코바 프로그래밍[1]
코바 프로그램 이것만은 알고 하자.

코바 프로그램이라는 주제를 가지고 여섯번의 연재를 통해 독자를 만나게 되었다. 여러 번의 기사를 통해 기본적인 내용을 접해온 독자도 있을 것이고 코바의 개념이나 용어만 들어본 독자도 있을 것이다. 연재는 기본적으로 코바의 개념과 샘플 코드 정도는 작성해본 개발자 위주로 진행될 것이다.

프로그램 환경

연재에 소개되는 내용을 따라 하기 위해서는 비지브로커 C++와 자바를 설치해야 하며 기본적인 C++, 자바 개발환경이 갖추어져 있어야 한다. 비지브로커는 웹을 통해서 평가판을 다운 받을 수 있는데, 비지브로커 C++는 3.3 이 가장 최신 버전이며 비지브로커 자바는 3.4 가 가장 최신 버전이다. 부록 CD 에 제공되는 vbcpp33.exe 는 C++용 설치 프로그램이고, vbj33.exe 와 vbj34.exe 는 자바용 설치 프로그램이다. 두 가지 중에서 하나만 설치하면 되는데, 비지브로커 자바 3.3 버전 까지는 JDK1.2 에서 제공하는 자바 ORB(Sun 에서 제공하는 CORBA 프로그램)와 충돌이 생긴다. 따라서 자바 ORB 를 지원하는 JDK1.2 를 사용하는 사용자라면 반드시 vbj34.exe 를 설치하기 바란다. 이와 함께 추가적으로 사용자가 알아야 할 것이 있는데, 비지브로커 C++ 3.3 과 비지브로커 3.4 을 같은 컴퓨터에 설치할 때 같은 디렉토리에 설치해서는 안 된다는 것이다. 비지브로커 자바 3.4 버전에서 JDK1.2 와의 충돌을 해결하기 위해 비지브로커 라이브러리를 약간 변경했는데, 그 이유 때문에 비지브로커 C++ 3.3 과 비지브로커 자바 3.4 를 같은 디렉토리에 설치하면 idl2cpp.exe 명령이 실행되지 않는다. 이 문제를 해결하기 위해서는 비지브로커 C++ 3.3 과 비지브로커 자바 3.4 를 다른 디렉토리에 설치해야만 한다. 자세한 설치과정은 코바사랑(http://www.object.pe.kr/) 홈페이지에서 다운 받아 보기 바란다.

필자가 연재를 통해 제공하게 될 소스를 작성한 환경은 다음과 같다.

언어 : C++/ 자바
개발툴 : Visual C++ 6.0 (버전은 4.1 이상이면 된다.) / JDK1.2
CORBA 제품 : 비지브로커 C++ 3.3/자바 3.4
O/S : Window 95/98, NT

코바 프로그램 실행원리(OSAgent 의 동작방식)

분산객체 프로그램을 작성한 후 실행하는 과정은 대개 비슷하며, 비지브로커의 경우도 크게 다르지 않다. 실행원리를 제대로 이해하기 위해서는 가장 중요한 OSAgent 의 동작방식을 이해하는 것이 중요하다.
OSAgent(Smart Agent)는 동적인 분산 디렉토리 서비스 데몬 프로그램으로 간단한 네이밍 서비스를 제공하여 클라이언트가 객체를 발견하는 메커니즘을 제공한다. 오빅스 데몬과 비교하여 서버 객체의 자동 활성화 기능(OAD 데몬을 통해 해결)이 빠지고, 유연한 객체 검색기능이 추가되었다. Visibroker 에서는 구현객체를 OSAgent 에 등록해야만 클라이언트에서 bind 오퍼레이션을 통해 구현객체를 찾거나 객체에서 제공하는 오퍼레이션을 수행할 수 있게 된다.

서버 정보 관리

OSAgent 의 가장 주된 기능은 객체의 레퍼런스를 얻기 위해 이름을 주면 이름에 맞는 등록된 객체의 IOR 정보를 넘겨주는 것이다. OSAgent 를 실행시키면 정해진 UDP 포트(디폴트는 14000 번)로 객체의 요청을 기다린다. 그런 후 서버 객체를 실행시키면 서버 객체의 정보가 OSAgent 에 전달되게(obj_is_ready 오퍼레이션을 통해) 되며, OSAgent 는 객체 정보를 리스트에 추가하여 관리하게 된다.
클라이언트가 실행이 될 때도 가장 먼저 OSAgent 에게 원하는 객체를 찾아 달라고 요청을 하며(bind 오퍼레이션을 통해), OSAgent 는 클라이언트가 원하는 객체의 정보를 IOR 로 변환하여 넘겨주게 된다. 클라이언트는 이 IOR 정보를 이용하여 다시 서버로 리퀘스트를 보내게 된다. 만약 등록된 서버가 정상적으로 종료가 될 경우 서버는 OSAgent 에 자신의 정보를 삭제해 달라고 요청(deactive_obj 오퍼레이션을 통해)한다. 정상적으로 종료되지 않을 경우(NT 의 경우 작업관리자로 프로세스를 종료시킬 때, 유닉스의 경우 kill –9 명령으로 강제 종료시킬 때) OSAgent 에게 삭제 요청을 보내지 못하고 종료되어 OSAgent 는 서버가 정상적으로 동작하고 있는 것으로 알고 있게 된다. 이 경우 OSAgent 가 리스트에서 서버 정보를 삭제하는 시기는 클라이언트가 연결이 되지 않는다고 정보를 넘겨줄 때이다.

서버/클라이언트가 OSAgent 와 주고 받는 메시지는 모두 UDP 프로토콜을 사용하며, OSAgent 를 찾기 위해서는 브로드 캐스트 방식을 취한다. (참고로 JDK1.1 버전 이하에서는 UDP 브로드캐스트 메시지를 지원하지않기 때문에 Java 어플리케이션은 이런 방법으로 OSagent 를 찾을 수 없다. 문제를 해결하기 위해 Agent 를 찾아 주는 vbj 유틸리티를 제공한다) 따라서, OSAgent 가 응답을 할 수 있는 도메인은 서브넷이 같고 동일한 포트를 사용하고 있는 LAN 으로 제한되어 있다. 이러한 조건만 만족한다면 OSAgent 는 도메인에 하나 이상만 실행이 되고 있으면 된다. 여기서 잊지 말아야 할 것은 OSAgent 와의 통신은 UDP 프로토콜을 사용하지만 서버와 클라이언트간의 통신은 TCP 프로토콜을 사용한다는 것이다. 이것은 동기호

출을 기본으로 사용하는 코바에서 당연히 예상할 수 있는 결과일 것이다.

다른 네트워크에 있는 OSAgent 와 통신

OSAgent 가 연결 가능한 도메인은 서브넷과 UDP 포트가 같을 경우라고 하였다. 또한 OSAgent 를 찾을 때 브로드캐스트를 한다고 하였는데, 다른 서브넷의 OSAgent 에 연결하거나 브로드캐스트를 사용하지 않는 방법은 없을까? 그림의 경우처럼 Local Network #1 에는 OSAgent 가 실행이 되지 않아 Local Network #2 의 OSAgent 에 연결을 해야 한다.

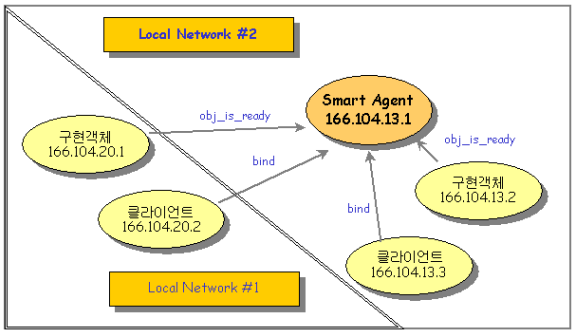

그림 1. 다른 네트워크에 있는 OSAgent 와 통신 (osagent6.gif)

이 경우 OSAGENT_ADDR 환경변수를 설정해 주면 된다. 윈도우 NT 의 경우 [제어판/시스템/환경]에 OSAGENT_ADDR 을 OSAgent 가 실행되고 있는 IPAddress 로 설정해주고, 윈도우 95/98 의 경우 autoexec.bat 파일에

```
set OSAGENT_ADDR=166.104.13.1
```

과 같이 설정한다.
유닉스의 경우

```
prompt> setenv OSAGENT_ADDR =166.104.13.1
```

과 같이 설정한다.
환경변수로 설정을 하지 않을 경우 클라이언트나 서버를 실행시킬 때 직접 옵션으로 명시해 주어도 된다.

```
prompt>Server –ORBAgentaddr 166.104.13.1
prompt>Client –ORBAgentaddr 166.104.13.1
```

만약 명시되어 있는 IPAddress 에서 OSAgent 의 응답을 얻지 못할 경우 다시 브로드 캐스트

를 하게 된다. OSAGENT_ADDR 환경변수나 –ORBAgentaddr 옵션은 서버/클라이언트가 OSAgent 와 통신할 때 사용하는 것이고, 다른 서브넷에 존재하는 OSAgent 와 OSAgent 간에 통신할 때 사용하는 환경변수는 OSAGENT_ADDR_FILE 이다. 이 경우 Local Network #1 과 #2 에 있는 OSAgent 가 서로 연결되어 양쪽 서브넷이 합해지는 결과를 볼 수 있다. 이때 설정은 Local Network #1 의 OSAgent 가 실행되고 있는 호스트의 OSAGENT_ADDR_FILE 환경변수에 Local Network #2 에서 실행되고 있는 OSAgent 의 IPAddress 가 적힌 파일을 설정해 주고. Local Netwokr #2 에는 반대로 설정해준다.

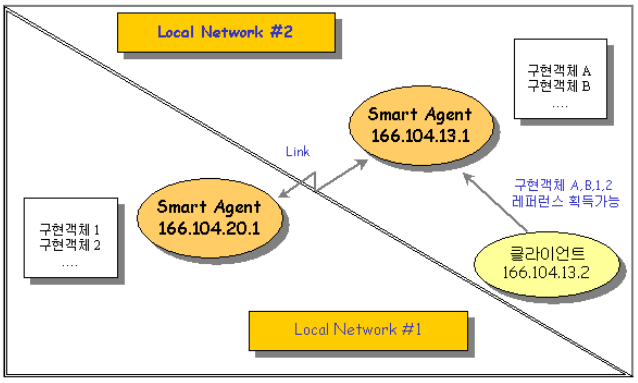

그림 2.OSAgent 와 OSAgent 통신 (osagent7.gif)

다음과 같이 환경 설정을 해주도록 하자. 윈도우 NT 의 경우 [제어판/시스템/환경]에 OSAGENT_ADDR_FILE 을 C:\inprise\vbroker\adm\agentaddr 로 설정해주고, 윈도우 95/98 의 경우 autoexec.bat 파일에

set OSAGENT_ADDR_FILE=C:\inprise\vbroker\adm\agentaddr

과 같이 추가 해준다.
유닉스의 경우
prompt> setenv OSAGENT_ADDR_FILE=/users/vbroker/adm/agentaddr

과 같이 설정한다.

파일의 내용은 OSAgent 의 IPAddress 를 적어준다.

//agentaddr
203.236.202.1

그리고 다른 포트로 실행되고 있는 OSAgent 와 통신할 경우 –ORBAgentport 를 주고서 실행

하거나 아예 환경변수에 OSAGENT_PORT 를 설정하면 된다.

```
prompt>Server –ORBAgentport 14001
```

이외에도 서버가 클라이언트와 사용하게 될 TCP 포트를 명시할 수 있는데, 서버를 실행할 때 –OAPort 를 주면 된다. 아래의 경우 서버는 TCP 14001 로 활성화 된다.

```
prompt>Server –ORBAgentport 14001
```

Fault-tolerance 와 로드밸런싱

OSAgent 는 Fault-tolerance 기능과 로드밸런싱 기능을 가진다. Fault-tolerance 기능은 두 가지로 생각해 볼 수 있는데 첫번째는 같은 구현 객체가 포함된 여러 개의 서버가 실행이 되고 있을 때 클라이언트가 bind( ) 오퍼레이션을 이용해 얻어온 서버가 실행이 종료되거나 네트워크 연결이 안될 경우 OSAgent 는 같은 기능을 하는 다른 서버로 연결을 해준다. 이 경우 클라이언트는 이런 내부적인 메커니즘을 느끼지 못한 채 계속 서비스 이용이 가능하다. 그림 3 의 경우 처음 클라이언트와 연결된 서버는 1 번인데, 1 번이 사용 가능하지 않게 되자 내부적인 메커니즘에 의해 다시 2 번으로 연결을 시켜준다. 이러한 Fault-tolerance 기능은 enable_rebind 옵션이 true 로 설정되어 있어야만 가능하며, 디폴트는 true 이다.

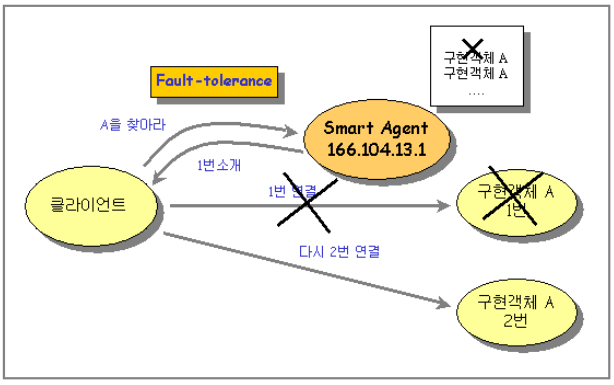

그림 3. Fault-tolerance 기능 1 (osagent9.gif)

두번째는 OSAgent 의 프로세스가 종료되거나 네트워크 연결이 안되었을 경우이다. 모든 서버는 OSAgent 에 등록을 한 후 OSAgent 가 정상적으로 동작하고 있는지 15 초에 한번씩 체크를 한다. 이때 OSAgent 가 응답을 하지 않으면 서버들은 다시 신호를 보내고 그래도 응답하지 않으면 자신들의 위치정보를 관리해줄 새로운 OSAgent 를 찾기 위해 브로드캐스트 메시지를 보낸다. 이때 응답을 하는 OSAgent 가 있을 경우 자신의 위치정보를 등록하게 되며, 응답을 하지 않을 경우 응답을 할때까지 15 초에 한번씩 신호를 보낸다.

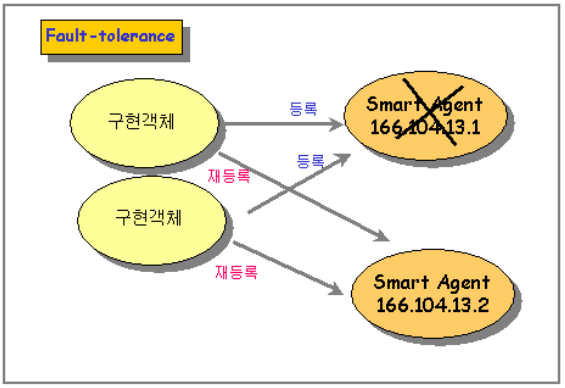

그림 4. Fault-tolerance 기능 2 (osagent10.gif)

로드밸런싱 기능은 같은 서버가 여러 개가 서비스되고 있을 경우 클라이언트 요청이 오면 라운드 로빈 방식을 사용하여 순서대로 돌아가면서 연결을 시켜주는 기능이다. 그림의 경우처럼 같은 서버가 세 개가 실행되고 있을 때 처음 요청은 1 번으로 연결하고 다음 요청은 2 번, 3 번 차례대로 연결시켜 주는 기능을 말한다.
또한 OSAgent 의 동작원리를 이해할 때 주의해야 할 것은 OSAgent 는 클라이언트가 서버의 위치를 찾아 통신을 시작하면 메시지를 전달하는 것에는 개입하지 않는다는 것이다.

// 박스 처리 //
[서버객체와 서버프로그램]
기존에는 클라이언트와 서버 프로그램이 객체단위로 프로그램 되어 있지 않았다. 하지만, 분산 객체 환경에서는 클라이언트와 서버 프로그램 모두 객체단위로 이루어지기 때문에 서버 프로그램을 서버 객체 혹은 구현객체(Object Implementation)라는 용어로 표현하기도 한다. 좀더 정확한 언어로 표현을 하자면, CORBA 인터페이스를 구현하고 있는 클래스를 구현객체라고 하며, 구현객체를 등록하여 클라이언트의 요청을 기다리는 프로그램을 서버 프로그램, 서버 프로그램을 통해 구현 객체를 호출하는 프로그램을 클라이언트 프로그램이라고 한다. 보통 구현객체와 서버 프로그램을 하나로 묶어서 서버라고 얘기하기도 하고, 구현객체를 서버라고 표현하기도 한다. 여기서도 서버객체와 서버프로그램, 서버라는 용어를 혼용해서 사용하고 있기 때문에 용어상의 혼란이 없기 바란다.
//

다양한 프로그램

본론으로 들어가서 문자열과 시퀀스 형을 코바 C++와 자바로 어떻게 전송하는지 살펴보기로 하자. 코바에서 이루지는 데이터 전달 방식은 크게 in, inout, out, return 네 가지로 이루어

진다.

문자열 전송

문자열을 전송하는 방식은 여러 가지가 있을 수 있다. char 형 배열을 사용하거나 char 시퀀스형을 사용할 수도 있는데, char 형 배열의 경우 Null 문자를 고려하여 크기를 항상 신경써야 하고, 시퀀스의 경우도 실행 시에 크기를 결정해줘야 하기 때문에 추천할 만한 방법은 아니다. 결론적으로 문자열을 전송하는데 가장 많이 사용하는 방법은 string 타입이다. string을 사용할 때 주의할 점은 C++ 프로그램과 자바 프로그램을 혼합하여 사용할 경우 유니코드를 위해 wstring 을 사용해야 한다는 것이다. 이 경우 C++ 측에서는 ANSI 스트링과 유니코드를 변환하는 코드를 추가해야 하는 불편함이 있다.

```
// Sample.idl
module Sample
{
    interface Manager
    {
//          wstring oper1(in wstring inString, inout wstring inoutString, out wstring outString); //Java
            string oper1(in wstring inString, inout wstring inoutString, out wstring outString); //C++
    };
};
```

자바와 C++의 구현 객체, 클라이언트 코드를 어떻게 작성하는지 차례대로 살펴보기로 하자. 먼저 자바의 경우 idl2java 컴파일러를 통해 Holder 클래스와 Helper 클래스가 생성이 된다.

[Holder 클래스]
자바에서 모든 파라미터는 값에 의한 전달을 하기 때문에 참조에 의한 전달이 필요한 inout, out 타입의 경우 Holder 클래스를 사용해야 한다. 클라이언트와 서버가 통신할 때 실제 타입을 전달하는 것이 아니라 Holder 클래스 인스턴스를 주고 받게 된다. 클라이언트 측에서는 인스턴스를 생성하고 오퍼레이션을 호출할 때 인자로 넘겨주며, 서버는 Holder 클래스의 value 을 통해 값을 설정하거나 변경하게 된다.

[Helper 클래스]
코바에서는 모든 사용자 정의형 타입에 Helper 클래스를 제공한다. Helper 클래스에는 객체를 다루기에 유용한 오퍼레이션들이 포함되어 있다. 이처럼 Helper 클래스를 제공하는 이유는 매핑되는 클래스마다 필요하지도 않는 여러 가지 오퍼레이션을 포함할 경우 코드가 무거워

지기 때문이다.

Helper 클래스에는 객체 레퍼런스를 얻어오는 bind 오퍼레이션 뿐만 아니라 객체정보를 스트림으로 기록하고 읽어 들이는 오퍼레이션 등을 가지고 있다.

[자바 구현객체 코드]

inout, out 의 경우 StringHolder 를 사용하고 있으며 값을 넣거나 빼낼 때 value 를 이용한다.

```
public String oper1(String inString, StringHolder inoutString, StringHolder outString)
{
        //in 값 출력
         System.out.println("in 출력(inString):" + inString);
         //inout 값 출력및 변경
         System.out.println("inout 출력(inoutString):" + inoutString.value);
        inoutString.value = "변경된 inout string 입니다";
         //out 값 할당
        outString.value = "out string 입니다";
         //return 선언및 생성, 할당
        return "return string 입니다";
 }
```

[자바 클라이언트 코드]

객체의 레퍼런스를 얻을 때 ManagerHelper 클래스에 있는 bind( ) 을 이용하고 있으며, 클라이언트에서는 서버 호출에서 inout, out 의 경우 StringHolder 를 이용한다.

```
//객체 레퍼런스 획득
Manager manager = ManagerHelper.bind(orb,"manager");
//in, return 변수 선언
String inString, retString;
//in 값 할당
inString = "in string 입니다";
//inout 변수 선언및 할당
StringHolder inoutString = new StringHolder("inout string 입니다");
//out 변수 선언및 생성
StringHolder outString = new StringHolder();
//함수 호출
retString =  manager.oper1(inString, inoutString, outString);
```

```
//inout 값 출력
System.out.println("inout 출력(inoutString):" + inoutString.value);
//out 값 출력
System.out.println("out 출력(outString):" + outString.value);
//return 값 출력
System.out.println("return 출력(retString):" + retString);
```

[코바 C++에서 제공하는 문자열 다루는 함수]

IDL 에서 정의한 string 형은 C++의 char* 로 매핑된다. 따라서, 문자열이 리턴값이나 매개변수로 전송될 때는 데이터를 저장하는 메모리를 동적으로 할당해야 하며 필요에 따라 메모리를 삭제해야 한다. 코바에서는 이를 위해 세가지 함수를 제공한다.

```
// CORBA 네임스페이스에 정의됨
// 문자열을 넘겨받아 같은 크기의 메모리를 할당하고 같은 내용으로 초기화 한다.
char * string_dup(const char*);
// 지정된 길이만큼의 메모리만 할당한다.
char * string_alloc(CORBA::ULong len);
// 할당받은 메모리를 해제한다.
void   string_free(char*);
```

[C++ 구현객체 코드]

코바에서는 문자열을 쉽게 다루기 위해 String_var 클래스를 제공한다. _var 클래스는 문자열 외에도 각 인터페이스, 가변길이의 복합 데이터형(예를 들어 문자열 배열, 인터페이스의 시+퀀스 등), 고정길이의 복합 데이터형에도 제공되며, 메모리 관리를 쉽게 하기 위해 코바 C++ 프로그램시 많이 사용하게 된다.

```
char* oper1(const char* inString,char*& inoutString,char*& outString)
{
        //in 값 출력
        cout << "in 출력(inString):" << inString << endl;
        //inout 값 출력및 변경
        cout << "inout 출력(inoutString):" << inoutString << endl;
        CORBA::string_free(inoutString);
        inoutString = CORBA::string_dup("변경된 inout string 입니다");
        //out 값 할당
        outString = CORBA::string_dup("out string 입니다");
```

```
        //return 선언및 생성
        return CORBA::string_dup("return string 입니다");
}
```

[C++ 클라이언트 코드]

String_var 를 사용하지 않고 char *로 선언한 경우 반드시 stinrg_free( ) 을 이용하여 할당된 메모리를 해제해 주어야 한다. 즉,

```
CORBA::String_var inString = CORBA::string_dup("in string 입니다.");
```

대신에

```
char* inString = CORBA::string_dup("in string 입니다.");
CORBA::string_free(inString);
```

과 같이 사용한다는 얘기이다.

```
// 객체 레퍼런스 획득
Sample::Manager_var manager = Sample::Manager::_bind("manager");
//in 변수 선언 및 할당 : (1),(2),(3)번 모두 가능.
//in 의 경우 굳이 String_var 를 사용하지 않아도 된다.
//(1)번 방법
const char * inString = "in string 입니다";
//(2)번 방법
//CORBA::String_var inString = (const char *) "in string 입니다";
//(const char*)를 사용하지 않으면 inString 이 삭제될때 메모리 에러가 발생할 수 있다.
//(3)번 방법
//CORBA::String_var inString = CORBA::string_dup("in string 입니다.");
//inout, out, return 변수 선언
CORBA::String_var inoutString, outString, retString;
//inout 값 할당 : (1), (2)번 모두 가능
//(1)번 방법
//inoutString = CORBA::string_dup("inout string 입니다.");
//(2)번 방법
inoutString = (const char *)"inout string 입니다.";
```

```
//함수 호출
retString =   manager->oper1(inString, inoutString, outString);
//inout 값 출력
cout << "inout 출력(inoutString):" << inoutString  << endl;
//out 값 출력
cout << "out 출력(outString):" << outString<< endl;
//return 값 출력
cout << "return 출력(retString):" << retString << endl;
```

시퀀스형 전송

이번에는 시퀀스형 전송에 대해 살펴보기로 하자. 코바 프로그램을 할 때 구조체 리스트 형태의 데이터를 전송하는 경우가 많이 발생한다. 예를 들어 학사관리 프로그램을 구현한다고 할 때 학생들의 정보를 보여주는 업무를 구현하기 위해서는 학생들의 정보를 구조체 리스트 형태로 조합하여 전송하게 된다. 먼저 IDL 에서 Person 이라는 구조체를 정의한 다음 Person 의 일차원 가변 배열 타입인 시퀀스 형을 이용하여 PersonSeq 를 정의하게 된다. Manager 가 제공하는 oper1 오퍼레이션은 PersonSeq 를 네 가지 전송 방식으로 주고 받는다.

```
// Sample.idl
module Sample
{
    //학생정보
    struct Person
    {
        short id;
        wstring name;
    };
    typedef sequence<Person> PersonSeq;
    interface Manager
    {
        PersonSeq oper1( in PersonSeq inPersonSeq, inout PersonSeq inoutPersonSeq,
                         out PersonSeq outPersonSeq);
    };
};
```

[자바 구현객체 코드]

자바 구현객체 코드를 살펴보면 가장 눈에 띄는 것이 배열을 리턴하는 코드이다. C++의 경

우 리턴 타입으로 배열형이 오면 안되지만 자바의 경우는 배열형을 리턴하는 것이 가능하다.

```
Person[] retPersonSeq = new Person[2];        ---------- (1)
for(i =0; i < retPersonSeq.length; i++)
{
                retPersonSeq[i] = new Person((short)4, "name4");
}
```

시퀀스형은 자바의 배열 타입으로 매핑이 되며 자바에서 배열은 객체와 똑같이 취급되어, (1)번 방법과 같이 인스턴스를 생성하고, 배열의 크기만큼 루프를 돌며 실제 값을 채워주게 된다. 자바에서는 new 를 사용하지 않으면 실제 객체 인스턴스의 메모리가 할당되지 않아 인스턴스를 사용하려고 할 경우 초기화 에러가 발생하게 된다.

```
Person[] tempIoPersonSeq = inoutPersonSeq.value;          ----------(1)
outPersonSeq.value = tempOPersonSeq;                      ----------(2)
```

위의 코드처럼 inout, out 의 경우 Holder 클래스를 이용하기 때문에 value 멤버 데이터를 이용하여 접근하게 된다. (1)번의 코드의 경우는 클라이언트에서 넘어 온 데이터를 읽는 것이고 (2)번 코드의 경우는 데이터를 쓰는 것이다.

```
public Person[] oper1(Person[] inPersonSeq,PersonSeqHolder inoutPersonSeq,PersonSeqHolder
outPersonSeq)
{
        //in 값 출력
          int i;
         for(i =0; i < inPersonSeq.length; i++)
          {
                    System.out.println("in 출력(inPersonSeq)-id: " + inPersonSeq[i].id +
                                        ", name:" + inPersonSeq[i].name);
          }
          //inout 값 출력 및 변경
         Person[] tempIoPersonSeq = inoutPersonSeq.value;
           for(i =0; i < tempIoPersonSeq.length; i++)
           {
                    System.out.println("inout 출력(inoutPersonSeq)-id: " + tempIoPersonSeq[i].id
                                + ", name:" + tempIoPersonSeq[i].name);
```

```
                tempIoPersonSeq[i] = new Person((short)22, "변경된 name2");
        }
        inoutPersonSeq.value = tempIoPersonSeq;
          //out 값 할당
          Person[] tempOPersonSeq = new Person[2];
          for(i =0; i < 2; i++)
          {
                tempOPersonSeq[i] = new Person((short)3, "name3");
          }
          outPersonSeq.value = tempOPersonSeq;
           //return 값 선언및 생성
           Person[] retPersonSeq = new Person[2];
           for(i =0; i < retPersonSeq.length; i++)
          {
                retPersonSeq[i] = new Person((short)4, "name4");
           }
           return retPersonSeq;
}
```

[자바 클라이언트 코드]

클라이언트의 경우 매핑 언어가 C++인지 자바인지에 따라 약간의 차이가 발생하는 부분이 있는데, 그 중 하나가 out 타입 처리이다.

```
Sample::PersonSeq_var outPersonSeq;                    -------(1)
PersonSeqHolder outPersonSeq = new PersonSeqHolder();      -------(2)
```

(1)번 방법처럼 C++의 경우 out 타입은 클라이언트에서 타입 선언하면 되지만, 자바의 경우는 (2)번 방법처럼 실제 메모리까지 할당해주어야 한다.

```
//객체 레퍼런스 획득
Manager manager = ManagerHelper.bind(orb,"manager");
//in 변수 선언및 생성
int i;
Person[] inPersonSeq = new Person[2];
for(i =0; i < inPersonSeq.length; i++)
{
```

```
        inPersonSeq[i] = new Person((short)1, "name1");
}
//inout 변수 선언및 생성
Person[] tempIoPersonSeq = new Person[2];
for(i =0; i < tempIoPersonSeq.length; i++)
{
        tempIoPersonSeq[i] = new Person((short)2, "name2");
}
//(1),(2)번 방법 모두 가능
/* //(1)번 방법
PersonSeqHolder inoutPersonSeq = new PersonSeqHolder();
inoutPersonSeq.value = tempIoPersonSeq;*/
//(2)번 방법
PersonSeqHolder inoutPersonSeq = new PersonSeqHolder(tempIoPersonSeq);
//out, return 변수 선언및 생성
PersonSeqHolder outPersonSeq = new PersonSeqHolder();
Person[] retPersonSeq;
//함수 호출
retPersonSeq =  manager.oper1(inPersonSeq, inoutPersonSeq, outPersonSeq);
//inout, out 값 출력

… 중간 코드 생략 …

//return 값 출력
for(i =0; i < retPersonSeq.length; i++)
{
          System.out.println("return 출력(retPersonSeq)-id: " + retPersonSeq[i].id +
                    ", name:" + retPersonSeq[i].name);
}
```

[C++ 구현객체 코드]

데이터형(기본형, 가변길이의 복합형, 고정길이의 복합형)과 객체형의 시퀀스 타입을 전송하는 방법은 모두 같다. 여러 가지 방법이 있겠지만 시퀀스형의 인스턴스를 생성한 후 값을 채워주는 방법을 가장 많이 사용한다. 인스턴스를 생성할 때 in, inout 의 경우 매핑된 코드가 포인터를 사용하지 않기 때문에 굳이 new 를 사용할 필요가 없으며, out 의 경우에는 new 를 사용하여 동적으로 메모리를 할당하게 된다.

```
Sample::PersonSeq_var newSeq = new Sample::PersonSeq(2);    ------(1)
//Sample::PersonSeq_var newSeq = new Sample::PersonSeq;      -------(2)
newSeq->length(2);                                          -------(3)
```

즉, 위의 코드의 경우 (1)번의 과정은 일차원 가변 길이 배열이 가질 수 있는 최대 길이를 명시하는 과정이며 실제 메모리 할당은 (3)번의 과정을 통해 이루어 진다. (2)번처럼 배열이 가지는 최대 길이를 명시하지 않아도 아무런 문제가 없으며, (3)번의 과정을 생략할 경우 실제 메모리가 잡히지 않기 때문에 메모리 에러가 발생하게 된다. 그리고, _var 클래스에는 포인터가 들어 올 경우 _var 타입으로 변환시켜 주는 할당 연산자가 오버로딩 되어 있기 때문에

_var 타입 = 포인터 타입

과 같은 문법이 가능하게 된다.

```
//return 값 선언 및 생성
Sample::PersonSeq_var retPersonSeq = new Sample::PersonSeq(2);
retPersonSeq->length(2);                          ---------------------- (1)
… 값 채우는 과정 생략 …
return retPersonSeq._retn();                      ---------------------- (2)
```

또, 하나 주의해서 봐야 할 코드가 있는데 .과 ->의 사용을 구별하는 것이다. retPersonSeq 는 PersonSeq_var 타입으로 선언되었다. (2)번의 경우처럼 PersonSeq_var 클래스의 멤버 함수인 _retn( )을 호출하기 위해서는 .을 사용하며, PersonSeq 클래스의 멤버 함수인 length( )를 사용하기 위해서는 ->를 사용한다. ->는 PersonSeq_var 클래스가 가지는 연산자 오버로딩으로서 PersonSeq_var 의 멤버 데이터인 PersonSeq 포인터 또는 PersonSeq_ptr 을 이용할 수 있게 해준다.

```
Sample::PersonSeq* oper1( const Sample::PersonSeq& inPersonSeq,
            Sample::PersonSeq& inoutPersonSeq,   Sample::PersonSeq_ptr& outPersonSeq)
{
        //in 값 출력
        CORBA::ULong i; // int i 라고 선언하면..[]오퍼레이터에서 에러가 발생한다.
       for(i =0; i < inPersonSeq.length(); i++)
        {
```

```
                cout << "in 출력(inPersonSeq)-id: " << inPersonSeq[i].id <<
                        ", name:" << inPersonSeq[i].name<< endl;
            }
            //inout 값 출력 및 변경
            for(i =0; i < inoutPersonSeq.length(); i++)
            {
                cout << "inout 출력(inoutPersonSeq)-id: " << inoutPersonSeq[i].id
                        << ", name:" << inoutPersonSeq[i].name<< endl;
                inoutPersonSeq[i].id = 22;
                inoutPersonSeq[i].name = CORBA::string_dup("변경된 name2");
        }
            //out 값 할당
/*          //(1)번 방법
            outPersonSeq = new Sample::PersonSeq(2);
            outPersonSeq->length(2);
            for(i =0; i < outPersonSeq->length(); i++)
            {
                (*outPersonSeq)[i].id = 3;
                (*outPersonSeq)[i].name = CORBA::string_dup("name3");
            }*/
            //(2)번 방법
            Sample::PersonSeq_var newSeq = new Sample::PersonSeq(2);
            newSeq->length(2);
            Sample::Person_var person = new Sample::Person();
            for(i =0; i < newSeq->length(); i++)
            {
                person->id = 3;
                person->name = CORBA::string_dup("name3");
                newSeq[i] = person;
            }
            outPersonSeq = newSeq._retn();
            //return 값 선언및 생성
            Sample::PersonSeq_var retPersonSeq = new Sample::PersonSeq(2);
            retPersonSeq->length(2);
            for(i =0; i < retPersonSeq->length(); i++)
            {
```

```
            retPersonSeq[i].id = 4;
            retPersonSeq[i].name = CORBA::string_dup("name4");
        }
        return retPersonSeq._retn();
}
```

[C++ **클라이언트 코드**]

in, inout **의 경우 포인터를 사용하지 않기 때문에 굳이** _var **타입을 사용하지 않아도 된다. 따라서, 그냥 객체를 생성하는** (1)**번 방법과 동적으로 생성하는** (2)**번 방법 모두 가능하다.**

```
Sample::PersonSeq inPersonSeq;                                  ---------- (1)
Sample::PersonSeq_var inPersonSeq = new Sample::PersonSeq(2);   ---------- (2)
```

생성된 인스턴스에 값을 채워주는 방법은 (1)**번 방법처럼 멤버 데이터에 바로 접근하거나** (2)**번 방법처럼 구조체를 만든 다음 할당하는 방법 두 가지 모두 가능하다.**

```
//(1)번 방법
inoutPersonSeq[i].id = 2;
inoutPersonSeq[i].name = CORBA::string_dup("name2");
//(2) 번
person->id = 2;
person->name = CORBA::string_dup("name2");
inPersonSeq[i] = person;
```

아래 실제 소스코드를 적어 놓았으니 참고하기 바란다.

```
//in, inout, out, return 변수 선언
Sample::PersonSeq_var inoutPersonSeq, outPersonSeq, retPersonSeq;
//in 값 할당 : (1),(2) 번 모두 가능하고, inout 의 경우도 두가지 모두 사용가능하다.
Sample::Person_var person = new Sample::Person();
CORBA::ULong i;
/* //(1)번 방법
Sample::PersonSeq inPersonSeq;
inPersonSeq.length(2);
for(i =0; i < inPersonSeq.length(); i++)
{
```

```
		inPersonSeq[i].id = 1;
		inPersonSeq[i].name = CORBA::string_dup("name1");
}*/
//(2)번 방법
Sample::PersonSeq_var inPersonSeq = new Sample::PersonSeq(2);
inPersonSeq->length(2);
for(i =0; i < inPersonSeq->length(); i++)
{
		inPersonSeq[i].id = 1;
		inPersonSeq[i].name = CORBA::string_dup("name1");
}
//inout 값 할당
inoutPersonSeq = new Sample::PersonSeq(2);
inoutPersonSeq->length(2);
for(i =0; i < inoutPersonSeq->length(); i++)
{
		//(1),(2) 번 모두 사용가능
/*		//(1)번
		inoutPersonSeq[i].id = 2;
		inoutPersonSeq[i].name = CORBA::string_dup("name2"); */
		//(2) 번
		person->id = 2;
		person->name = CORBA::string_dup("name2");
		inPersonSeq[i] = person;
}
//메쏘드 호출
retPersonSeq =  manager->oper1(inPersonSeq, inoutPersonSeq, outPersonSeq);
//inout, out 값 출력

… 중간생략 …

//return 값 출력
for(i =0; i < retPersonSeq->length(); i++)
{
		cout << "return 출력(retPersonSeq)-id: " << retPersonSeq[i].id
			<< ", name:" << retPersonSeq[i].name<< endl;
```

```
}
```

이상으로 간략하나마 첫번째 연재를 마치겠다. 지면의 한계로 많은 내용을 설명하지는 못했다. 연재 중에 설명하지 못하고 빠진 부분은 코바 사랑 사이트(http://www.object.pe.kr/)**를 방문하면 내용과 소스 코드가 게재 되어 있다. 아무쪼록 코바에 관심이 많은 독자들에게 여섯번의 연재가 많은 도움이 되었으면 하는 바람이다.**