효율적인 분산 객체 관리를 위한 구현저장소 인터페이스 설계

o
구현주*, 이 윤*, 박세명*
*인제대학교 전산학과

Design of Implementation Repository Interface for Effective Management of Distributed Object

Koo-Hyun Ju*, Lee-Yoon*, Park-Se Myung*,
*Department of Computer Science, Inje University

요 약

분산 객체 컴퓨팅 환경이 대두되고 있는 오늘날, 네트워크 상에 분산되어 있는 객체들을 효율적으로 관리할 수 있는 기능이 필요하게 되었다. OMG(Object Management Group)의 CORBA(Common Object Request Broker Architecture)에서는 객체를 바인딩하기 위한 두 가지 형태의 객체참조, 즉 일시적 객체참조(Transient Object Reference)와 영구적 객체참조(Persistent Object Reference)가 있으며 영구적 객체참조를 바인딩하기 위해서 ORB는 구현저장소를 제공해야 한다. 이를 위해 CORBA 스펙에서는 구현저장소의 개념만을 정의되어 있을 뿐 인터페이스는 아직 정의 되어 있지 않은 상태이다. 본 논문에는 구현저장소의 도움을 받아 구현객체를 바인딩할 수 있도록 하기 위한 구현저장소의 인터페이스를 설계하였다.

1. 서론

네트워크는 인터넷 및 인트라넷의 보급이 확산되면서 컴퓨터 환경에 기반을 이루게 되었으며 이로 인해 이기종간의 분산 객체 컴퓨팅의 중요성이 부각되고 있다. 그러나 분산 객체 컴퓨팅은 상이한 운영체제, 네트워크, 언어, 하드웨어 등으로 매우 복잡하고 어려운 작업으로 여겨지고 있다. 이의 대안으로 OMG의 CORBA, Microsoft사의 DCOM(Distributed Component Object Model), Sun사의 Java RMI (Remote Method Invocation) 등의 ORBs(Object Request Brokers)를 들 수 있다. 객체의 위치, 객체의 활성화, 파라미터 마샬링, 보안등의 네트웍 프로그래밍의 지식 없이도 클라이언트와 구현객체간의 IDL(Interface Definition Language)을 이용하여 ORB로 통신할 수 있다[1].

여기서 클라이언트는 객체참조(Object Reference)를 얻어야만 구현객체의 오퍼레이션을 실행시킬 수 있다. 또한 객체참조에는 두 가지 타입, 즉 일시적 객체참조(Transient Object Reference)와 영구적 객체참조(Persistent Object Reference)가 있으며 영구적 객체참조를 바인딩하기 위해서 ORB는 구현저장소를 제공해야 한다[2].

본 논문에서는 클라이언트가 구현저장소의 도움을 받아 구현객체를 바인딩할 수 있도록 효율적인 구현저장소를 CORBA의 표준 스펙을 따르는 Free Java ORB인 JacORB[3]의 구현저장소 IDL를 설계하였다.

본 논문의 2장에서는 IOR(Interoperable Object Reference), 바인딩, 구현저장소의 개념을 살펴보고, 3장에서는 구현저장소의 인터페이스 설계와 이를 적용한 JacORB의 구현저장소 IDL를 대해 설명하고, 마지막 4장에서는 결론과 향후 연구에 대해 논한다.

2. 관련연구

2.1 상호운용의 객체참조(IOR)

객체참조는 ORBs 시스템 내에서 객체를 유일하게 식별하여 기술할 수 있는 정보를 제공한다. CORBA 스펙으로 정의만 되어져 있고 객체참조에 대한 구현은 벤더에게 맡겨져 있다.

CORBA는 객체참조를 일시적 객체참조와 영구적 객체참조, 두 가지의 형태로 정의한다. 그 둘의 차이점은 참조를 생성한 서버 프로세스의 수명에 관련되어 있다. 우선, 일시적 객체참조의 수명은 이것의 서버 프로세스의 수명에 제한되어 있다. 한번 서버 프로세스가 종료하고 나면 더 이상 존재하지 않게 된다. 즉, 이것은 객체에 대한 모든 참조들이 무효화되어버린다는 것이다. 비록 서버가 다시 재시작 되더라도 일시적 객체참조는 결코 다시 작동하지 않는다. 반면에, 영구적 객체참조는 그들의 서버 프로세스에 영향을 받지 않는다. 그러므로, 비록 서버가 종료되고 다시 재시작 하더라도 똑같은 객체를 가리키고 있다.

일반적으로, 상호운용의 객체참조(IOR)는 그림 1에서와 같은 구조를 가진다[2].

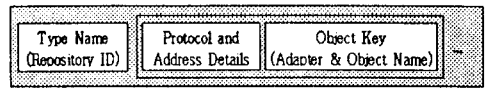

그림 1. 상호운용의 객체참조(IOR)의 구조

- Type Name은 참조가 생성될 때 알려지는 가장 최근의 유도타입을 나타내는 식별자로서 저장소 ID라고도 한다. 저장소ID는 인터페이스 저장소(Interface Repository)에서 색인의 역할을 한다. 이것은 클라이언트가 객체에서 지원할 수 있는 인터페이스를 알고 싶을 때 사용된다. 또한 실행시에 타입을 확인하기 위해 ORB에 의해 사용될 수도 있다.
- Protocol and Address Details 필드는 프로토콜과 그 프로토콜에 적당한 주소정보를 기술한다. 인터넷에서 가장 많이 사용되는 TCP/IP위에 구현된 IIOP(Interoperable Inter-ORB Protocol)의 경우는 호스트이름과 TCP포트번호로 구성된 주소정보를 나타낸다.
- Object Key는 IOR에 의해 실제적인 객체를 가리켜지는 값을 나타낸다. 이는 두 가지의 구성요소 즉, 객체어댑터(Object Adapter)와 객체이름으로 구성되어있다.

2.2 바인딩(Binding)

클라이언트가 IOR를 통해 요청을 하는 경우 ORBs는 실행 시에 정확한 구현객체에게 요청을 보내는 역할을 하고 있다. 구현객체와 함께 이러한 요청에 관련된 프로세스를 바인딩이라고 한다. 바인딩은 프로토콜에 제한적인데 본 논문에서는 IIOP에 적용하고 있다.

IOR은 일시적이나 영구적이든지 간에 서버 애플리케이션 코드안에서에 생성되어진다. 그러나 ORBs는 일시적 참조와 영구적 참조를 서로 다르게 바인딩하고 있다.

2.2.1 일시적 참조(Transient Reference)의 바인딩

그림 2는 일시적 참조의 바인딩의 예를 보여주고

있다[2].

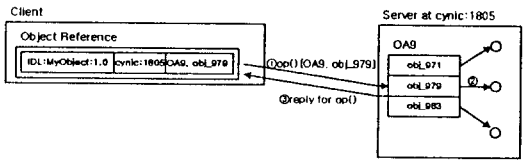

그림 2. 일시적 객체참조의 바인딩

일시적 참조의 바인딩은 다음의 네 가지 시나리오 중 하나의 경우에 해당한다.

첫째, 참조할 서버가 있는 호스트에서 실행중인 경우 바인딩은 성공적이다. 따라서 서버는 Object Key를 사용해서 구현객체를 결정하여 클라이언트에게 응답을 돌려준다.

둘째, 명시된 주소에서 실행하고 있는 프로세스가 없는 경우 클라이언트 측의 응용프로그램 코드에 의해서 TRANSIENT 예외처리가 발생한다.

셋째, 참조를 생성한 최초의 서버가 종료되었고 다른 서버가 똑같은 호스트와 포트에서 실행하고 있는 경우 클라이언트는 잘못된 서버에게 요청을 했기 때문에 그 요청은 OBJECT_NOT_EXIST 예외처리발생과 동시에 실패할 것이다.

마지막으로 참조를 생성한 최초의 서버가 종료되었지만 나중에 다시 재시작 되었고 똑같은 포트 번호를 가지게 된 경우, 클라이언트는 올바른 서버에게 요청을 보내지만 참조는 일시적이므로 그 요청은 OBJECT_NOT_EXIST라는 예외처리를 발생시키고 실패하게 될 것이다.

2.2.2 구현저장소(Implementation Repository)

CORBA 스펙에 따라 구현저장소는 ORB가 구현된 객체를 위치시키고 활성화시킬 수 있는 정보를 포함한다. 구현 저장소의 대표적인 기능, 세 가지는 다음과 같다.

첫째, 알려진 서버의 레지스터리를 관리한다.

둘째, 현재 실행중인 서버는 어떤 것인지, 그리고 사용하고 있는 호스트와 포트는 무엇인지를 기록한다.

셋째, 요구된 서버가 만약 구현저장소에 등록되어 있다면 그 서버를 활성화시킨다.

구현저장소는 전형적으로 고정된 주소, 즉 호스트와 포트 번호를 가진 주소에서 실행하고 있는 프로세스로 구현되어진다. 만약에 서버가 영구적 IOR을 생성한다면 서버의 호스트 이름은 구현저장소의 주소로 설정되어야만 한다. 구현저장소는 다음의 표1과 같은 구조 형태로 정보를 유지한다[2].

Adapter Name	Start-up Command	Address
Yoon	java example1.Server	cynic:1799
Ninec	/usr/local/bin/bank	
Ican1		deepbox:3333

표 1 구현저장소의 구성의 예

- Adapter Name은 요청을 진행하는 서버 측의 객체어댑터를 나타낸다.
- Start-Up Command는 클라이언트가 객체에 연결을 시도할 때 서버가 어떻게 시작될 수 있는지를 기록한다. 이 칼럼이 비어 있다는 것은 서버가 수동으로 시작될 것이다라는 것을 의미한다.
- Address 필드는 서버가 현재 실행하고 있는 호스트와 포트 번호를 기록한다. 이 필드가 비어있는 것은 서버가 실행되고 있지 않다는 것을 나타낸다.

영구적 참조를 생성하기를 원하는 서버는 구현저장소에 등록되어져야만 한다. 서버는 로컬 시스템의 환경설정으로부터 구현 저장소를 찾을 수 있으므로 이를 바탕으로 시작하는 동안 구현 저장소에 어댑터 이름과 호스트 그리고 포트 번호를 보낸다. 물론 서버의 객체어댑터가 등록되어있지 않다면 구현 저장소는 클라이언트의 응용프로그램 코드에 의해 BAD_PARAM이나 혹은 OBJ_ADAPTER와 같은 예외처리로 돌려보낸다.

이러한 방식으로 구현저장소는 각각의 실행중인 서버에 대한 가장 최근의 주소를 항상 알고 있고 단지 등록되어진 서버만이 영구적 참조를 생성할 수 있다. 반면에 일시적 참조를 생성하는 서버는 등록될 필요가 없다.

2.2.3 영구적 참조(Persistent reference)의 바인딩

영구적 참조가 2.2.1의 일시적 참조와 다른점은 구현저장소의 호스트 이름과 포트 번호를 가진다는 것이다. 그러나 클라이언트가 영구적 IOR을 바인딩하는 것은 일시적 IOR을 바인딩할때와 마찬가지의 방법을 사용한다. 물론 영구적 참조에서는 먼저 구현저장소에 연결한다. Object Key에서 객체어댑터 이름을 알아낸 다음 서버 테이블에서 색인으로 사용되어진다. 2.2.1의 일시적 참조의 바인딩과 마찬가지로 네 가지의 시나리오 중 하나의 경우에 해당한다.

첫째, 만약 서버가 등록되어있지 않다면 저장소는 응용프로그램 코드에 따라 클라이언트에게 OBJECT_NOT_EXIST라는 예외처리를 돌려보낸다.

둘째, 서버가 수동으로 시작하도록 등록되어있고 아직 실행되고 있지 않은 경우는 저장소가 TRANSIENT 예외처리로 리턴 한다.

셋째, 만약 자동으로 시작하도록 등록되어 있고 아직 실행중이 아니라면 저장소는 서버를 시작하고 서버의 주소를 제공하는 서버로부터 메시지를 받을 때까지 기다린다.

넷째, 만약 서버가 실행중이라면 저장소는 클라이언트에게 실제의 서버위치의 호스트이름과 포트번호를 알려주는 location-forward라는 응답을 리턴 한다.

그림3은 서버가 자동으로 시작해서 성공적인 persistent reference의 바인딩을 위한 연속적인 상호동작을 보여준다[2].

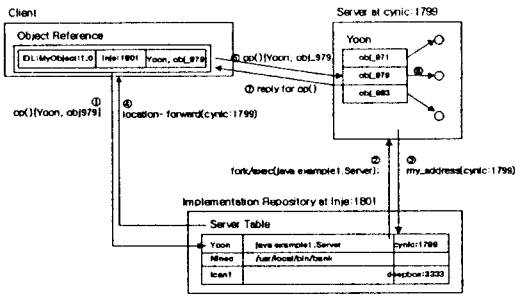

그림 3 서버가 자동으로 실행하는 영구적 참조의 바인딩

클라이언트는 구현저장소에 대해서 혹은 서버가 자동으로 시작하는지 어떤지에 대해서 전혀 알지 못하고 단지 요청을 보내 뒤 응답을 기다릴 뿐이다.

3. 구현저장소 인터페이스의 설계

3.1 인터페이스의 엔트리정보

CORBA 스펙에는 단지 구현저장소의 정의만 내려져 있고, 이것의 인터페이스는 정의되어 있지 않다. 일반적인 구현저장소의 엔트리에는 다음과 같은 정보가 포함될 수 있다[5].

- 서버이름(디렉토리 구조)을 가진다.
- 기본활성화 모드(공유, 비공유, per-메소드)
- 보조활성화 모드(per-클라이언트, per-클라이언트-프로세스, 다중-클라이언트)
- 서버가 영구 서버인지 아닌지에 대한 정보
- 서버 소유권자, 서버 허가 권한

여기서 본 논문에서 사용된 구현저장소 인터페이스의 엔트리정보에는 다음의 표2와 같다.

구성요소	설 명
srv_name	서버의 유일한 이름
mode	활성화 정책 모드
type_ids	IDL의 repository ID들
class_name	구현객체의 클래스 경로와 파일

표 2 구현저장소의 엔트리 구성요소

3.2 JacORB의 구현저장소 IDL 인터페이스

구현객체들의 엔트리정보를 찾거나, 생성, 소멸시킬 수 있는 메소드를 제공해야 한다. Free ORB인 JacORB을 이용하여 구현하였다. 구현저장소의 IDL은 그림4에 나타나 있다.

인터페이스 ImplementationDef에는 서버이름, 활성화 정책(공유, 비공유, per-메소드, 영구), 저장소 ID의 리스트, 클래스위치와 이름을 정의하고, 인터페이스 ImplementationRepository에는 객체를 등록하거나 해제, 객체의 정보를 찾아볼 수 있는 메소드를 정의하고 있다.

```
module CORBA {
    ...
/* Implementation Repository Entry */

  interface ImplementationDef {

      enum ActivationMode {
          shared,
          unshared,
          per_method,
          persistent
      };

      typedef sequence<string> type_id_list;

      readonly attribute string srv_name;
      attribute ActivationMode mode;
      attribute type_id_list type_ids;
      attribute string class_name;
  };

  /*  Implementation Repository    */

  interface ImplementationRepository {

  typedef sequence<ImplementationDef> ImplementationDefSeq;

      ImplementationDef create (
          in ReferenceData  id,
          in InterfaceDef intf,
          in ImplementationDef impl
      );
      void dispose (in ImplementationDef impl);

      ImplementationDef register (
          in string srv_name,
          in int mode,
          in string typeId,
          in string impl
      );
      ImplementationDef unregister ( in string srv_name );

      ImplementationDef get_id (in ImplementationDef impl );
      ImplementationDefSeq get_name (in string srv_name );
      ImplementationDefSeq get_all ();

      void change_implementation (
          in Object obj,
          in ImplementationDef impl
      );
    };
    ...
};
```

그림 4. 구현저장소의 IDL

3.2 구현객체 관리

구현객체의 엔트리의 생성은 putit이라는 명령어로 할 수 있고 다음의 표3과 같은 명령어들로 엔트리는 관리되어 진다.

명령어	설 명
listit <list>	구현저장소에 있는 모든 엔트리의 이름을 보여준다.
catit <name>	주어진 이름의 상세한 엔트리 정보를 보여준다.
dellit <name>	주어진 이름의 엔트리를 삭제한다.
rmit	구현저장소의 엔트리를 제거한다.

표 3 객체관리를 위한 명령어

4. 결론 및 향후 연구 과제

본 논문은 CORBA 스펙에서 아직 인터페이스가 정의되어 있지 않은 효율적인 분산객체를 관리할 수 있는 구현저장소와 관련된 개념을 살펴보았다. 그리고 이러한 개념을 기반으로 인터페이스를 설계하고 IDL로 정의하였다.

향후 정의된 IDL을 개선하고 이를 이용하여 위의 표3의 명령어들을 구현하고 이런 명령어를 통합하여 보안도 고려한 구현저장소 브라우저를 구현할 계획이다.

참고문헌

[1] Irfan Pyarali and D. C. Schmidt, "An Overview of the CORBA Portable Object Adapter",ACM, 1998

[2] Michi Henning, "Binding, Migration, and Scalability in CORBA", http://danzon.cs.wustl.edu/binding.pdf

[3] Gerald Brose, "JacORB Programming Guide, v0.9"(March 1998), ftp://ftp.inf.fu-berlin.de/pub/jacorb/doc/ProgGuide_0.9.ps.gz

[4] TAO Implementation Repository, http://siesta.cs.wustl.edu/~schmidt/corba.html

[5] 왕창종 외 1명, "분산객체 컴퓨팅 기술 CORBA 프로그래밍", 대림, 1998