CORBA 객체의 지속성을 위한 관계 데이타베이스용 객체 데이타베이스 어댑터의 구현

박우창*

An Implementation of Object Database Adapter on Relational Database Systems for the Persistence of CORBA Objects

Uchang Park

〈요 약〉

CORBA는 분산 시스템의 객체를 관리하기 위한 모형으로 여러 응용에서 장점을 보인다. 그러나 CORBA 객체의 정의에는 지속성을 가정하고 있지 않다. 본 논문에서는 CORBA의 BOA(Basic Object Adapter)에 객체 데이타베이스 어댑터(Object Database Adapter, ODA)를 구현하여 객체를 관계 데이타베이스에 저장함으로써 객체의 지속성을 유지하도록 하였다. 객체의 관리는 객체지향 데이타베이스 시스템이 자연스러우나 본 논문에서는 이미 구축된 관계 데이타베이스 시스템과 응용을 공유할 수 있도록 관계 데이타베이스용을 구현하였다. ORB는 Orbix를 사용하였고, 튜플의 객체로의 대응은 Tie 법과 객체 포장기 개념을 사용하였다. CORBA 응용의 관점에서 데이타베이스 시스템을 이용한 객체의 지속성 유지는 지속성 유지와 더불어 많은 수의 객체를 효율적으로 관리할 수 있고, 병행성, 회복기법 등을 이용할 수 있는 여러 장점이 있다. 또 데이타베이스 시스템 측면에서 CORBA와 데이타베이스 어댑터의 이용은 사용자에게 데이타베이스 스키마에 관한 사항을 숨김으로써 데이타베이스 접근을 쉽게 하고, 분산된 데이타베이스 환경에서는 데이타베이스 모델과 언어의 이질성을 해소할 수 있는 장점이 있다.

* 덕성여자대학교 전산학과

1. 서론

분산 시스템의 표준 모델인 CORBA (Common Object Request Broker Architecture)는 객체지향 개념에 기초한 모델이다. CORBA는 분산 환경에서 객체지향 기술을 표준화하기 위한 컨소시엄인 OMG(Object Management Group)[1,2]에 의하여 개발되었다. CORBA는 객체를 캡슐화 하여 데이타와 그 구현에 관한 사항을 숨김으로써, 분산 환경에 서로 다른 언어와 시스템들을 한 개의 시스템으로 운영할 수 있게 한다. 현재 CORBA는 거의 모든 플래폼에 구현되어 분산 시스템의 표준 환경으로 자리 잡아가고 있다.

모델에 의하면 객체는 캡슐화된 데이타와 데이타에 관한 연산으로 정의되며, 이 연산을 통하여 클라이언트의 요구에 응답을 함으로써 분산 환경에서 시스템이나 언어에 의존하지 않는 독립된 서버나 클라이언트를 구축할 수 있다. 객체는 IDL(Interface Definition Language)에 의하여 기술되고 객체에 제공된 연산은 외부에 인터페이스 기능을 한다.

많은 분산 시스템에서 CORBA 객체는 객체를 호출하는 프로세스보다 더 오랜 지속성을 요구하지만 CORBA 객체는 기본적으로 지속성을 지원하고 있지 않다. OMG의 CORBA 2.0에서는 POS라는 부가 서비스 명세를 통하여 CORBA 객체의 지속성을 유지하는 방법을 제안하고 있지만[2], 그 구현에 관한 세부 사항은 제시되어 있지 않다.

객체의 지속성(persistence)은 객체를 파일이나 데이타베이스에 저장함으로써 가능하다. CORBA의 기본적인 어댑터(adapter)는 BOA (Basic Object Adapter)이지만, 객체와 데이타베이스의 연결에 관한 사항을 또 다른 어댑터인 ODA(Object Database Adapter)에서 기술하고 있다[3]. 본 논문의 어댑터는 BOA 기능의 일부로 CORBA 객체를 관계 데이타베이스에 기록함으로써 객체의 지속성을 유지하고자 한다.

CORBA 객체의 지속성 유지는 지속성 유지와 객체의 관리 등 여러 가지 목적으로 객체지향 데이터베이스와 연결이 시도되고 있다. 아직 실험용이나 상용화를 목적으로 한 제품은 IONA의 Orbix+ObjectStore Adapter[4]가 있다. OOSA는 CORBA 객체를 Ostore[5] 데이타베이스에 저장한다. 지속성에 관한 객체지향 데이타베이스와 CORBA와의 결합은 Francisco[6]에 의하여 연구된 바 있다. 객체지향 데이타베이스의 객체는 CORBA 객체로 쉽게 참조될 수 있기 때문에 지속성 유지를 위한 구현이 쉽다. 본 논문은 객체지향 데이타베이스보다는 관계 데이타베이스가 기존의 데이타베이스 응용을 많이 구축하고 있는 점을 고려하여 CORBA와 관계 데이타베이스를 연결하는 관계 데이타베이스용 ODA를 구현하고자 한다.

ORB는 Iona의 Orbix[4]를 사용하였고 ODA는 객체를 관계 데이타베이스의 튜플로 저장하는 객체 포장기 개념을 이용하였다. IDL 인터페이스와 구현 클래스의 접합은 Tie 법을 이용하였다.

2절에서는 CORBA와 CORBA 객체의 지속성에 대하여 설명한다. 3절에서는 지속성을 유지하기 위한 객체 데이타베이스 어댑터의 개념과 관계 데이타베이스와의 연결에 관하여 설명한다. 4절에서는 객체 데이타베이스 어댑터를 CORBA 지속성 서비스와 비교하고 여러 장점들에 대하여 설명한다.

2. CORBA 객체와 지속성

2.1. CORBA 와 BOA

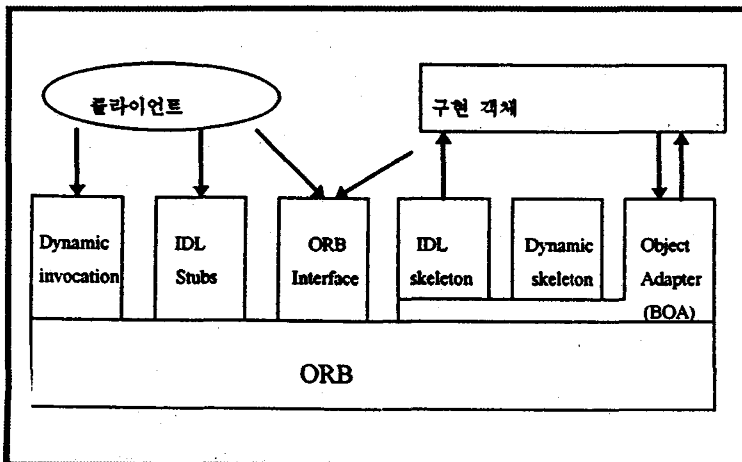
분산 환경을 위한 CORBA 모델은 다음과 같이 구성되어 있다. 그 중요한 부분은 다음의 세가지로 설명할 수 있다(그림 1).

- IDL(Interface Definition Language) - 객체의 인터페이스를 기술하는 언어이다.
- ORB(Object Request Broker) - 객체의 관리와 호출에 대한 처리를 한다.
- BOA(Basic Object Adapter) - ORB와 객체의 구현에 관한 중재를 담당한다.

IDL은 객체의 데이타 멤버와 연산을 기술하는 언어로 기술된 내용을 통하여 클라이언트에게 객체의 내부에 관한 사항을 숨기면서 객체를 접근하도록 한다. 객체가 구현된 프로세스를 서버(server)라고 하고 객체에 대한 메소드를 호출하는 프로세스를 클라이언트(client)라고 한다. 객체를 기술하는 단위를 인터페이스(interface)라고 하며 인터페이스는 프로그래머의 구현 언어에 의하여 구현 클래스로 정의된다. 이렇게 정의된 객체의 인터페이스는 객체를 구현하는 언어와 객체를 접근하는 클라이언트 구현 언어간에 독립성을 제공하여 분산 환경에서 언어의 상호작동성(interoperability) 을 유지시킨다. IDL언어와 C++, Smalltalk 언어간의 대응은 OMG에 의하여 표준으로 제시되어 있고, 그 외 COBOL, Java, Objective_C, Eiffel, Common Lisp 등 여러 언어간의 대응도 정의되어 가고 있다. IDL에 의한 인터페이스는 객체의 연산과 입출력 인수, 결과의 데이타 타입 등이 기술되며, 객체간의 상속성도 기술된다. C++, Java, Smalltalk등 IDL 컴파일러가 개발되어 있으며, IDL의 컴파일 결과는 서버와 클라이언트의 구현을 위한 stub을 제공한다.

ORB는 클라이언트와 서버를 연결하여 클라이언트의 요구를 서버에 전달하고, 서버의 응답을 클라이언트에게 전달한다. 그 중요한 기능들은 다음과 같다.

- 객체의 참조를 만들고 해석한다.
- 서버를 시작하고 종료한다.
- 서버의 각 객체를 시작하고 종료한다.
- 클라이언트의 요구를 서버에게 전달하고 메소드나 인수를 해석하여 해당되는 메소드를 호출한다. 또 서버의 수행 결과를 클라이언트에게 전달한다.



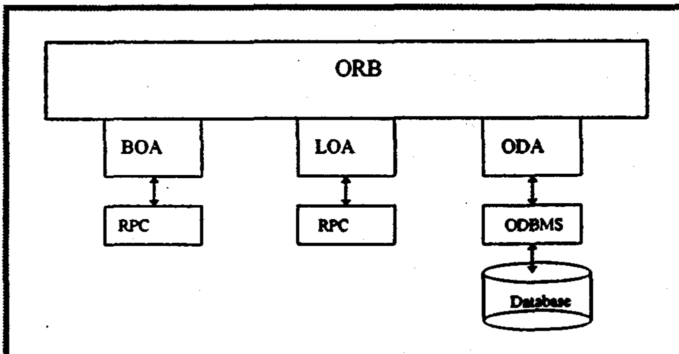
〈그림 1〉 CORBA 모델

BOA는 ORB로부터 요구를 전달 받아서 서버의 시작과 종료를 관리하며 클라

이언트가 요구하는 각 객체를 활성화 시킨다. ORB와 BOA에서 객체의 활성화 과정은 다음과 같다.

1. 서버를 활성화 시킨다.
2. 서버를 등록한다 - 서버가 시작된 다음 impl_is_ready() 함수를 통해서 BOA에 준비됐음을 알린다.
3. 객체를 활성화 시킨다.
4. 메소드를 호출한다.
5. 서버는 필요한 BOA 서비스를 통해 메소드에 응답한다.

BOA는 ORB에서 제공하는 서비스를 사용자에게 제공하는 인터페이스 역할을 하지만 ORB가 제공하지 않는 서비스를 구현하여 제공할 수 있다. CORBA의 명세에는 그 예로서 LOA (Library Object Adapter)와 객체지향 데이타베이스 시스템을 위한 객체 데이타베이스 어댑터 (ODA, Object Database Adapter)를 제시하고 있다(그림 2). 각 어댑터는 서로 대치할 수 있게 정의되어 있지만 본 논문에서는 BOA를 대치하는 ODA를 구현하지 않고 BOA에 데이타베이스를 연결하는 기능을 추가하는 ODA를 구현하고자 한다.



〈그림 2〉 OMG ORB 구조의 ODA 제안

2.2. 지속성

객체의 지속성(persistence)은 OMG의 정의에 의하면 "객체는 객체를 만든 프로세스보다 더 오래 지속될 수 있다. 지속성을 지닌 객체는 명시적으로 지우지 않는 한 계속 존재한다"고 설명하고 있다[2]. 또 객체 참조(object reference)의 지속성은 "객체 참조를 가진 클라이언트는 객체를 구현한 서버가 비활성화되있거나 시스템이 재가동되어도 객체 참조를 통해서 객체를 접근할 수 있어야 한다."고 설명하고 있다. 객체나 객체 참조가 프로세스보다 더 오래 지속되려면 객체를 2차 기억장치인 파일이나 데이타베이스에 기록해야 한다. 또 객체 참조의 지속성은 CORBA 객체의 객체 참조를 통해서 데이타베이스의 객체와 대응시킬 수 있는 방법이 필요하다. 본 논문에서는 객체 참조의 지속성보다는 객체의 지속성에 초점을 맞추고자 한다. CORBA 객체의 지속성을 유지하기 위한 CORBA 객체와 데이타베이스 객체의 연결은 두 가지의 구현 방법을 생각해 볼 수 있다. 첫째는 CORBA 객체가 데이타베이스 객체를 접근함으로써, CORBA 객체가 지속성을 갖는 것처럼 구현하는 방법과 둘째는 CORBA 객체를 직접 데이타베이스에 저장하는 방법이다. 첫째 방법은 기존의 데이타베이스 응용 인터페이스(API)가 데이타베이스를 접근하는 것과 같은 방법으로 구현하면 됨으로 문제가 되지 않는다. 그러나 CORBA가 객체지향 모델인 점을 고려하면, 객체의 캡슐화, 상속성 등 객체지향 개념의 특성들을 상실하게 됨으로 둘째 방법으로 구

현하여야 한다.

둘째 방법의 구현은, 먼저 CORBA 객체를 그대로 기억장소에 저장함으로써 객체의 지속성을 유지하는 방법을 생각해 볼 수 있다. 이 방법은 다음과 같은 점에서 문제점이 생긴다. 첫째, CORBA 객체는 단순한 상태 값을 가지는 것이 아니라 IDL에서 정의된 상위 인터페이스의 데이타, 함수, 또 CORBA::Object에서 상속된 데이타 등을 포함하기 때문에 데이타베이스의 기억장소를 많이 차지한다. 둘째로, CORBA 객체는 참조 변수를 갖고 있어서 참조되면 증가하고 해지되면 감소한다. 이런 참조 변수를 데이타베이스에 저장하면 데이타베이스 참조 시간에 의해서 ORB의 응답 시간이 많이 걸리게 된다.

본 논문에서는 위와 같은 직접적인 방법을 피하여 CORBA 객체를 데이타베이스에 저장하지 않고, 주 기억장치에 저장하는 대신, CORBA 객체의 클래스 정의와 객체의 상태 값만을 지속성을 갖는 기억장소에 저장한다. 객체가 생성되거나 참조되면 참조 변수와 객체의 대리 객체가 기억장소에 생성되며, 객체의 메소드가 호출되면 메소드는 구현 객체의 메소드를 사용하지만 객체의 상태 값은 대응되는 객체의 데이타베이스 값을 참조한다. 본 논문에서는 IDL 인터페이스 구현 방법 중 대리법(delegation)을 쓴다.

3. 객체 데이타베이스 어댑터의 구현

3.1. IDL의 구현 방법

CORBA는 객체를 기술하는 언어로 IDL (Interface Definition Language)을 제공하고 있으며, IDL 언어의 인터페이스(interface)는 객체를 기술하는 단위가 된다. IDL에서 정의된 인터페이스는 프로그래머의 구현 언어에 의하여 구현 클래스와 연결된다. 인터페이스와 구현 클래스를 연결하는 방법은 상속법(inheritance)과 Tie로 알려진 대리법(delegation)이 있다. Iona의 Orbix는 이 두 가지를 구현하고 있다. 이 두 방법은 서버를 구현하는 방법이며 클라이언트의 구현은 영향을 받지 않는다.

상속법은 구현 객체가 인터페이스의 속성을 상속 받음으로써 구현된다. 예를 들어 프로그램 1의 IDL을 상속법에 의하여 구현한 구현 클래스의 C++ 언어는 프로그램 2와 같이 된다. 클래스 A_inherit 은 클래스 A의 하위 객체이고 클래스 A에서 상속 받는다. 상속 받지 않는 메소드는 A_inherit 클래스에 추가한다.

프로그램 1 : 인터페이스 A의 IDL

```
interface A {
        short op1();
        void op2(in long l);
};
```

프로그램 2 : 상속법에 의한 인터페이스 A의 구현

```
class A_inherit : public A {
public :
        CORBA::short op1();
        void op2(CORBA::Long l);
};
```

대리법은 IDL 컴파일러에 의해 생성된 Tie라고 부르는 구현 클래스의 대리 클래스를 사용하는 방법이다. 이 클래스는 구현 클래스 객체를 가리키는 포인터를 가진다. 프로그램 1은 인터페이스 A이고 프로그램 3는 Tie 클래스 tie_A이다. tie_A()는 생성자 이다. Tie 클래스 객체의 데

이타 ref는 구현된 클래스 객체의 포인터 값을 갖고, 함수는 구현된 함수의 주소 값을 갖는다.

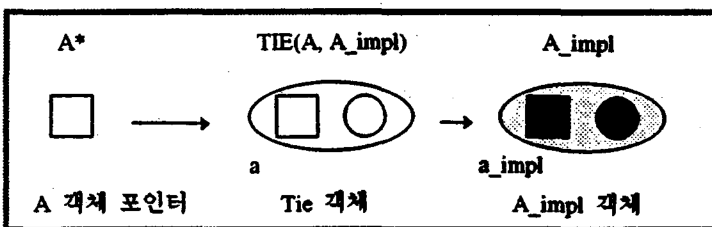
프로그램 3 : IDL이 생성한 인터페이스 A의 Tie 클래스

```
template <class Impl>
class tie_A: public A {
private :
        Impl& ref;
public :
        tie_A(Impl& impl_obj) : ref(impl_obj){ }
        CORBA::short op1() { return ref.op1(); }
        void op2(CORBA::Long l) { ref.op2(l); }
};
```

구현 객체는 컴파일 시간때 A_impl 클래스와 tie가 되며, 다음 문장에 의하여 수행 시간에 <그림 3>과 같이 Tie 객체 a_obj 와 구현 객체 a_obj가 연결된다.

```
tie_A<A_impl> a_obj
      = tie_A<A_impl>(a_impl);
```

Tie 클래스는 tie_A는 인터페이스 A를 대신하며, 따라서 Tie 객체 참조 a의 메소드 a->op1()은 a_impl->op1()이 된다. 객체 데이타베이스 어댑터의 구현은 대리법을 쓴다. 구현 객체가 인터페이스 객체보다 먼저 생성되고 또 오래 지속성을 유지하려면 IDL 인터페이스의 상속 체계를 벗어나서 독립적으로 구현이 가능하여야 한다. 또 CORBA와 관계없는 응용도 구현 객체를 접근하려면 구현 객체는 대리법에 의하여 인터페이스와 독립적으로 구현되어야 한다.



<그림 3> Tie에 의한 객체 A의 구현

3.2. 객체-관계 대응

CORBA 명세에 의하면 ORB와 데이타베이스의 연결은 객체지향 데이타베이스 시스템을 중심으로 기술되어 있다. CORBA와 객체지향 데이타베이스가 객체 중심의 시스템이기 때문에 두 시스템의 연결은 자연스럽다. 그러나 현재 관계 데이타베이스가 널리 사용되고 있는 점을 고려하면 ORB와 관계 데이타베이스의 연결이 중요하다. 기존의 관계 데이타베이스의 객체지향 접근은 객체 포장기(object wrapper)를 이용한다. 객체 포장기를 이용한 관계 데이타베이스의 객체화 시스템으로는 Persistence, DBconnect[7,8,9] 등이 있으며, 기존의 관계 데이타베이스 시스템들도 이러한 개념을 도입하여 데이타베이스의 모델 능력을 확장하고 있다. 기존의 객체 포장기는 상용 데이타베이스에 연결되어 객체와 객체의 스키마를 대응시켜 관계 데이타베이스를 접근하도록 하나, 본 논문에서는 간단한 형태의 객체-관계 대응기를 구현하였다.

본 논문에서는 관계 데이타베이스의 튜플을 CORBA의 객체 단위로 대응시킨다. 관계 데이타베이스의 튜플은 객체의 식별자에 해당되는 값을 가지고 있지 않기 때문에 따로 식별자에 해당되는 속성을 추가

한다.

관계 데이타베이스의 튜플을 객체로 대응시키기 위하여 포장기 클래스(wrapper class)를 정의하였다. 본 논문에서는 Object-Tuple이라고 이름 붙였다. 여기서 클래스 Object-Tuple은 관계 데이타베이스의 튜플을 구현하기 위한 모형(Template)클래스이다. 객체의 메소드는 관계 데이타베이스에 없으므로 객체의 상태 값과 새로 생성된 튜플의 식별자만 관계 데이타베이스에 저장한다. 식별자의 유일성은 Object-Tuple 클래스에 의하여 관리된다. 클래스 Book을 예로 들면 다음과 같다.

(객체 클래스 Book)

```
class Book : {
private :
char        bookname[20];
char        bookauthor[20];
public :

            (methods)
}
```

(관계 데이타베이스의 릴레이션 Book)

```
Book(
tuple_id int;
bookname char(20);
bookauthor char(20);

}
```

릴레이션 book의 튜플이 Object-Tuple<Book>에 의하여 객체로 포장되는 방법은 다음과 같다. 예를 들어 객체 포인터 x에 해당하는 bookname을 읽는다면(튜플의 식별자 값은 ID), 관계 데이타베이스에서 ID 값을 가진 튜플을 찾아 Book 객체를 생성하고 메소드 bookname()을 적용한다. 이 때, 그림 4에서 보는 것처럼 Object-Tuple<Book>의 참조는 참조 연산자 ->가 클래스 Book을 참조하도록 오버로딩되어 있다. 여기서 pointer->bookname()의 "->" 연산자는 이 기능을 수행하도록 Object-Tuple에서 연산자 오버로딩 기법을 이용한 것이다.

(객체 Object-Tuple<Book>의 접근)

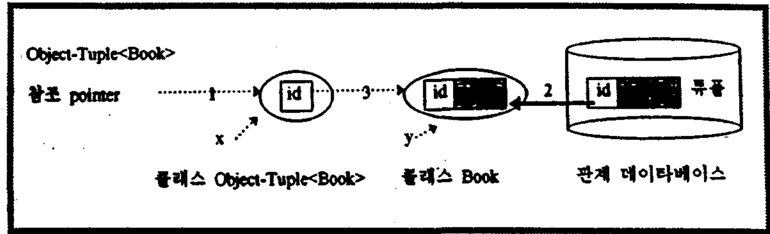
```
Object-Tuple<Book>* pointer;
pointer = x;
/*연산자 ->의 실행*/



pointer->bookname();
```

(객체 Object-Tuple<Book>의 접근의 실행)

```
Object-Tuple<Book>* pointer;
pointer = x;
{       select (bookname, bookauthor) from Book
        where tuple_id = ID;
        p = new Book(bookname, bookauthor);
        pointer= p;
}
pointer->bookname();
```

이 관계를 그림으로 표시하면 위와 같다(그림 4)

1. 객체 x가 참조된다.
2. 객체 y가 로더에 의하여 생성된다.
3. 참조 y는 Object-Tuple의 연산자 ->의 오버로딩에 의하여 참조 x를 대신한다.

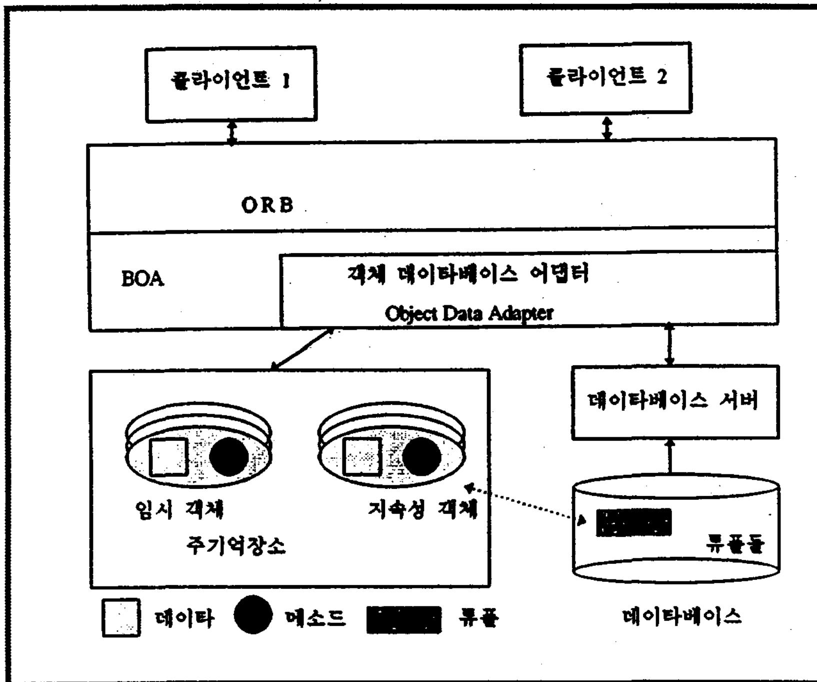


〈그림 4〉 수행 시간 시 튜플의 객체화

3.3. 객체 데이타베이스 어댑터의 구조와 작동

OMG의 정의에는 ODA는 대해서는 여러 대치 가능한 어댑터 중 하나로 제시되어 있다. 본 구현에서는 BOA를 대치하는 것이 아니라 BOA에 ODA에 추가함으로써 구현하였다. 구현 객체의 상태는 데이타베이스에 저장이 되고 객체의 참조가 일어날 때 Tie 법에 의하여 주기억장치에 객체가 생성된다. ODA와 BOA의 관계는 다음과 같다(그림 5).

CORBA 객체를 관계 데이타베이스에 연결하는 방법은, 본 논문에서는 CORBA 객체가 직접 지속성을 유지하는 방법을 사용하였다. 또 모든 CORBA 객체를 데이타베이스에 저장하지 않고 객체의 상태 값만을 데이타베이스에 저장하여 객



〈그림 5〉 ODA에 의한 객체의 지속성 관리

체의 서버 객체를 이용하여 객체를 접근하는 방법을 사용하였다. 이 과정에서 다음과 같은 기능들이 충족되어야 한다.

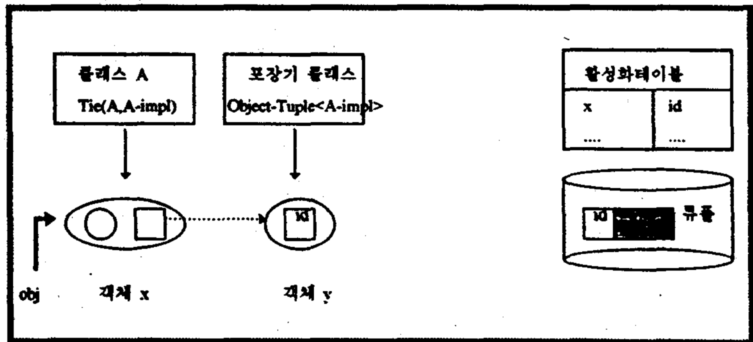
1. CORBA 객체의 정의는 관계 데이타베이스의 스키마와 대응되어야 한다.
2. 관계 저장된 객체는 식별자를 갖고서 이 식별자를 이용하여 튜플을 접근할 수 있어야 한다.
3. CORBA 객체가 참조되면 데이타베이스에서 주 기억장치로 객체가 생성되어야 한다.
4. CORBA 객체의 상태 값이 변하면 데이타베이스 객체의 상태 값도 변해야 한다.

상용 객체-관계 데이타베이스의 기능 중 일부는 위의 과정의 일부를 구현한 것들이 있다. Persistence[8]는 위 과정 중 2번의 기능을 구현하였으며, 도 ObjectStore의 OOSA[[4]는 1번과 4번의 기능을 수행한다.

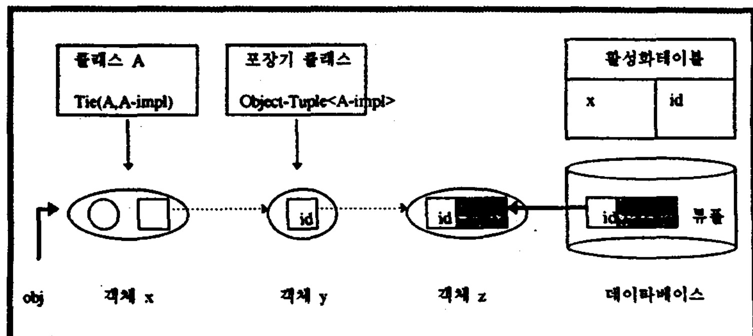
본 논문의 ODA의 구현 과정을 위와 같은 상용 데이타베이스 기능과 결합하여 구현하는 것도 고려할 수 있다. 본 논문에서의 ODA는 위의 기능 중 2, 3, 4번을 수행하도록 구현하였으며, 1번은 프로그래머가 직접 대응을 시켜야 한다. ODA의 주요 기능은 다음과 같다.

1. 지속성 객체를 위한 Tie 클래스를 정의한다.
2. 서버 프로그램의 지속성 객체 참조를 처리한다(3번 기능).
3. 클라이언트의 지속성 객체 참조를 서버의 지속성 객체와 연결시킨다(4번 기능).
4. CORBA 객체의 식별자와 관계 데이타베이스의 튜플 식별자에 관한 테이블을 갖고 있다. 객체에 대한 참조가 일어나면 이 테이블을 참조하여 데이타베이스를 접근하여 객체를 활성화 시킨다.(2번 기능)
5. 활성화된 객체의 테이블을 참조하여 객체의 수가 많아지면 선입선출 순으로 객체를 해지(release)시켜서 클라이언트 응용이 객체의 관리를 하지 않아도 객체를 비활성화 시키도록 한다.

지속성 객체에 관한 연산은 다음과 같이 행해



〈그림 6〉 객체의 생성(A* obj = new A_impl())



〈그림 7〉 객체의 참조(obj ->)

진다. 앞에서 예로 든 객체 A를 이용하여 객체 어댑터의 작동을 설명하면 다음과 같다.

1. 객체의 생성(그림 6) : 지속성 객체가 생성이 되면 객체의 Tie 객체 x가 생기며, 객체의 데이타 부분은 관계 데이타베이스에 저장된다. 이때 관계 데이타베이스의 튜플에는 튜플의 식별자가 생긴다. 이 식별자는 ODA의 식별자 생성기에 의하여 유일한 값을 생성한다. 인터페이스 A를 참조하는 포인터 obj가 새로운 객체를 생성하면 obj는 Tie 클래스 객체 x를 가리킨다. Tie(A,a) x는 ref 변수에 객체 포인터 y 값을 갖는다. 객체 y는 A-impl 식별자 값 id를 갖는다. ODA는 활성화 테이블에 (x,id)를 저장한다.

2. 객체의 참조(그림 7) : 클라이언트에 의하여 객체가 참조되면 객체 식별자에 의하여 ODA의 참조 테이블을 검색한다. 만약 객체가 활성화 되어 있다면 이 객체를 이용하여 클라이언트에게 응답한다. 객체가 활성화 되어 있지 않다면 어댑터를 이용하여 앞에서 설명한 객체를 생성하는 과정을 거친다. load() 함수는 CORBA::LoaderClass에서 상속되는 함수로 객체가 활성화되어 있지 않을 때 호출이 된다. 데이타베이스의 해당되는 id의 튜플을 객체 z로 생성하면, z의 값이 obj의 값으로 대응된다.
3. 객체의 참조 해지 : ODA는 활성화된 객체의 활성화 테이블을 갖고 있어서, 시스템의 제한에 따라 객체를 해지(release) 함수에 의하여 비활성화 시킨다. 비활성화 대상 객체는 선입선출 순으로 ODA에 의하여 리스트로 관리된다. 객체가 비활성화되면 x, y, z 모두 없어진다.

4. 객체 데이타베이스 어댑터의 구현 결과

4.1. 응용 프로그램에서의 어댑터의 사용

본 절에서는 ODA가 실제 응용 프로그램에서 사용되는 예를 들어 보기로 한다. 예는 간단한 도서관리 프로그램으로 서버는 C++ 언어로 구현하였다. IDL 인터페이스는 author와 title 데이타로 구성된 객체 book과 객체 book의 서버 객체로서 book에 관한 메소드를 정의하는 library 객체로 구성되어 있다. 프로그램 4는 IDL 인터페이스이다.

프로그램 4 : 인터페이스 book과 library의 IDL

```
typedef string<20> Name;
interface book {
    readonly attribute Name author;
    readonly attribute Name title;
};
typedef sequence<book> bookList;
interface library {
    readonly attribute Name libname;
    void add_book(in Name name, in Name author);
    bookList book_list();
}
```

인터페이스 book과 library를 구현하는 구현 객체의 Tie 객체는 헤더 파일 library.h에 의하여 다음과 같이 구현된다. library.h의 RODA__def__server()와 RODA__def__ persistent() 문장은 ODA의 마크로 문으로 인터페이스와 구현 객체를 연결하는 Tie 객체를 정의하는 문장이

며, RODA__def__persistent()는 지속성 객체 book과 구현 객체 Book의 tie 객체 ODA__book__Book 클래스를 정의한다. 또 RODA__def__server()는 지속성 객체는 아니지만 지속성 객체를 관리하는 메소드를 정의하는 서버 객체 library와 구현 객체 Library의 Tie 객체 ODA__library__Library 클래스를 정의한다. ODA의 Tie 클래스 ODA__book__Book의 객체는 포장기 클래스 Object__Tuple<Book> 객체의 포인터를 갖고 있으며 이 객체에 의하여 관계 데이타베이스의 튜플을 접근한다.

프로그램 5 : 헤더 파일 library.h

```
#include "library..hh"    // IDL-generated skeletons
#include "roda.h"         // ODA header files
// implementation class for book
class Book : public virtual Object__Tuple::Object {
public:
        .... implementations of methods
}
// implementation class library
class Library : public virtual Object__Tuple::Object {
public:
        .... implementations of methods
};
// ODA directives
RODA__def__server(library,Library); // top-level
                                     object tie by ODA
RODA__def__persistent(library,Library); // persistent
                                     object tie by ODA
```

한편, 서버의 구현 시 Tie 객체의 접근은 다음과 같이 한다. 프로그램 6는 library 인터페이스의 book__list() 메소드를 구현하는 예이다. book__list()는 모든 Book 자료를 보여 주는 메소드이며, Oracle 관계 데이타베이스를 사용한 예이다. 프로그램 6의 (1)번 문장은 Book 객체를 객체 식별자 oid에 의하여 참조하여 리스트로 관리하는 문장이며. (2)번 문장은 Tie 객체를 참조하는 문장이다.

프로그램 6 : library.cc중 book_list() 함수

```
bookList* Library::book__list(CORBA::Environment &) const
{
    plist->remove__all();
    EXEC SQL BEGIN DECLARE SECTION;
    struct bookdat{ int oid; } bookrec;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE nums__book CURSOR FOR
    SELECT oid FROM book;
    EXEC SQL OPEN nums__book;
    int i = 0;
    EXEC SQL WHENEVER NOT FOUND DO break;
    for( ; ; ) {
        EXEC SQL FETCH nums__book INTO :bookrec;
        plist->insert(Ref<Book>(0, bookrec.oid));…… (1)
        i++;
}
    EXEC SQL CLOSE nums__book;
    EXEC SQL COMMIT WORK ;
    long n__books = plist->cardinality();
    book__ptr* buf = bookList::allocbuf(n__books);
    Ref<Book> pi;
    Cursor< Ref<Book> > c(*plist); i=0;
    for (pi=c.first();c.more();pi=c.next(), i++) {
    buf[i] = roda__persistent__book(pi); …… (2)
    book::__duplicate(buf[i]);
    }
    return new bookList(n__books,n__books,buf,1);
}
```

4.2. POS와의 관계

CORBA의 ORB는 객체의 관리에 관한 기본적인 서비스를 한다. 1995년에 제안된 CORBA 명세[2]는 기본적인 서비스 외에 상위 계층의 서비스에 관한 명세를 제공하고 있다. 그 중 하나가 지속성에 관한 서비스이다. POS는 객체의 동적 상태와 지속 상태의 두 가지 저장 모델을 정의 한다. OMG에서 제안된 방법은 지속성 유지에 관한 모든 명령을 클라이언트가 제어하도록 되어 있다. 클라이언트는 connect, disconnect, store, restore, 그리고 delete 명령을 사용하여 객체의 지속성을 관리한다.

POS의 서버 POM(Persistence Object Manager)이 데이타 저장과 프로토콜의 정의에 관한 PDS(Persistence Data Service)를 관리한다. POS의 하부구조는 여러 구현자 층을 포함하기 위하여 많은 사항을 구현자에게 정의하도록 하고 있다.

본 논문의 객체 데이타베이스 어댑터는 CORBA 객체에 지속성을 부여하는 점은 POS의 기능과 같으나, 객체의 지속성 유지를 클라이언트에게는 투명하도록 하고 있다. 또 한 개의 층으로 구성된 저장 모델로 클라이언트가 특별한 관리를 하지 않고 객체가 관리된다.

4.3. 어댑터의 데이타베이스 관점

데이타베이스 관점에서 CORBA 응용을 접속시키려면 먼저 원자성(atomicity)에 관한 트랜잭션 개념을 포함시켜야 한다. CORBA의 데이타베이스 접근은 보통 원격지에서 접근하며 이때 지역 데이타베이스 트랜잭션과 일치성 문제가 제기된다. 지역 데이타베이스 트랜잭션이 없이 CORBA 객체의 지속성만을 위해서 데이타베이스를 사용한다면 객체는 항상 ODA를 통해서 접근을 하기 때문에 지속성 객체의 조작이 일어나도 일관성 문제가 제기되지 않는다. 그러나 지역 데이타베이스의 트랜잭션이 있을 때, 지속성 객체의 조작 연산을 데이타베이스에 기록하여야 조작 이상이 일어나지 않는다. 또 지역 데이터베이스의 변경은 CORBA 객체의 일치성 문제를 야기하기 때문에, CORBA에 의한 지속성 객체의 참조는 참조가 일어난 시점에 데이타베이스를 읽어야 한다. 본 논문의 데이타베이스 어댑터는 서버를 구현할 때 이 두 가지를 사용자가 구현할 수 있도록 하였다. 트랜잭션의 단위는 객체의 메소드로 하였다.

ODA는 CORBA 객체를 데이타베이스에 저장 함으로써 CORBA 응용이 지속성을 얻고 또 많은 객체를 관리할 수 있다. 반대로 데이타베이스의 응용 프로그램의 개발로서 CORBA의 이용은 데이타베이스 사용자에게 또 다른 장점을 갖다준다. 데이타베이스 클라이언트는 일반적으로 데이타베이스를 접근하려 할 때 데이타베이스가 저장된 논리적 구조인 스키마에 관한 지식이 필요하다. 스키마에 대한 지식을 통해서 조인 등을 통한 스키마간의 관련 데이타를 추출해 낼 수 있다. 그러나 CORBA의 IDL 언어를 통한 데이타베이스 객체의 기술은 IDL언어에 기술되어 객체의 연산을 통해서 스키마에 대한 상세한 접근을 피하면서 데이타베이스를 접근하여 한층 더 높은 데이타독립성을 유지할 수 있다.

CORBA의 분산 환경에서의 응용은 또 분산된 데이타베이스의 이질성을 해소하는 또 다른 기법이 될 수 있다. 분산된 데이타베이스들이 서로 다른 제품이거나 또 같은 제품이더라도 다른 스키

마를 갖거나 하는 경우, 또 각각 다른 언어로 개발된 응용의 경우, IDL을 통한 데이타의 객체화와 일치된 메소드를 제공하게 함으로써 분산 데이타베이스 시스템 환경에서 상호작동을 가능하게 한다.

5. 결론

CORBA 객체의 지속성 유지는 분산 시스템 환경에 필수적인 요소라 할 수 있다. 본 논문은 Iona의 Orbix와 관계 데이타베이스를 이용하여 CORBA 객체의 지속성을 구현하였다. 구현된 데이타베이스 어댑터는 기존의 BOA를 대치하지 않고 부가되어 사용하는 방법으로 Tie 법과 Loader 개념을 이용하였으며 여러 상용 관계 데이타베이스 시스템에 적용되어 사용할 수 있다.

구현된 ODA는 CORBA의 POS에 비하여 지속성 유지가 클라이언트에게 투명할 뿐 아니라, 데이타베이스에 객체를 저장하여 많은 수의 객체를 쉽게 관리할 수 있으며, 데이타베이스가 가지는 병행성, 회복기법, 질의어 등을 이용할 수 있다.

또 기존의 구축된 데이타베이스 시스템의 측면에서 CORBA와 ODA의 이용은 데이타베이스 시스템의 사용 환경을 객체의 개념을 이용하여 자세한 사항을 축약하여, 사용자가 스키마에 대한 지식이 없이 데이타베이스를 접근할 수 있다. 또 분산된 데이타베이스 환경에서 언어나 모델의 이질성을 해소시킬 수 있다.

〈참고문헌〉

[1] Object Management Group, *The Common Object Request Broker : Architecture and Specification*, John Wiley &Sons, Inc, 1991.

[2] Object Management Group, *Common Object Request Broker: Architecture and Specification*, at http://www.omg.org/corbask.html, 1996.

[3] R.G.G. Cattell, *Object Database Standard: ODMG-93*, Morgan Kaufmann, 1993.

[4] Orbix at http://www.orbix.com/Orbix/index.html.

[5] Object Design, Inc at http://www.odi.com.

[6] Francisco Reverbel, *Persistence in Distributed Object Systems: ORB/ODBMS Integration*, Ph.D. Thesis, University of New Mexico, 1996, April.

[7] Object Design, *ObjectStore/DBConnect, Product Brief*, Object Design, Inc., Burlington, MA, 1995.

[8] Keller, A. M., R. Jensen, and S. Agrawal, "Persistence Software: Bridging Object-Oriented Programming and Relational Databases, Proceedings of the 1993 ACM SIGMOD, Washington, DC, May 1993.

[9] Object Wrapping for WWW at http://www.ansa.co.uk/ANSA/ISF/1464/1464prt1.html.