person_i:: get_coupleEmpl_relation{
  // concurrency control
  for( i=0;i<Employee_Factory::Count;i++)
  {
if((Person_association_table[i].TO->get_handle())->coupleEmpl==
  Employee_association_table[i].TO->ger_reference){
  CORBA_Employee::_duplicate(Employee_association_table[i].TO);
      return Employee_association_table[i].TO;
}
  // construct the new CORBA_Employee and register it to OTS
    CORBA_Employee_ptr tmp = new CORBA_Employee_i;
    CORBA_Employee_Resource ERtmp = newCORBA_Employee_Resource_i;
    Tx_CO->register_resource(ERtmp);
    tmp->fetch(recovery_obj->dirty->couple);
    Employee_association_table[++EmployeeFactory::Count].TO = tmp;
    CORBA_Employee::_duplicate(tmp);
    return tmp;
   }
}

// set the relationship
void CORBA_Person_i :: set_coupleEmpl_relation (CORBA_Employee aemployee){
    this.set_relationship(aemployee->get_relationship());
}
```

the time when *prepare* method is called. In prepare and commit phases, the RO object creates a *Recovery Object* to log its states in OODB for consistency checking after the occurrence of failures.

We provide a shared server for each client process. All the subsequent requests to the server are served in the same process. The main program is one of the automatically generated files of ODL preprocessor. The variable *<class_name>_i* means that it is the implementation of server object. While the CORBA clients send requests to bind RO Factory, this server process will be activated by the Object Adapter and return the bound server object to clients. The following tables are the implementation for the *CORBA_Person_i* class (see Table 17), which implements the CORBA object and serves as an agent to the OODB object implementation of Person object defined in ODL.

After the preprocessor automatically generates files, including the server main program, the RO Factory, RO object implementation, and the necessary header files, the OODB developers can compile the generated source codes with their handwritten OODB object implementation codes to create the RO. When everything in the server side is ready, the CORBA client AP developer can run the applications to get the services in the CORBA environment.

## ACKNOWLEDGMENTS

We are grateful for the many excellent comments and suggestions made by the anonymous referees. We also thank the Institute for Information Industry of the Republic of China for their financial support through the project, No. 787143.

## REFERENCES

[1] *Object Management Architecture Guide,* Object Management Group Inc., OMG TC Document 92.11.1, Rev. 2.0, Sept. 1992.
[2] *Common Object Request Broker Architecture and Specification,* Object Management Group Inc., Rev. 2.2, Feb. 1998.
[3] *Common Object Services Specification,* Object Management Group Document 1996-7-15, 1996.
[4] W. Lo, D. Liang, Y.M. Kao, S.M. Yuan, and Y.S. Chang, "A Fault Tolerant Object Transaction Service in CORBA," *Proc. 21st Ann. Int'l Computer Software and Application Conf. (COMSAC '97),* pp. 115-120, Washington D.C., Aug. 1997.
[5] K.C. Liang, S.M. Yuan, D. Liang, and W. Lo, "Nested Transaction and Concurrency Control Services on CORBA," *Proc. 1997 Joint Int'l Conf. Open Distributed Processing and Distributed Platforms, (ICODP '97),* pp. 236-247, Toronto, May 1997
[6] K.C. Liang and S.M. Yuan, "Transaction Programming in CORBA," *Proc. Ninth Int'l Conf. Information Resource Management Assoc. (IRMA '98),* pp. 452-460, Boston, May 1998.

[7] J. Kleindienst, F. Plasil, and P. Tuma, "What We Are Missing in the CORBA Persistent Object Service Specification," *Proc. OOPLSA '96 Workshop,* RL: http://www.infosys.tuwien.ac.at/Research/Corba/archive/special/missing-persistence.ps.gz.
[8] S. Baker, "CORBA and Database: Do You Really Need Both?" *ObjectExpert,* May 1996, URL: http://galaxy.uci.agh.edu.pl/~vahe/index.htm.
[9] W. Kim, *Introduction to Object-Oriented Databases,* Cambridge, Mass.: MIT Press, 1990.
[10] "The Object Database Standard: ODMG 2.0," R.G.G. Cattell, ed., San Francisco: Morgan Kaufmann, 1997.
[11] M.L. Griss and R.R. Kessler, "Building Object-Oriented Instrument Kits," *Object,* vol. 6, no. 2, pp. 71-81, Apr. 1996.
[12] S. Mafeis and D.C. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Comm.,* pp. 62-70, Feb. 1997.
[13] V. Srinivasan and D.T. Chang, "Object Persistence in Object-Oriented Applications," *IBM System J.,* vol. 36, no. 1, pp. 60-87, 1997.
[14] P.A. Bernstein and E. Newcomer, *Principles of Trans. Processing,* Morgan Kaufmann, 1997.
[15] V. Vasudevan and R. Anthony, *Approaches for the Integration of CORBA with OODBs,* URL: http://www.infosys.tuwien.ac.at/Research/Corba/archive/special/ORB_OODB.ps.gz, Aug. 1994.
[16] "Orbix+ObjectStore White Paper," IONA Technologies, Dublin, 1995.
[17] "Orbix+Versant Adapter White Paper," IONA Technologies, 1997.
[18] V. Amirbekyan and K. Zielinski, "What CORBA/ODB Integration Technique to Choose: Adapter vs. Wrapper," *Proc. OOPSLA '97 Workshop,* http://galaxy.uci.agh.edu.pl/~vahe/ad_vs_wr.htm.
[19] "Programming Guide: Orbix 2 Distributed Object Technology," IONA Technologies Ltd., Release 2.0, Nov. 1995.
[20] "Reference Guide: Orbix 2 Distributed Object Technology," IONA Technologies Ltd., Release 2.0, Nov. 1995.
[21] "The Webbased Object Oriented—DataBase C++ References," Inst. for Information Industry, Release 1.0, 1998.

**Ruey-Kai Sheu** received the BS and MS degrees in computer and information science from National Chiao-Tung University, Republic of China, in 1996 and 1998, respectively. He is now a PhD candidate in the Computer and Information Science Department of National Chiao-Tung University. His research interests include distributed system design, databases, and World Wide Web technologies.

**Kai-Chih Liang** received the BS degree in 1994 and the MS degree in computer and information science in 1996, both from National Chiao-Tung University, Republic of China. Currently, he is a PhD candidate in computer and information science at National Chiao-Tung University. His research interests include distributed computing systems, distributed object computing, distributed databases, networking, fault-tolerant computing, the Internet, and real-time systems.

**Shyan-Ming Yuan** received the BSEE degree from National Taiwan University, Republic of China, in 1981, the MS degree in computer science from the University of Maryland—Baltimore County in 1985, and the PhD degree in computer science from the University of Maryland—College Park in 1989. He joined the Institute and Department of Computer and Information Science, National Chiao-Tung University, Hsinchu, Taiwan, Republic of China, as an associate professor in September 1990. He was promoted to professor there in June 1995. His current research interests include distributed system design, fault-tolerant computing, network management, computer-supported cooperative work, multimedia application environments, and intelligent computer-assisted learning in distinct cooperative learning environments. He is a member of the ACM, the IEEE, and the IEEE Computer Society.

**Win-tsung Lo** received the BS and MS degrees in applied mathematics from National Tsing Hua University, Taiwan, Republic of China, and the MS and PhD degrees in computer science from the University of Maryland. He is now an associate professor of computer science at Tung Hai University, Taiwan, Republic of China. His research interests include architecture of distributed systems, data exchange in heterogeneous environments, and multicast routing in computer networks.