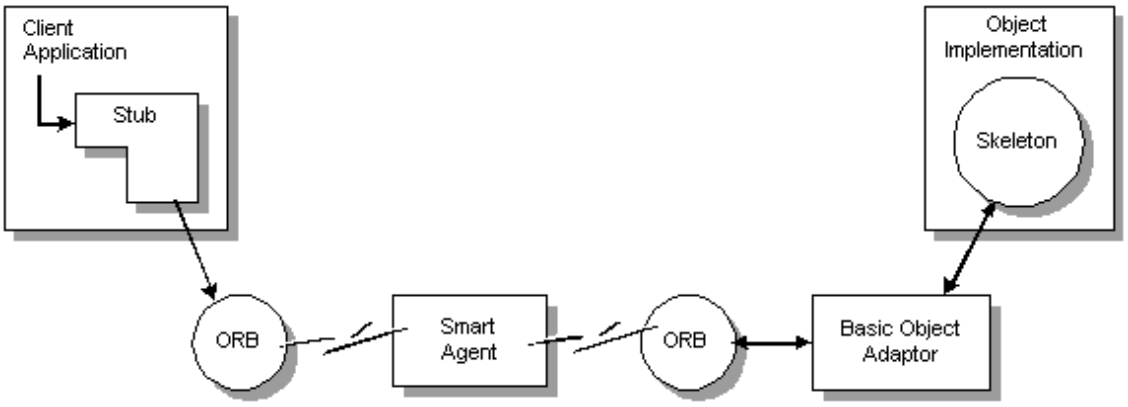
CORBA 의 개념과 활용 (I)

델파이 4 에서는 델파이 3 에서 COM 을 완벽하게 지원하게 된데 이어서 CORBA(Common Object Request Broker Architecture)에 기초한 분산 어플리케이션을 작성하기 쉽도록 여러 가지 위저드와 클래스 들을 제공한다.
CORBA 는 OMG(Object Management Group)에서 채용한 분산 객체 어플리케이션 개발에 대한 표준 스펙으로, 분산 어플리케이션을 제작하는데 객체 지향적인 접근 방법을 제공한다. 이는 인터넷 서버 어플리케이션의 제작에서 HTTP 어플리케이션에 대한 메시지 지향 접근 방법과 비교된다. CORBA 하에서 서버 어플리케이션은 잘 정의된 인터페이스를 기반으로 클라이언트 어플리케이션에 의해 원격으로 사용되는 객체를 구현한다.
CORBA 스펙은 클라이언트 어플리케이션이 어떻게 서버에 구현된 객체와 통신하는지를 정의하는데, 이러한 통신은 ORB(Object Request Broker)에 의해 다루어진다. 델파이 4 에서는 Inprise 의 VisiBroker ORB 를 CORBA 를 지원하는데 사용한다.
클라이언트와 서버 기계 상의 객체 사이의 통신을 가능하게 하는 기초적인 ORB 기술에 추가하여, CORBA 표준은 수많은 표준 서비스를 정의하고 있다. 이러한 서비스 들은 잘 정의된 인터페이스를 사용하기 때문에, 개발자는 비록 다른 업체에서 만든 서비스라 할 지라도 같은 인터페이스를 구현했으면 같은 방법으로 서비스를 사용할 수 있다.
델파이 4 의 클라이언트/서버 버전은 기초적인 ORB 기술을 지원하며, 엔터프라이즈 버전은 레코드 형과 같은 추가적인 데이터 형과 SSL 보안과 같은 CORBA 서비스를 제공한다.
이번 장에서는 CORBA 에 대한 전체적인 개념과 델파이 4 에서 지원하는 CORBA 와 관련된 클래스의 사용 방법에 대해서 알아보도록 할 것이다.

CORBA 어플리케이션의 개괄

CORBA 어플리케이션의 디자인은 다른 객체 지향 어플리케이션과 별반 다를 것이 없지만, 다른 기계에 존재하는 객체와 네트워크 상에서 통신할 수 있도록 추가적인 계층(layer)를 포함해야 한다는 점이 다르다. 이런 추가적인 계층은 스텁(stub)과 스켈레톤(skeleton)이라고 불리는 특수한 객체에 의해 조절된다.
CORBA 클라이언트에서 스텁(stub)은 서버 기계에 실제로 구현된 객체에 대한 프록시(proxy)로 행동한다. 클라이언트는 인터페이스를 구현한 객체와 직접 상호작용 하듯이 스텁에 접근하게 된다.
어쨌든 인터페이스를 구현한 대부분의 객체와는 달리, 스텁은 클라이언트 기계에 설치된 ORB 소프트웨어를 호출하여 인터페이스 호출을 다루게 된다. ORB 는 LAN 상의 어느 곳에서든 동작하고 있는 스마트 에이전트 (Smart Agent, osagent)를 이용하게 되는데, 스마트 에이전트는 다음 그림과 같이 실제 객체의 구현을 지원하는 가능한 서버에 위치하여 동적이

고 분산된 디렉토리 서비스를 하게 된다.



CORBA 서버에서 ORB 소프트웨어는 인터페이스 호출을 자동으로 생성된 스켈레톤에 넘겨주게 되며, 스켈레톤은 BOA(Basic Object Adaptor)를 통해 ORB 소프트웨어와 통신하게 된다. BOA 를 사용하여 스켈레톤은 객체를 스마트 에이전트에 등록하게 되는데, 여기에서 객체의 범위(원격 기계에서 사용될 지 여부 등)와 객체가 인스턴스화 되어 클라이언트에 반응할 수 있게 되는 시기 등을 결정하게 된다.

- 스텁과 스켈레톤의 이해 (Understanding stubs and skeletons)

스텁과 스켈레톤은 CORBA 어플리케이션이 인터페이스 호출을 다음과 같이 마샬링(marshaling)한다.

- 서버 프로세스의 인터페이스 포인터를 가져다가 이 포인터를 클라이언트 프로세스의 코드에서 접근할 수 있도록 해준다.
- 클라이언트로 부터의 인터페이스 호출의 argument 들을 원격 객체의 프로세스 공간에 위치시킨다.

어떤 인터페이스 호출에서도 호출자는 argument 들을 스택에 밀어 넣고, 함수 호출을 인터페이스 포인터를 통해 시도한다. 객체가 인터페이스를 호출한 코드와 같은 프로세스 공간에 있지 않으면, 호출은 같은 프로세스 공간의 스텁을 거쳐 바깥으로 나가게 된다. 스텁은 argument 들을 마샬링 버퍼에 밀어넣고, 원격 객체에 구조체의 형태로 호출을 전송한다. 서버 스켈레톤은 이 구조체를 풀어서 argument 들을 스택에 밀어넣고 객체의 구현부분을 호출한다.
스텁과 스켈레톤은 객체의 인터페이스를 정의할 때 자동을 생성된다. 개발자가 인터페이스를 정의할 때 이들의 정의는 _TLB 유닛에 생성된다.

- 스마트 에이전트의 활용 (Using Smart Agents)

스마트 에이전트는 객체를 구현한 서버에 위치한 동적, 분산 디렉토리 서비스이다. 만약 선택할 서버가 많으면, 로드 밸런싱(load balancing)을 지원한다. 또한, 서버에 접속이 실패하면 서버의 재시작을 시도하거나, 다른 호스트 컴퓨터에 위치한 서버를 실행한다. 스마트 에이전트는 클라이언트와 서버 어플리케이션 양측에 완전히 투명하게 접근할 수 있다.
스마트 에이전트는 LAN 상에서 최소한 하나의 호스트에서 시작되어야 하며, ORB 는 스마트 에이전트의 위치를 브로드케스트 메시지를 보내어 확인한다. 네트워크에 여러 개의 스마트 에이전트가 있으면, ORB 는 먼저 처음으로 반응하는 스마트 에이전트를 사용한다. 일단 스마트 에이전트의 위치가 확인되면 ORB 는 point-to-point UDP 프로토콜을 사용하여 스마트 에이전트와 통신하게 된다. UDP 프로토콜은 TCP 연결에 비해 적은 네트워크 리소스를 사용한다는 장점이 있다.
네트워크에 여러 개의 스마트 에이전트가 있으면, 각각의 스마트 에이전트는 사용가능한 객체를 인식하여 다른 스마트 에이전트의 위치를 직접 연결할 수 있게 되며, 하나의 스마트 에이전트가 비정상적으로 종료될 경우, 객체 들은 자동으로 다른 스마트 에이전트에 재등록되어 사용할 수 있게 된다.

- 서버 어플리케이션의 활성화 (Activating server applications)

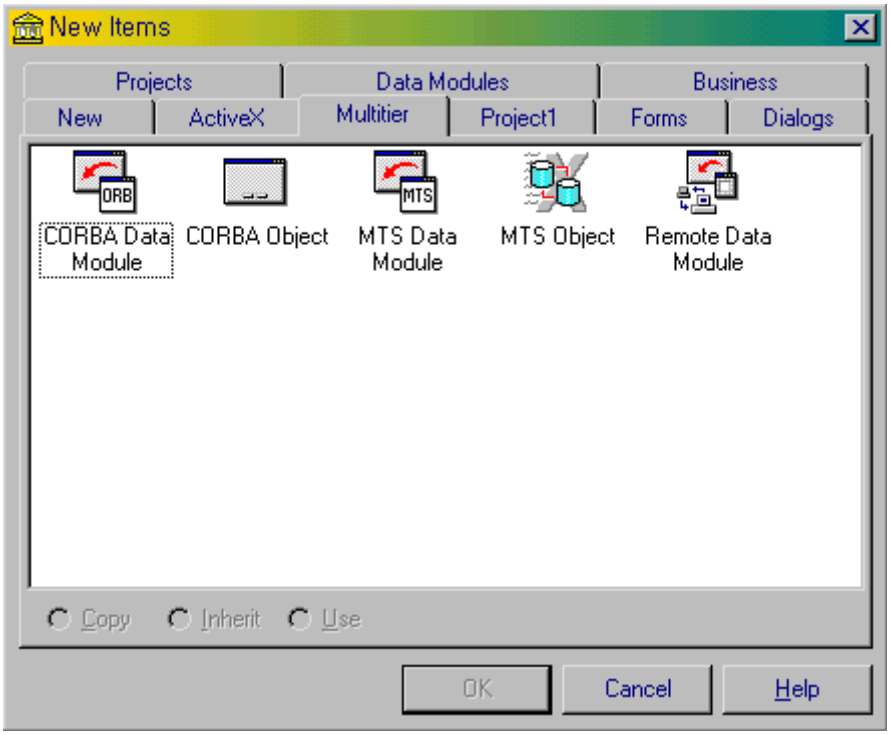
서버 어플리케이션은 자신이 시작될 때, 클라이언트의 호출을 받을 수 있는 인터페이스의 ORB 에 BOA 를 통해서 이를 알린다. 이렇게 ORB 를 초기화하고, 서버가 실행되었으며 실행이 가능하다고 알리는 코드는 CORBA 서버 어플리케이션을 시작할 때 사용하는 위저드에 의해 추가된다.
전형적으로 CORBA 서버 어플리케이션은 수동으로 시작하지만, OAD(Object Activation Daemon)를 사용하면 서버를 자동으로 시작하게 하거나 클라이언트가 필요할 때 객체를 인스턴스화 할 수 있다.
OAD 를 사용하기 위해서는 객체를 반드시 등록해야 한다. 객체를 OAD 에 등록할 때에 객체와 객체를 구현한 서버 어플리케이션 사이의 연관 관계를 구현 저장소(Implementation Repository)에 저장한다.
객체가 일단 구현 저장소에 등록되어 엔트리를 가지고 있으면, OAD 는 어플리케이션을 ORB 에 시뮬레이션하게 된다. 그러다가, 클라이언트가 객체를 요구하면 ORB 는 OAD 와 연계하여 마치 서버 어플리케이션이 실행중인 것처럼 반응하고, 동시에 OAD 는 클라이언트의 요구를 실제 서버에 전달하게 된다. 이 과정에서 필요하면 서버 어플리케이션을 시동한다.

- 인터페이스 호출의 동적 바인딩 (Binding interface calls dynamically)

전형적으로 CORBA 클라이언트는 서버 상의 객체의 인터페이스를 호출할 때 정적 바인딩(static, early binding)을 사용한다. 이 방법은 빠른 수행성능과 컴파일 시 데이터 형 검사 등의 많은 장점을 가지고 있다. 그렇지만, 런타임이 될 때까지 어떤 인터페이스를 사용하기를 원할 지 모를 수도 있는데 이럴 때에는 인터페이스를 런타임에 동적으로 바인딩할 수 있도록 허용하고 있다.
이런 동적 바이딩의 장점을 이용하고자 하면, 반드시 인터페이스를 인터페이스 저장소(Interface Repository)에 idl2ir 유틸리티를 이용해서 저장해야 한다.

CORBA 서버의 제작

New Items 대화상자의 Multitier 페이지에 있는 다음과 같은 2 개의 위저드를 사용하여 CORBA 서버를 생성할 수 있다.



- CORBA 데이터 모듈 위저드를 이용하면 멀티-tiered 데이터베이스 어플리케이션에 대한 CORBA 서버를 제작할 수 있다.
- CORBA 객체 위저드는 다양한 CORBA 서버를 생성할 때 사용한다.

또한, 기존에 작성한 OLE 자동화 서버가 있으면 오른쪽 버튼을 클릭하고 Exposing As

CORBA Object 메뉴를 선택하는 것만으로 간단하게 CORBA 서버로 전환할 수 있다. 자동화 서버를 CORBA 객체를 지원하도록 하면, 어플리케이션은 COM 클라이언트와 CORBA 클라이언트를 동시에 서비스할 수 있게 된다.

- CORBA 위저드의 활용

위저드를 시작하려면, File|New 메뉴를 선택하여 New Items 대화상자가 나오도록 하고, Multitier 페이지를 선택하고 적절한 위저드를 더블 클릭하면 된다.
이때 CORBA 객체의 클래스 이름을 적어 주면, 이 클래스가 TCorbaDataModule 이나 TCorbaImplementation 클래스를 상속받게 된다. 클래스 이름을 MyObject 라고 하면 위저드는 IMyObject 인터페이스를 구현한 TMyObject 를 선언하는 새로운 유닛을 생성한다.
또한, 위저드에서는 서버 어플리케이션의 인스턴스 모델과 쓰레드 모델을 선택해야 한다.
인스턴스 모델에는 다음의 2 가지가 있으며 적절한 것을 선택한다.

1. 인스턴스-per-클라이언트 모델

CORBA 데이터 모듈 인스턴스는 각 클라이언트 연결마다 하나씩 생성되며, 연결이 지속되는 동안에는 지속된다. 클라이언트의 연결이 종료되면 인스턴스가 메모리에서 해제된다. 이 모델은 분리된 클라이언트가 각각의 프로퍼티 설정에 서로 간섭하지 않기 때문에 지속적인 상태 정보(persistent state information)를 사용할 수 있다. 클라이언트 시스템이 객체 인스턴스를 해제하지 않고 종료되는 것을 막기 위해 어플리케이션은 주기적으로 클라이언트가 아직도 실행되고 있는지를 점검하게 되므로 공유 인스턴스 모델에 비해 이런 메시지 만큼의 네트워크 오버헤드를 가지게 된다.

2. 공유 인스턴스(shared instance) 모델

CORBA 데이터 모듈의 단일 인스턴스는 모든 클라이언트 요구를 다루게 된다. 이 모델은 전통적인 CORBA 개발 방법에 기초한 것이다. 단일 인스턴스는 모든 클라이언트를 공유해야 하므로, 이 인스턴스는 프로퍼티 설정과 같은 지속적인 상태 정보를 사용할 수 없다. 이는 클라이언트가 CORBA 데이터 모듈에서 프로바이더와 통신하게 되는 IProvider 인터페이스를 이용할 수 없다는 의미가 된다.

인스턴스 모델을 선택했으면, 이번에는 쓰레드 모델을 선택해야 한다. 쓰레드 모델에는 다음의 2 가지가 있다.

1. 싱글 쓰레드(single threaded) 모델

각각의 데이터 모듈 인스턴스는 한 번에 하나씩의 클라이언트 요구만 처리한다. 그렇기 때문에, 프로퍼티나 필드와 같은 인스턴스 데이터는 쓰레드에 안전하다. 그렇지만, 전역 메모리를 사용하는 경우에는 명확하게 이를 보호해야 한다.

2. 다중 쓰레드(multi threaded) 모델

각각의 클라이언트 연결은 자신의 쓰레드를 가지게 되지만, 데이터 모듈의 입장에서 보면 여러 클라이언트의 호출을 동시에 받게 되며 각각의 클라이언트는 자신의 쓰레드를 처리하게 된다. 그러므로 전역 메모리를 사용하는 경우 뿐만 아니라 프로퍼티나 필드 등의 인스턴스 데이터가 변화될 가능성에 대비하여 이를 보호할 책임이 있다.

- 객체 인터페이스의 정의 (Defining object interfaces)

전통적인 CORBA 도구에서는 객체의 인터페이스를 정의하기 위해서 CORBA IDL(Interface Definition Language)을 사용해야 했다. 그리고 나서, 이들을 스텁과 스켈레톤 코드로 생성해내는 유틸리티를 실행하여 사용하였다. 그렇지만, 델파이에서는 IDL 과 스텁, 스켈레톤을 모두 자동으로 생성해 준다. 개발자가 할 일은 타입 라이브러리 에디터를 이용하여 인터페이스를 편집하면 델파이가 알아서 적절한 소스 파일들을 업데이트 한다.
타입 라이브러리 에디터는 델파이 3 에서 처음 소개 되었는데, 여기서는 COM 기반의 타입 라이브러리를 제작하는 도구로 사용되었다. 그렇기 때문에, CORBA 어플리케이션과는 맞지 않는 옵션과 컨트롤 들을 많이 포함하고 있다. 개발자가 이런 옵션 들을 사용하려고 하면 이들의 설정은 CORBA 에 의해서는 무시된다. 그러므로, COM 자동화 서버를 만들면서 CORBA 서버로도 사용할 수 있도록 할 때에 이런 설정 값들이 자동화 서버를 만들 때 적용된다.
델파이 4 에서는 타입 라이브러리 에디터에서 오브젝트 파스칼 문법이나 COM 객체를 정의할 때 사용되는 마이크로소프트 IDL 문법을 모두 사용할 수 있다. 이때 어느 것을 디폴트로 할 것인지는 Environment Option 대화상자의 Type Library 페이지에서 설정할 수 있다. 주의할 것은 IDL 을 이용하겠다고 선택한 경우, Microsoft IDL 은 CORBA IDL 과 다소 차이가 있으므로, 인터페이스를 디자인할 때 다음에 소개하는 데이터 형만을 사용할 수 있다.

이 름	설 명	이 름	설 명
ShortInt	8 비트 부호 있는 정수	Byte	8 비트 부호 없는 정수
SmallInt	16 비트 부호 있는 정수	Word	16 비트 부호 없는 정수
LongInt	32 비트 부호 있는 정수	Cardinal	32 비트 부호 없는 정수
Single	4 바이트 부동 소수점 실수	Double	8 바이트 부동 소수점 실수

TDateTime	Double 값으로 넘어감	PWideChar	유니코드 문자열
PChar, String	문자열은 반드시 PChar 형태로 넘어가야 함	Variants	CORBA 에서는 Any 로 넘긴다. 배열 등을 넘길 수 있다.
Boolean	CORBA_Boolean 으로 넘어감	Enumeration	정수로 넘어간다.

이 밖에 Object Reference 나 Interface 형은 CORBA 인터페이스 형으로 넘길 수 있다.
참고로, 타입 라이브러리 에디터를 사용하지 않고 코드 에디터에서 오른쪽 버튼을 클릭하고 Add To Interface 를 선택해서 인터페이스를 추가하는 방법도 있다. 그렇지만, 이 경우에도 타입 라이브러리 에디터를 이용해서 .IDL 파일로 저장해야 사용이 가능하다.
파라미터를 사용하는 프로퍼티는 추가할 수 없으며, 이런 프로퍼티를 지원하게 하려면 get, set 메소드를 이용해서 구현해야 한다. 배열이나 Int64, Currency 등의 일부 데이터 형은 Variants 형으로 지정해야 한다. 레코드는 C/S 버전에서는 지원되지 않으며 Enterprise 버전에서 지원된다.
인터페이스에 대한 변경 사항은 자동으로 생성되는 스텁-스켈레톤 유닛에 반영되며, 이 유닛은 타입 라이브러리 에디터에서 Refresh 명령을 선택하거나 Add To Interface 명령을 사용하면 업데이트 된다. 이렇게 자동으로 생성된 유닛은 implementation 유닛의 uses 절에 추가되므로 스텁-스킬레톤 유닛을 직접 편집하는 것은 피하는 것이 좋다.
인터페이스를 편집하면 스텁-스켈레톤 유닛 이외에 서버 구현 유닛에 인터페이스 멤버에 대한 선언과 메소드 구현부에 대한 껍데기 코드가 자동으로 생성된다. 개발자는 구현부에 생성된 begin ~ end 사이에 적절한 인터페이스를 구현하면 된다.

참고:

CORBA 의 IDL 파일은 타입 라이브러리 에디터의 Export 버튼을 클릭하고 CORBA IDL 을 선택하면 따로 저장할 수 있다. 이렇게 저장한 .IDL 파일을 이용해서 인터페이스를 등록하거나 C++ 등의 다른 언어를 이용하여 스텁과 스켈레톤을 생성하도록 할 수 있다.

- 자동으로 생성된 코드

CORBA 객체 인터페이스를 정의하고 나면, 인터페이스 정의를 반영하기 위해 2 개의 유닛 파일이 자동으로 업데이트 된다.
첫번째 유닛은 스텁-스켈레톤 유닛으로 MyInterface_TLB.pas 라는 이름을 가지는 파일이다. 이 유닛은 클라이언트 어플리케이션에 의해 사용되는 스텁 클래스를 정의하고, 인터페이스 형과 스켈레톤 클래스에 대한 선언부를 포함한다. 이 파일은 직접 편집하면 안되며, 구현 유닛의 uses 절에 자동으로 추가된다.
스텁-스켈레톤 유닛은 CORBA 서버에 의해 지원되는 인터페이스에 대한 스켈레톤 객체를

정의한다. 스켈레톤 객체는 TCorbaSkeleton 클래스의 자손으로 인터페이스 호출을 마샬링하는 세부적인 동작을 처리하게 된다. 이 객체는 정의한 인터페이스를 직접 구현하는 것은 아니지만, constructor 가 인터페이스 인스턴스를 이용하여 모든 인터페이스 호출을 처리하게 된다.
두번째로 업데이트 되는 유닛은 실제 구현을 담당하는 implementation 유닛이다. 이 유닛의 디폴트 이름은 Unit1.pas 이다. 이 이름은 물론 다른 이름으로 바꿀 수 있다.
정의한 각각의 CORBA 인터페이스에 대해 구현 클래스는 자동으로 정의되어 implementation 유닛에 추가된다. 구현 클래스의 이름은 인터페이스 이름에 기초하는데, 예를 들어, 인터페이스 이름이 IMyInterface 라면 구현 클래스의 이름은 TMyInterface 가 된다. 기본적으로 인터페이스에 선언된 메소드에 대한 구현 부분의 껍데기는 자동으로 생성되므로 이들에 대한 몸체만 코딩하면 된다.
또한, implementation 유닛의 initialization 섹션에는 CORBA 클라이언트 들에 노출된 각각의 객체 인터페이스에 대한 TCorbaFactory 객체를 생성하는 코드가 추가된다. 클라이언트가 CORBA 서버를 호출하면 CORBA 팩토리 객체가 생성되거나 구현 클래스의 인스턴스를 위치시키고 이를 인터페이스로 하여 해당되는 스켈레톤 클래스에 대한 constructor 에 넘겨진다.

- 서버 인터페이스의 등록 (Registering server interfaces)

클라이언트 호출에 대해서 서버 객체를 정적 바인딩만 사용하려 한다면 서버 인터페이스를 반드시 등록할 필요는 없지만, 등록시키는 것이 좋다. 인터페이스를 등록하는데 사용할 수 있는 유틸리티는 다음의 2 가지가 있다.

1. 인터페이스 저장소 (Interface Repository)

인터페이스 저장소에 인터페이스를 등록하면 클라이언트는 동적 바인딩을 사용할 수 있다. 이를 통해 클라이언트가 DII(dynamic invocation interface)를 사용했다면 델파이가 아닌 다른 언어로 작성된 클라이언트에도 서버가 반응할 수 있다. 인터페이스 저장소를 이용하면 다른 개발자가 인터페이스를 직접 보고, 이를 이용해서 클라이언트 어플리케이션을 작성하도록 할 수 있는 장점이 있다.

2. 객체 활성화 데몬 (Object Activation Daemon, OAD)

객체 활성화 데몬에 인터페이스를 등록하면 객체가 클라이언트에 의해 요구를 받을 때까지 인스턴스화 되지 않게 할 수 있다. 이를 통해 서버 시스템의 리소스를 많이 절약할 수 있다.

CORBA 클라이언트의 제작

CORBA 클라이언트를 작성할 때의 첫번째 단계는 클라이언트 어플리케이션이 클라이언트의 ORB 소프트웨어와 통신할 수 있도록 하는 것이다. 이를 위해서는 유닛 파일의 uses 절에 CorbaInit.pas 유닛을 추가하면 된다.
그리고 나서는 일반적인 다른 델파이 어플리케이션을 개발하는 방법과 똑같이 개발하면 된다. 서버 어플리케이션에 정의된 객체를 사용하고자 할 때에는 객체 인스턴스를 직접 호출하지 않고, 객체에 대한 인터페이스를 얻어서 이를 이용하여 작업을 하면 된다. 인터페이스를 얻는 방법에는 크게 정적 바인딩(static, early binding)과 동적 바인딩(dynamic, late binding)이 있다.
정적 바인딩을 이용하려면 스텁-스켈레톤 유닛을 클라이언트 어플리케이션에 추가해야 하는데, 이 유닛은 서버 인터페이스를 저장할 때 자동으로 생성된다. 정적 바인딩이 동적 바인딩에 비해 수행속도가 빠르며, 컴파일 시에 데이터 형을 검사하는 장점이 있고 동시에 델파이 3 에서부터 지원하는 코드 완료(code completion)를 사용할 수 있다.
그렇지만, 어떤 객체나 인터페이스를 사용해야 하는지를 런타임이 될 때까지 모를 때에는 동적 바인딩을 사용해야 한다. 동적 바인딩은 스텁 유닛을 요구하지 않지만, 사용하는 모든 원격 객체 인터페이스(remote object interface)가 LAN 상에서 동작하는 인터페이스 저장소에 등록되어 있어야 한다.

- 스텁의 활용 (Using stubs)

스텁 클래스는 CORBA 인터페이스를 정의할 때 자동으로 생성된다. 이 클래스는 스텁-스켈레톤 유닛에 정의되어 있다. 스텁-스켈레톤 유닛의 이름은 BaseName_TLB.pas 이다.
CORBA 클라이언트를 작성할 때, 스텁-스켈레톤 유닛의 코드는 직접 수정하지 않는다. 단지 이 유닛을 uses 절에 추가하고, 인터페이스를 호출하면 된다.
각각의 서버 객체에 대해 스텁-스켈레톤 유닛은 해당 스텁 클래스에 대한 인터페이스 정의와 클래스 정의를 포함하고 있다. 예를 들어, 서버가 TServerObj 객체 클래스를 정의한다면 스텁-스켈레톤 유닛은 IServerObj 인터페이스에 대한 정의와 TServerObjStub 클래스에 대한 정의를 포함한다. 스텁 클래스는 TCorbaDispatchStub 클래스의 자손으로 해당되는 인터페이스의 마샬링하여 CORBA 서버를 호출하는 역할을 한다. 스텁 클래스 외에도 스텁-스켈레톤 유닛에는 각각의 인터페이스에 대한 스텁 클래스 팩토리를 정의하고 있다. 스텁 클래스 팩토리는 인스턴스화 되지 않으며, 하나의 클래스 메소드만 정의한다.
클라이언트 어플리케이션에서는 CORBA 서버의 객체에 대한 인터페이스를 필요로할 때 스텁 클래스의 인스턴스를 직접 생성하지 않는다. 그 대신에 스텁 팩토리 클래스의 CreateInstance 메소드를 호출한다. 이 메소드는 옵션 인스턴스 이름을 하나의 argument

를 가지며, 서버 상의 객체 인스턴스에 대한 인터페이스를 반환한다.
사용법은 다음과 같다.

```
var
  ObjInterface : IServerObj;
begin
  ObjInterface := TServerObjFactory.CreateInstance('');
  ...
end;
```

CreateInstance 를 호출할 때 이 메소드는 ORB 에서 인터페이스 인스턴스를 얻어오며, 인터페이스를 이용하여 스텁 클래스의 인스턴스를 생성한다. 마지막으로 결과 인터페이스를 반환한다.

- DII(dynamic invocation interface)의 활용

DII 는 클라이언트 어플리케이션이 인터페이스 호출을 마샬링하는 스텁 클래스를 사용하지 않고도 서버 객체를 호출할 수 있도록 해준다. DII 는 컴파일 시에 인터페이스 호출을 바인딩하지 않기 때문에 스텁 클래스를 사용하는 것보다 다소 느리게 동작한다.
DII 를 사용하기 전에 서버 인터페이스는 LAN 상에서 동작하는 인터페이스 저장소에 등록되어 있어야 한다. 클라이언트 어플리케이션에서 DII 를 사용하려면 서버 인터페이스를 얻어서 이를 TAny 데이터 형 변수에 대입한다. TAny 는 CORBA 에서 사용하는 특수한 Variant 데이터 형이다. 그리고 나서 TAny 변수를 인터페이스 인스턴스인 것처럼 사용하여 인터페이스 메소드를 호출하면 된다.

- 인터페이스의 획득 (Obtaining the interface)

DII 를 이용한 동적 바인딩을 사용하기 위해서는 전역 CorbaBind 함수를 호출하면 된다. CorbaBind 함수에는 서버 객체의 Repository ID 나 인터페이스 형을 정해주어야 한다. 이 정보를 이용해서 ORB 에게 인터페이스를 요구하고, 스텁 객체를 생성하게 된다.
CorbaBind 함수를 호출하기 전에 인터페이스 형과 인터페이스의 Repository ID 는 반드시 CorbaInterfaceIDManager 함수를 이용해서 등록해야 한다.
클라이언트 어플리케이션이 인터페이스 형에 대한 스텁 클래스를 등록하면, CorbaBind 함수는 그 클래스의 스텁을 생성하고, 이 함수에 의해 반환된 인터페이스는 'as' 형변환을 통한 정적 바인딩과 동적 바인딩에 모두 사용될 수 있다. 이때 인터페이스 형에 대해 등록된 스텁 클래스가 없다면, CorbaBind 는 기본 스텁 객체(generic stub object)에 대한 인터

페이스를 반환한다. 기본 스텁 객체는 DII 를 이용한 동적 호출을 이용해야만 사용할 수 있다. 실제 사용하는 코드를 살펴 보자.

```
var
  IntToCall: TAny;
begin
  IntToCall := CorbaBind('IDL:MyServer/MyServerObject:1.0');
  ...
```

- DII 를 이용한 인터페이스 호출

인터페이스가 TAny 형 변수에 대입되면, DII 를 이용하여 다음과 같이 간단하게 호출하여 사용할 수 있다.

```
var
  HR, Emp, Payroll, Salary: TAny;
begin
  HR := CorbaBind('IDL:CompanyInfo/HR:1.0');
  Emp := HR.LookupEmployee(Edit1.Text);
  Payroll := CorbaBind('IDL:CompanyInfo/Payroll:1.0');
  Salary := Payroll.GetEmployeeSalary(Emp);
  Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit2.Text) / 100));
end;
```

DII 를 사용할 때에는 모든 인터페이스 메소드가 대소문자를 가리므로 주의하기 바란다. DII 를 이용해서 인터페이스를 호출하면 모든 파라미터가 TAny 데이터 형으로 취급된다. 이 데이터 형을 사용하면 서버가 호출을 받은 후에 데이터 형을 해석하는 것을 허용한다. 파라미터가 항상 TAny 값으로 취급되기 때문에 파라미터 데이터 형을 적절하게 맞추기 위해 먼저 형변환을 할 필요가 없다. 예를 들어, 앞에서의 SetEmployeeSalary 메소드의 경우 문자열 대신 실수를 파라미터로 사용해도 된다.
그렇지만 구조체를 파라미터로 사용하려면 전역 ORB 변수의 변환 메소드를 이용하여 적절한 TAny 데이터 형 값을 만들어 사용해야 한다. 이때 레코드는 MakeStructure, 정적 배열은 MakeArray, 동적 배열은 MakeSequence 라는 메소드를 이용하여 변환을 한다.
이런 함수를 이용할 때에는 생성하고자 하는 레코드, 배열의 데이터 형을 반드시 기술해야 하는데, 이런 데이터 형은 다음과 같이 ORB 의 FindTypeCode 메소드에서 Repository ID 를 이용하면 동적으로 얻을 수 있다.

```
var
  HR, Name, Emp, Payroll, Salary: TAny;
begin
  with ORB do
  begin
    HR := Bind('IDL:CompanyInfo/HR:1.0');
    Name := MakeStructure(FindTypeCode('IDL:CompanyInfo/EmployeeName:1.0',
                                       [Edit1.Text, Edit2.Text]);
    Emp := HR.LookupEmployee(Name);
    Payroll := Bind('IDL:CompanyInfo/Payroll:1.0');
  end;
  Salary := Payroll.GetEmployeeSalary(Emp);
  Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit3.Text) / 100));
end;
```

CORBA 어플리케이션 구워 삶기

ORB 와 BOA 라는 2 개의 전역 변수를 이용하면 어플리케이션이 CORBA 소프트웨어와 상호작용하는 방법을 여러 가지로 적용해볼 수 있다.
클라이언트 어플리케이션은 ORB 변수를 사용해서 ORB 소프트웨어의 환경을 설정하고, 서버와의 연결을 해제하며, 인터페이스에 바인드하여 객체에 대한 문자열을 얻는 등의 작업을 하게 된다.
서버 어플리케이션은 BOA 변수를 이용해서 BOA 소프트웨어의 환경을 설정하고, 숨겨진 객체를 노출시키며, 클라이언트 어플리케이션에 의해 주어진 사용자 정의 정보를 가져올 수 있다.

- 사용자 인터페이스에 객체 보여주기

CORBA 클라이언트 어플리케이션을 제작할 때, 사용가능한 CORBA 서버 객체의 이름을 사용자에게 보여주고 싶을 때가 있다. 이럴 때에는 객체의 인터페이스를 문자열로 변환해서 보여주어야 하는데, ORB 변수의 ObjectToString 메소드가 이런 역할을 한다. 예를 들어, 다음의 코드는 3 개의 객체를 리스트 박스에 보여준다.

```
var
  Dept1, Dept2, Dept3: IDepartment;
```

```
begin
  Dept1 := TDepartmentFactory.CreateInstance('Sales');
  Dept1.SetDepartmentCode(120);
  Dept2 := TDepartmentFactory.CreateInstance('Marketing');
  Dept2.SetDepartmentCode(98);
  Dept3 := TSecondFactory.CreateInstance('Payroll');
  Dept3.SetDepartmentCode(49);
  ListBox1.Items.Add(ORB.ObjectToString(Dept1));
  ListBox1.Items.Add(ORB.ObjectToString(Dept2));
  ListBox1.Items.Add(ORB.ObjectToString(Dept3));
end;
```

ORB 변수는 이 뿐만 아니라 문자열을 이용해서 객체를 생성할 수 있도록 StringToObject 메소드 역시 제공하고 있다. 다음 코드를 살펴보자.

```
var
  Dept: IDepartment;
begin
  Dept := ORB.StringToObject(ListBox1.Items[ListBox1.ItemIndex]);
  ...      {이 객체를 사용하는 코드}
```

- CORBA 객체의 노출과 숨김

CORBA 서버 어플리케이션이 객체 인스턴스를 생성할 때, BOA 변수의 ObjIsReady 메소드를 호출하면 클라이언트에서 이 객체를 사용할 수 있게 된다. ObjIsReady 메소드를 통해 사용가능해진 객체는 서버 어플리케이션이 다시 숨길 수 있다. 이를 위해서는 BOA 의 Deactivate 메소드를 사용한다.
서버의 객체를 사용할 수 있는지 여부를 클라이언트 어플리케이션이 알기 위해서는 스텁 객체의 NonExistent 메소드를 사용해서 알아낼 수 있다. 서버 객체가 비활성화된 경우에 이 메소드는 True 를 반환하며, 서버 객체를 사용할 수 있으면 False 를 반환한다.

- 클라이언트 정보를 서버 객체로 넘겨주기

CORBA 클라이언트의 스텁 객체는 연관된 서버 객체에 TCorbaPrincipal 을 이용해 정보를 전송할 수 있다. TCorbaPrincipal 은 클라이언트 어플리케이션에 대한 정보를 가지고 있는 일종의 배열이다. 스텁 객체는 이 배열의 값을 SetPrincipal 메소드를 이용해서 설정할 수

있다.
일단 CORBA 클라이언트가 데이터를 서버 객체 인스턴스에 기록하면, 서버 객체는 이 정보에 접근할 때 BOA 의 GetPrincipal 메소드를 사용하면 된다.
TCorbaPrincipal 이 바이트의 배열이기 때문에, 개발자가 보낼 수 있는 어떤 정보도 전송할 수 있다. 예를 들어, 특정 메소드를 실행가능하게 만들기 전에 서버가 검사할 수 있는 키 값을 클라이언트가 전송하도록 만들 수도 있다,

CORBA 어플리케이션의 배포

일단 클라이언트나 서버 어플리케이션을 제작하고 이들을 테스트하고 나면 클라이언트 어플리케이션은 사용자의 데스크 탑 컴퓨터에게, 서버 어플리케이션은 서버 기계에 배포해야 한다. 이때 클라이언트와 서버 어플리케이션에는 다음과 같은 파일이 반드시 설치되어 있어야 한다.

1. ORB 라이브러리는 모든 클라이언트와 서버 기계에 설치되어 있어야 한다. 이들 라이브러리는 VisiBroker 를 설치할 때 Bin 서브 디렉토리에 설치되므로 여기에서 찾을 수 있다.
2. 클라이언트가 DII(dynamic invocation interface)를 사용하도록 하였다면, 네트워크 상에 최소한 하나의 호스트에서 인터페이스 저장소(Interface Repository) 서버를 실행해야 한다.
3. 서버가 수동으로 시작되는 것이 아니라 클라이언트의 요구가 있을 때에만 실행하도록 하고 싶다면, OAD(Object Activation Daemon)가 네트워크 상에 최소한 하나의 호스트에서 실행되어야 한다.
4. 네트워크 상에 최소한 하나의 호스트에는 스마트 에이전트(osagent)가 반드시 설치되어야 한다.

그리고, CORBA 어플리케이션을 배포할 때 다음과 같은 환경 변수를 설정할 필요가 있을 수 있다.

변 수	의 미
PATH	ORB 라이브러리가 포함된 디렉토리
VBROKER_ADM	인터페이스 저장소, OAD, 스마트 에이전트에 대한 환경설정 파일을 포함하는 디렉토리
OSAGENT_ADDR	스마트 에이전트가 사용될 호스트 컴퓨터의 IP 주소. 이 변수가 설정되지 않으면 CORBA 어플리케이션은 브로드캐스트 메시지에 응답하는 첫번째 스마트 에이전트를 사용한다.

OSAGENT_PORT	스마트 에이전트가 클라이언트의 요구를 기다릴 포트를 지정한다.
OSAGENT_ADDR_FILE	다른 네트워크 상의 스마트 에이전트 주소를 담고 있는 파일의 패스
OSAGENT_LOCAL_FILE	멀티 호스트에서 동작하는 스마트 에이전트에 대한 네트워크 정보를 담은 파일의 패스
VBROKER_IMPL_PATH	구현 저장소(Implementation Repository)에 대한 디렉토리 지정 (OAD 에 여기에 대한 정보를 담고 있다).
VBROKER_IMPL_NAME	구현 저장소의 디폴트 파일 이름을 지정한다.

- 스마트 에이전트의 환경 설정

CORBA 어플리케이션을 배포할 때, 최소한 하나의 스마트 에이전트가 실행되고 있어야 한다. 하나 이상의 스마트 에이전트를 배포할 때에는 현재 실행되는 스마트 에이전트의 실행이 비정상 종료되었을 때 이를 보호하기 위한 방법을 제공해야 한다.
개발자는 네트워크를 ORB 도메인에 따라 스마트 에이전트를 배포한다. 이때 ORB 도메인을 넓히기 위해서 다른 네트워크에 있는 스마트 에이전트를 연결할 수도 있다.

1. 스마트 에이전트의 시작

스마트 에이전트를 시작하기 위해서는 osagent 유틸리티를 실행해야 한다. osagent 유틸리티에는 다음과 같은 커맨드 라인 arguments 를 사용할 수 있다.

Argument	설 명
-v	verbose 모드를 활성화한다. 정보와 시스템 진단 메시지가 osagent.log 파일에 기록된다. 이 파일은 VBROKER_ADM 환경 변수에 지정된 디렉토리에서 찾을 수 있다.
-p<n>	스마트 에이전트가 브로드캐스트 메시지를 기다릴 UDP 포트를 지정한다.
-C	NT 서비스로 설치된 경우 스마트 에이전트를 콘솔 모드로 동작하게 한다.

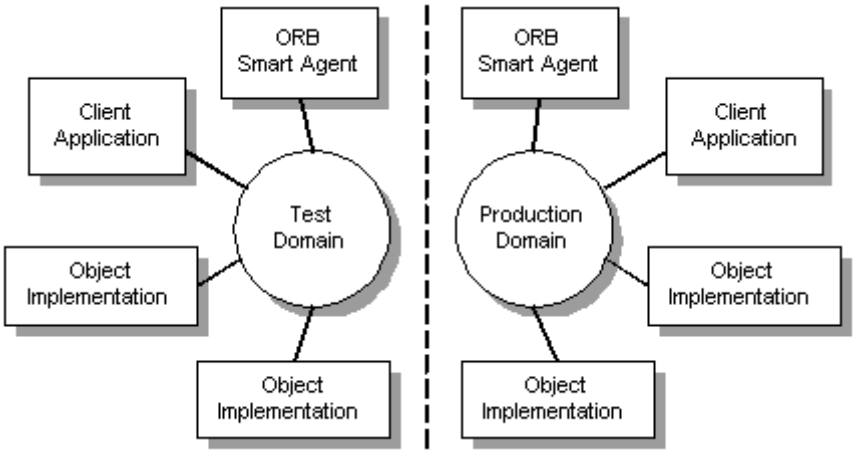
예를 들어, 도스창이나 실행 명령에서 다음과 같이 지정할 경우 UDP 포트는 디폴트로 지정된 14000 을 사용하지 않고, 11000 을 사용하게 된다.

```
osagent -p 11000
```

스마트 에이전트가 브로트캐스트를 기다리는 포트를 바꿀 경우 여러 개의 ORBB 도메인이 생성되는 것을 허용할 수 있다.

2. ORB 도메인 환경 설정

2 개 이상의 분리된 ORB 도메인을 실행하는 경우 상당한 장점이 있다. 하나의 도메인에는 클라이언트 어플리케이션의 정식 버전(production version)과 객체를 구현하고, 다른 도메인에는 같은 클라이언트와 객체의 테스트 버전을 포함한다. 이 경우 여러 명의 개발자가 같은 네트워크에서 작업을 할 경우 각각의 개발자는 테스트를 다른 개발자 들을 간섭하지 않고 할 수 있게 된다.



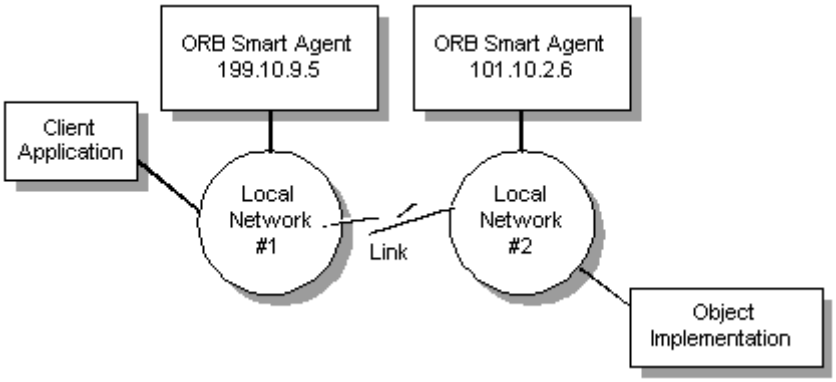
같은 네트워크 상의 2 개 이상의 ORB 도메인은 각 도메인의 osagents 의 유일한 UDP 포트 이름을 이용해서 구별한다. 디폴트 포트 번호는 14000 으로 ORB 가 설치될 때 윈도우 레지스트리에 기록된다. 이 값을 재정의하려면 OSAGENT_PORT 환경 변수를 다르게 설정하면 된다. 또한, 환경 변수의 값을 또 다시 변경하려면 스마트 에이전트를 시작할 때 –p 옵션을 주면 된다.

3. 다른 네트워크 상의 스마트 에이전트와의 접속

네트워크 상에 여러 개의 스마트 에이전트를 시작할 경우, 이들은 UDP 브로드캐스트 메시지를 이용해서 서로를 발견하게 된다. 그러므로, 브로드캐스트 메시지가 전달되는 범위가 결정되는 IP 서브넷 마스크 범위 내에 있어야 한다. 그렇지만, 서로 다른 네트워크 상에 있어도 서로 통신이 가능한데 다음과 같은 방법을 사용할 수 있다.

- agentaddr 파일을 이용하는 방법

다음 그림과 같은 2 개의 스마트 에이전트가 있다고 하자. #1 네트워크 상의 스마트 에이전트는 IP 주소 199.10.9.5 를 사용하고, #2 네트워크 상의 스마트 에이전트는 101.10.2.6 을 IP 주소로 사용한다.



여기서 #1 의 스마트 에이전트는 #2 의 스마트 에이전트와 접속하기 위해서 agentaddr 이라는 파일을 이용하는데, 다음과 같은 줄을 포함하고 있으면 된다.

101.10.2.6

스마트 에이전트는 VBROKER_ADM 환경 변수에 지정된 디렉토리에서 이 파일을 찾아서 연결하게 된다.

- 멀티-홈 호스트(multi-homed host)의 이용

스마트 에이전트를 하나 이상의 IP 주소를 가진 멀티-홈 호스트에서 실행하는 경우, 다른 네트워크 상에 있는 객체와의 브리징을 할 수 있는 강력한 방법이 있다. 이 경우에는 호스트와 연결된 모든 네트워크는 하나의 스마트 에이전트와 통신할 수 있게 되므로, agentaddr 파일은 필요하지 않다. 어쨌든 멀티 홈 호스트에서는 스마트 에이전트가 적절한 서브넷 마스크와 브로드캐스트 주소 값을 결정할 수 있다. 개발자는 이 값들을 localaddr 파일에 설정해 주어야 한다. 이 값들은 네트워크 환경 설정에서 TCP/IP 프로토콜의 설정 값을 이용하거나, 윈도우 NT 의 경우 ipconfig 명령을 이용하여 얻을 수 있다.
localaddr 파일에는 IP 주소, 서브넷 마스크, 브로드캐스트 주소의 조합을 포함하게 된다. 예를 들어, 다음의 리스트는 2 개의 IP 주소에 대한 스마트 에이전트의 localaddr 파일의 내용이다.

216.64.15.10 255.255.255.0 216.64.15.255
214.79.98.88 255.255.255.0 214.79.98.255

개발자는 OSAGENT_LOCAL_FILE 환경 변수에 localaddr 파일의 경로를 설정해 주어야 한다.

정 리 (Summary)

이번 장에서는 CORBA 에 대한 전체적인 개념과 델파이 4 에서 지원하는 여러가지 클래스와 도구 들의 사용방법에 대해서 알아보았다. 새로운 기능이기 때문에 다소 어렵게 느껴질 수도 있지만, 분산 환경을 구현하는데 매우 강력하게 동작하므로 그 활용도는 높다고 할 수 있겠다.