論文 95-2-2-11

클라이언트/서버 모델의 분산 프로그래밍을 위한 C++클래스 라이브러리의 설계 및 구현

正會員 趙 光 濟*, 崔 英 根*

A design and implementation of C++ class library for distributed programming in client/server model

Kwang Je Cho*, Young Keun Choi* Regular Members

要 約

기존의 클라이언트/서버 모델의 분산 프로그램을 개발하는데 많이 이용되었던 RPC는 C 또는 Pascal과 같은 명령형 언어의 라이브러리로 제공되므로 객체 지향적 프로그래밍에 제약을 받는다. 따라서 본 논문에서는 분산 환경하에서 실행하는 클라이언트/서버 모델의 응용 프로그램을 개발하는데 필요한 개발 환경으로서 객체 지향 기법을 적용한 C++언어의 클래스 라이브러리인 ROCL(Remote Object Class Library)을 설계, 개발하였다.

ROCL은 클라이언트/서버의 통신 모듈인 스터브 부분과 서비스 관련 부분들을 객체의 멤버에 포함하므로써 C++ 언어를 사용한 객체 지향 프로그래밍에 적합한 개발 환경을 제공한다. 또한 동기화, 다중 처리 서버, 비동기화, 비동기 다중 처리 서버와 같은 클라이언트/서버의 세부 모델들을 ROCL의 기반 클래스로 제공하며, 응용 프로그램에서 적용하고자 하는 모델에 따라 이들을 계승하여 사용하도록 한다.

ABSTRACT

Conventional RPC has widely been used in the development of distributed program in the client/server model. It has some limitations in the object-oriented programming environment. This paper presents, however, design and implementation of ROCL(Remote Object Class Library), a class library that supports distributed programming in the client/server model.

ROCL provides a suitable environment for the object-oriented programming using the C++ language. An object contains a stub and services for client/server applications with a programmer can design objects from other parts of application. ROCL supports four kind of client/server models: a synchronous, a multi-tasking server, a asynchronous and a asynchronous multi-tasking server. As corresponding to each models, a base class is inherited from a derived class in the application

* 광운대학교 전자계산학과
Dept. of Computer Science, Kwangwoon University
論文番號 : 94256
接受日字 : 1994年 9月 27日

Ⅰ. 서 론

오늘날 대형 기종에서 처리하던 데이타 베이스 관리, 트랜젝션 처리, 이미지 처리와 같은 고속의 연산이 요구되는 작업들이 하드웨어 기술의 발전과 네트워크의 고속화로 분산처리가 가능하게 되었다.(15) 이들중 클라이언트/서버 모델은 분산처리의 일반화된 모델로서, 서비스를 제공하는 원격 서버와 서비스를 요구하는 클라이언트로 구성된다.(6,10,12) 클라이언트/서버 모델은 이미 X-Window System, NFS(Network Information Service) 그리고 상용화된 데이타베이스 관리 시스템인 Oracle 7, Sybase, Microsoft ODBC 등 다양한 분야에서 사용되고 있다.(6,11)

클라이언트/서버 모델의 분산 프로그램을 개발하기 위한 프로그래밍 환경으로서 현재까지 ONC(Open Network Computing), NCS(Network Computing System)와 OSF/1(Open System Foundation)등에서 개발한 RPC(Remote Procedure Calls) 시스템들이 많이 이용되어왔으며 각각 응용 프로그램을 개발하는데 필요한 독자적인 API(Application Programing Interface)를 제공하고 있다.(5,6,10,13) 그러나 이러한 API들은 C 또는 Pascal과 같은 명령형 언어(imperative language)의 라이브러리로 제공되므로 서버와 클라이언트의 관계가 서비스 함수의 호출과 결과의 반환이라는 단순한 형태에 머무를 수 밖에 없다.

객체 지향 기법은 클라이언트 측면에서 특정 서버에게 실제로 서비스로 요구하는 클라이언트내의 서비스 진입 함수들을 객체의 멤버 함수들로 정의하여 클라이언트내의 서비스 요구들이 하나의 객체를 통해 이루어지도록 할 수 있다. 서버의 관점에서도 서비스 기능을 수행하는데 필요한 서비스 함수들과 관련 데이타들을 서버에 정의된 객체의 멤버들로 포함하므로써 이들을 서버내의 다른 함수 및 데이타들로 구분할 수 있도록 한다. 즉 서버의 서비스는 기능적 측면만이 아닌 서비스 제공에 필요한 데이타들과 이들을 조작하는 함수들을 합친 객체로 볼수 있다.(1,2,4,7) 따라서 서버와 클라이언트의 관계는 서버 프로그램에 정의된 서비스 제공 객체와 클라이언트 프로그램에 정의된 서비스 요구 객체간의 관계로 재정의될 수 있다. 기존의 RPC들에서 제공하는 API는 서비스 제공 객체와 서비스 요구 객체간의 모델을 지원하지 않으므로 서버와 클라이언트의 관계를 객체간의 관계로 설계하고 구현하기 위해서 객체 지향 기법을 이용한 API를 개발할 필요과 있다.

본 논문에서는 클라이언트/서버 모델을 기반으로한 분산 프로그램을 C++언어로 작성하는 개발자들에게 C++언어의 객체 지향적 특성을 유지할 수 있도록 하고 클라이언트/서버의 서비스 제공과 서비스 호출을 객체라는 관점에서 프로그래밍할 수 있도록 C++언의 클래스 라이브러리인 ROCL(Remote Object Class Library)을 설계, 구현하였다. 제안된 ROCL의 사용상 특성은 서비스 제공과 서비스 요구간의 관계를 객체간의 관계로 표현할 수 있고, 프로그램 개발자에게 응용 프로그램의 서비스 초기화 및 서비스 호출방법과 같은 RPC의 프로그래밍 절차를 자세히 알아야 하는 부담을 줄여주며, 프로그래머가 직접 코딩해야 하는 많은 부분을 클래스 라이브러리로 대체하여 단순화 하므로써 코딩 오류를 미리 방지할 수 있다. 그리고 RPC에 관련된 많은 코드 부분이 ROCL에 의해 감추어지므로써 간략하고 읽기 쉬운 코드를 작성할 수 있다. 그리고 클래스 단위의 코드작성을 통한 모듈화(modulization), 계승(inheritance)에 의한 코드의 재사용성(reusability), 객체내의 서비스 멤버 함수명의 중복정의(function overloading)등 C++의 잇점을 살린 프로그래밍이 가능하다.(2,4,7)

또한, 클라이언트/서버 모델은 동기화, 다중 처리 서버, 비동기화, 비동기 다중 처리 서버와 같은 네가지의 세부적인 모델로 구분해볼 수 있는데 본 본문에서 제안된 ROCL은 이러한 각 모델에 대응되는 기반 클래스를 제공한다. ROCL의 기반 클래스들은 각 모델에 해당하는 서비스 제공 객체의 기반 클래스들과 서비스 요구 객체의 기반 클래스들로 나누어 설계되어 있다. 각각의 기반 클래스들은 클라어언트/서버 모델의 전형적인 예로서, 응용 프로그래머는 프로그램의 목적에 적합한 기반 클래스를 선택, 계승받으므로써 서비스 제공 객체와 서비스 요구 객체의 클래스를 정의할 수 있다. 서비스 제공 객체는 서비스 요구 객체와의 연결을 통해 클라이언트로부터의 서비스 요구에 응답하도록 하므로써 이들 객체간의 관계는 서로 대칭되는 형태를 갖게된다.

본 논문에서 설계, 구현한 ROCL은 ONC의 RPC 시스템을 구현 도구로 사용하였다. 그러나 ROCL의

설계시 제안되 사항들은 다른 RPC 시스템들을 사용한 ROCL의 재구성시에도 동일하게 적용될 수 있을 것이다.(13,14)

본 논문은 클라이언트/서버 모델의 분산 프로그램을 지원학기 위한 C++언어의 클래스 라이브러리의 설계, 구현을 목적으로 하고 있으며, 다음과 같이 구성되어 있다. 다음 장은 전반적인 클라이언트/서버 모델에 논의하며, 제 Ⅲ장은 ROCL의 설계 및 이를 이용한 구현 방법에 대해 알아보고, 제 Ⅳ장과 Ⅴ장은 실제 구현예와 ROCL을 통한 프로그램의 잇점을 분석하고, 제 Ⅵ장에 결론을 맺는 것으로 구성되었다.

Ⅱ. 클라이언트/서버 모델

분산 처리의 일반화된 모델인 클라이언트/서버 모델은 처리될 작업의 진입점을 포함하는 함수들의 집합과 실제 작업을 수행하는 또다른 함수들의 집합으로 구성된 컴퓨팅 환경이다. 예를들면 X-Window 시스템은 분산 환경에서 클라이언트/서버 모델을 제공하는 전형적인 예이며 RPC시스템은 클라이언트/서버 모델을 기반으로하는 응용 프로그램을 개발하는데 있어서 적절한 개발 환경으로 받아들여지고 있다. 클라이언트와 서버는 각각 독립적인 프로그램으로서 네트워크상의 어느 처리기든지 존재할 수 있으며 반드시 분리된 처리기상에 존재할 필요는 없다.(4,6,8,10)

본 논문에서 설계, 구현한 ROCL은 ONC의 RPC 시스템을 기반으로 하여 작성되었으며 이 시스템으로 개발된 기본적인 클라이언트/서버 응용 프로그램은 그림 2.1과 같이 실제로 함수를 실행하는 단일 서버 프로그램과 이것과 통신하는 달일 클라이언트 프로그램으로 구성된다.(6,13) 그림 2.1에서와 같이 ONC의 RPC로 작성한 클라이언트측 프로그램은 원격 서버 프로그램에게 인자를 전달하고 그 결과를 전달 받으며, 클라이언트와 서버의 서비스 함수 호출과 반환이 동기화된 형태로 발생하기 때문에 단일 프로그램 내에서의 함수 호출 및 반환과 유사하다.

ONC의 RPC 시스템은 다음과 같이 구성되어 있다. 첫째로 통신상의 메시지를 어떻게 구성하고 교환하는지에 대한 실제적인 구현부분인 스터브(stub)를 구축하는 API부분과 RPC 시스템들마다 독자적으로 지원하는 프로토콜 정의 언어(protocol definition

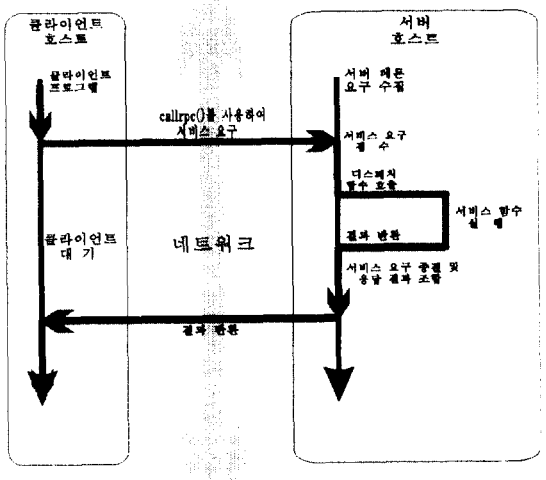


그림 2.1 RPC에 의한 기본적인 클라이언트/서버의 서비스 호출
Fig. 2.1 Service call for basic client/serve using RPC

language)를 클라이언트와 서버의 스터브 코드 및 서비스의 인자와 결과값의 전달에 사용되는 사용자 정의형의 데이타들을 위한 XDR 함수들을 자동 생성하는 프로토콜 컴파일러, 그리고 외부 시스템과의 메시지 교환시에 전송 메시지를 비트 스트림으로 변환하는 XDR(External Data Representation)부분으로 되어 있다.

RPC 시스템이 제공하는 프로토콜 컴파일러는 스터브 부분의 코드를 자동 생성하므로써 프로그램의 복잡성을 줄여주는 반면 기본적인 형태의 프로그램만을 고려한 코드가 생성되므로 응용 프로그램의 목적에 따라 프로그래머가 생성된 코드의 일부를 수정해야하는 번거로움이 있다. 그러나 XDR의 경우에는 응용 프로그램의 목적에 관계없이 사용이 가능하므로 ROCL에서는 프로토콜 컴파일러에 의해 생성된 XDR 부분은 그대로 사용하도록 하였다.

일반적인 클라이언트/서버 모델은 응용프로그램의 목적에 따라 좀더 세분화된 구분을 필요로하며 다음과 같이 네가지의 세부 모델로 구분될 수 있다.(6,10,12,13)

(1) 기존의 RPC 시스템엣서 제공하는 프로토콜 컴파일러인 RPCGEN에 의해 생성되는 가장 기본적인 형태인 서비스 호출과 결과의 반환이 동기화를 이루는 모델

(2) 여러 클라이언트로부터의 요구에 응답하기 위해서 서버 프로그램이 다중 처리가 가능하도록 작성된 다중 처리 서버 모델

(3) 클라이언트측의 서비스 요구와 서버에서의 결과 반환이 비동기적으로 이루어지는 형태로서 대화식의 프로그램이나 항공기의 예약 스케줄관리, 증권 거래 관리와 같은 응용 프로그램들에서 유용한 비동기화 모델

(4) 비동기화 모델과 다중 처리 서버가 합쳐진 비동기 다중 처리 서버 모델

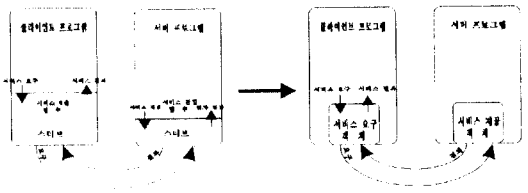
이러한 클라이언트/서버의 네가지 세부 모델들은 앞으로 설명할 ROCL의 클래스 설계 및 구현에 있어서 기본적인 모델이 되고 있다.

Ⅲ. ROCL의 설계 및 클라이언트/서버의 구현

3.1 ROCL의 설계

기존의 RPC로 개발된 클라이언트/서버 프로그램들은 비객체 지향적인 프로그램들로서 스터브 부분의 초기화 및 서비스 관련 부분들이 응용 프로그램의 다른 부분들과 섞여서 작성되기 쉬우며 일반 함수의 호출과 서비스 함수 호출간의 구분이 명확하지 않기 때문에 오류 발생시 이에 대한 디버깅이 어렵게 되고 서비스 관련 부분만의 독립적 설계를 어렵게 한다. 그러나 본 논문에서 설계한 ROCL은 스터브와 서비스 관련 부분을 객체내에 포함하므로써 이 객체를 통해 서비스가 이루어지도록 한다. 그러므로써 서비스 부분과 다른 부분과의 관계가 분명해지고 프로그램의 관리 및 유지, 보수가 쉬워지며 나중에 이 객체들을 다시 사용할 수 있으므로 재사용성을 높일 수 있다.

그림 3.1은 기존의 RPC로 작성한 클라이언트/서버 프로그램 모델을 ROCL을 이요한 프로그램으로 전환된 형태를 나타낸다. 그림 3.1 ii)에서 ROCL로 작성한 프로그램의 경우 서비스 요구객체와 서비스 제공 객체의 역할은 그림 3.1 i의 클라이언트/서버 내에서 서비스 요구 및 제공을 통합된 형태로 수행하게 된다. 또한 그림 3.1에서 알수 있듯이 기존 RPC 프로그램은 서비스 요구와 제공이 각각의 스터브를 통해 이루어지지만 ROCL의 경우에는 서비스 요구 객체와 서비스 제공 객체를 통해 이루어진다. 따라서 응용 프로그램의 개발시에 이들 객체들만의 독립된 설계가 가능하다.



i) 일반적인 서버/클라이언트 프로그램의 관계와 스터브의 역할　　ii) ROCL을 사용하여 정의한 서버/클라이언트의 객체물과 그 역할

그림 3.1 클라이언트/서버의 스터브를 서비스 요구 객체와 서비스 제공 객체로 전환

Fig. 3.1 Transform stubs for client/server to service-request-object and service and service-provide-object

다음의 그림 3.2는 ROCL을 이용한 응용 프로그램에 대한 기반 클래스와 파생 클래스의 구조 및 이들간의 관계를 나타낸 것이다. 그림 3.2에서의 각 파생 클래스는 대응되는 기반 클래스를 계승받으므로써 RPC의 스터브 부분처럼 원격 호출과 관련된 부분을 포함하게 된다. 따라서 이와 같은 구조의 ROCL은 디스패치 함수가 서버측 기반 클래스의 멤버 함수로서 제공되므로 기존의 RPC 서버 프로그램과 달리 응용 프로그램에서 별도의 디스패치 함수를 작성할 필요가 없게 된다. 다만 ROCL은 기반 클래스내의 디스패치 함수가 클라이언트의 요구에 맞는 적절한 서비스 함수와 XDR 함수를 선택할 수 있도록 하는 별도의 검색 기능을 필요로 한다. 이러한 검색은 실행중에 발생하므로 서비스 함수와 XDR 함수를 서비스 제공 객체의 생성시에 미리 서비스 데이타 베이스와 XDR 대이타 베이스에 등록한 후 디스패치 멤버 함수가 검색하도록 하므로써 가능하다.

ROCL을 이용한 클라이언트 프로그램에서 서비스 진입 함수는 서비스 요구 객체의 멤버 함수로서 정의된다. 서비스 요구 객체는 서비스 호출 함수라는 기반 클래스의 맴버 함수를 상속받는다. 서비스 호출 함수는 서비스 진입 함수에 의해 호출되어 RPC의 clnt_call()함수를 호출한다. 이 과정에서 서비스 호출 함수는 clnt_call()의 인자로 사용될 XDR 함수를 필요로 한다. XDR 함수는 서비스 함수의 인자값 또는 결과값을 전달받거나 전달할 때 사용된다.

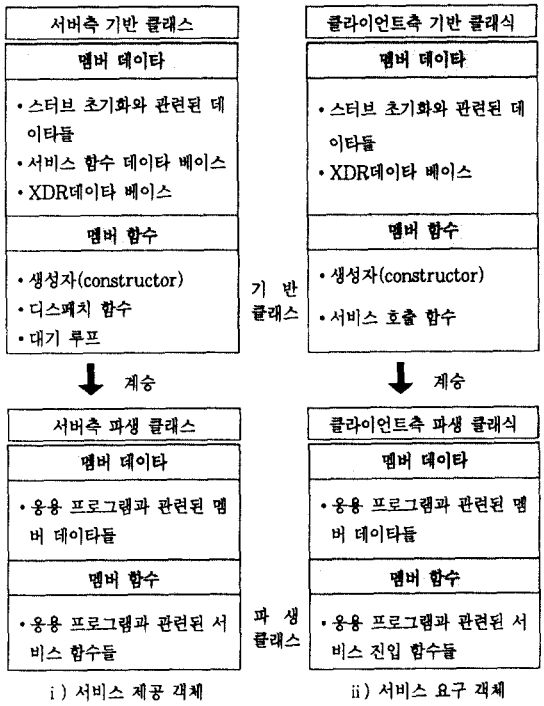


그림 3.2 클래스 계승과 객체들의 구성
Fig. 3.2 Class inheritance and object construction

그림 3.3은 XDR 데이타 베이스의 기능을 보여주고 있다. 여기서 XDR 데이타 베이스는 그림 3.3의 서비스 제공 객체일 경우에 디스패치 함수에서 해당 서비스 함수와 함께 XDR 데이타 베이스에 등록된 XDR 함수를 검색이 가능하도록 지원하고, 서비스 요구 객체일 경우에도 서비스 호출 함수를 통해 등록된 XDR 함수가 검색되도록 한다.

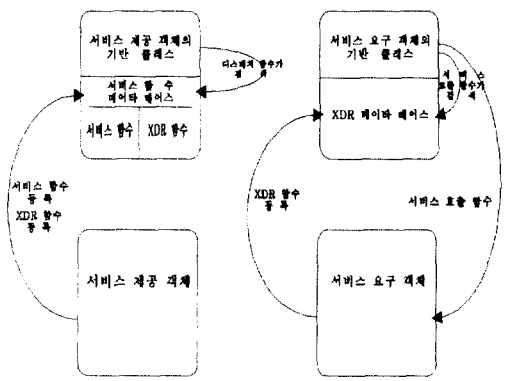


그림 3.3 객체내의 데이타 베이스와 그 기능
Fig. 3.3 Database operation in object

ROCL에서 사용한 데이타 베이스를 이용한 서비스 함수 및 XDR 함수의 등록, 검색 기법은 기반 클래스의 구현 부분에 대한 수정을 하지 않도록 하므로써 기존 RPC의 경우처럼 서비스 함수의 추가나 삭제시에 스터브 부분인 ROCL의 기반 클래스의 구현 부분에 대한 재컴파일을 필요로 하지 않는다.

ROCL은 동기화 모델뿐만 아니라 클라이언트/서버의 세부모델인 다중 처리 서버 모델, 비동기화 모델 및 비동기화 다중 처리 서버 모델의 각 클래스들을 포함하고 있는데 각 클래스는 그 특성에 맞게 내부 멤버 함수들이 별도로 작성되었지만 응용프로그래머는 이에 대한 이해가 필요치 않으며 단지 자신이 어느 클래스의 특성을 따를것인가만 결정하면 된다.

3.2 ROCL의 계승관계와 파생클래스

ROCL의 기반 클래스들은 동기화, 다중 처리 서버 비동기화 및 비동기 다중 처리 서버와 같은 클라이언트/서버의 세부 모델들을 기준으로 설계되었는데 각 모델에 해당하는 클래스들은 다음과 같다. 먼저 동기화 모델에 해당하는 기반 클래스들로는 서버측의 serverStub 클래스와 클라이언트측의 clientStub 클래가 있고, 다중 처리 서버 모델의 기반 클래스들로는 서버측에 multiServerStub 클래스와 클라이언트측의 multiClientStub클래스가 있다. asyncClientStub 클래스, 비동기 다중 처리 서버 모델에 있어서는 서버측에 asyncMultiServerStub 클래스와 클라이언트측에 asnyClientStub 클래스가 각각 해당 모델의 기반 클래스이다.

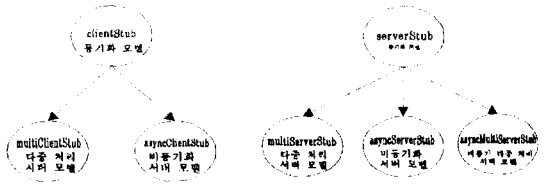
앞의 기반 클래스들중 asnyClientStub 클래스는 비동기화 모델과 비동기 다중 처리 서버 모델의 클라이언트측 기반 클래스로 중복 사용된다. 그 이유는 두 가지 모델에 있어서 클라이언트측 객체의 역할이 동일하기 때문이다. 표3.2는 이와 같은 모델과 클래스간의 관계를 나타낸다.

그림 3.4는 ROCL의 기반 클래스들간의 계승관계를 보여주고 있다. 이들 서버 기반 클래스와 클라이언트 기반 클래스들은 응용 프로그램에서 일대일로 상호작용을 하는데 예를 들면 응용 프로그램의 서버와 클라이언트가 동기화 모델인 경우 serverStub 클래스와 clientStub 클래스가 각각 서비스 제공 객체

와 서비스 요구 객체의 기반 클래스가 된다. 실제 응용 프로그램내에서 서비스를 제공, 요구하는 객체들은 프로그램의 성격과 목적에 따라 필요한 기반 클래스를 계승해야한다. 또한 새로운 모델의 기반 클래스를 설계하고자 할 경우 반드시 최상위 기반 클래스인 serverStub와 clientStub 또는 이들 클래스로부터 파생된 클래스를 계승받아야 한다.

표 3.2 모델별 ROCL의 클래스 정의
Table. 3.2 Class definition of ROCL for each model

모델	구분	클래스
동기화모델	클라이언트측	clientStub(최상위 기반 클래스)
	서버측	serverStub(최상위 기반 클래스)
다중 처리 서버 모델	클라이언트측	multiclientStub
	서버측	multiServeStub
비동기화 모델	클라이언트측	asyncClinetStub
	서버측	asyncServerStub
비동기 다중 처리 서버 모델	클라이언트측	asyncClientStub
	서버측	asyncMultiServerStub



i) 클라이언트측 클래스들의 계승관계　ii) 서버측 클래스들의 계승관계

그림 3.4 ROCL의 클래스 계승 관계
Fig. 3.4 Class inheritance relation of ROCL

ROCL의 각 기반 클래스들은 다음과 같은 멤버 데이타들과 멤버 함수들을 포함하고 있는데 이들은 응용 프로그램의 파생 클래스에서 스터브를 구축하기 위한 기본적인 기능들을 제공한다.

(1) 동기화 모델들의 클라이언트측 기반 클래스

동기화 모델이 클라이언트/서버 프로그램은 가장 일반적인 형태의 모델로서 그림 2.1과 같이 서비스의 호출과 반환이 일반 함수의 호출과 같이 동기화되어 발생한다. 따라서 서버는 반드시 현재의 클라이언트 요구에 대한 서비스를 끝내야만 다른 요구에 응답할 수 있다.

이 기반 클래스는 중요 멤버들로서 다음과 같은 데이타들과 함수들을 포함한다. 먼저 멤버 데이타인 cl은 서비스 요구 객체의 생성시에 서버 프로그램에 대한 정보들을 가지고 초기화 된다. 그리고 XDR 함수들을 등록하는데 필요한 데이타 베이스로 사용되는 xdrFuncs는 서비스 제공 객체의 서비스 함수의 인자형과 결과형에 해당하는 XDR 함수들을 원소로 하며 서비스 요구 객체의 생성시에 XDR 함수들로 초기화 된다. 등록된 XDR 함수들은 서비스 진입 함수에 의해 호출된 서비스 호출 함수인 callService( ) 멤버 함수에 의해 검색되어 사용된다.

그리고 멤버 함수들로 객체의 생성자와 XDR 함수를 데이타 베이스에 등록하는 addXdr( ) 함수, 서비스 함수를 호출하는 callService( ) 함수와 메시지의 전송에 필요한 대기 시간을 설정하는 changeTimeout( ) 함수 등이 있다. clientStub 클래스는 ROCL에서 제공하는 네가지 모델들 중 동기화 모델에 해당하는데 다른 모델의 클라이언트측 클래스들의 기반 클래스가 된다. 동기화 모델의 클라이언트측 기반 클래스는 다음과 같이 정의되어 있다.

```
// 동기화 모델의 클라이언트측 기반 클래스
// 다른 클라이언트측 기반 클래스들의 최상위 기반 클래스.
class clientStub
{
protected:
  CLIENT *cl;
  argAndResult xdrFuncs[MAX_XDR];
  struct timeval timeout;

  clientStub(char *, u_long, u_long);
  clinetStub( ) { };
  void setTimeoutValue(u_int, u_int);

  public:
  //CLIENT 구조체를 생성하는 멤버 함수.
  int createClient(char *, u_long, u_long);
  void zddXdr(u_long, xdrproc_t, xdrproc_t);
  void callService(u_long, char *, char *);
  void changeTimeout(u_long));
  // 에러 발생시 메시지 출력 함수
  void clientProcessError(char *);
};
```

클라이언트 프로그램의 초기화 과정은 다음과 같다. 앞에서 정의한 기반 클래스를 계승한 파생 클래스는 클라이언트 프로그램의 실행시에 서비스 요구 객체를 생성하는데 이 과정에서 기반 클래스의 객체 생성자는 상대 서버의 호스트 이름, 프로그램 번호, 버전등의 정보를 가지고 트랜스포트 핸들러를 생성한 후 addXdr( ) 함수를 사용하여 서비스 함수와의 메시지 교환에 필요한 XDR 함수들을 xdrFuncs에 등록한다. 초기화가 끝나게 되면 서버와 연결이 이루어진 상태이므로 클라이언트 프로그램은 서비스 요구

객체내의 멤버 함수들인 서비스 진입 함수들을 통해 서버에게 서비스를 요구할 수 있게 된다.

(2) 동기화 모델의 서버측 기반 클래스

동기화 모델의 서버측 기반 클래스는 중요 멤버 데이타로서 service_t 형의 배열인 SVCS와 programNo, versionNo 등을 포함하고 있다. SVCS는 서비스 요구 객체의 멤버 함수들인 서비스 함수들과 관련 XDR 함수에 대한 정보들을 저장하기 위한 데이타 베이스 역할을 한다. programNo과 versionNo은 protmapper에 등록될 서버 프로그램의 번호와 버전값을 유지한다.

이 클래스에서 정의된 중요 멤버 함수들은 서비스 함수를 호출하는 dispatcher( ) 함수, 서버 프로그램의 초기화를 수행하는 initiateServer( ) 함수, 서비스 함수와 XDR 함수들을 데이타 베이스에 등록하는 addService( ) 및 이를 검색하여 반환하는 getService( ) 그리고 프로토콜에 따라 트랜스포트 핸들러를 생성하는 createTCP( ), createUDP( ) 함수와 서버 프로그램의 대기 루프를 구성하는 runService( ) 함수 등으로 되어 있다. 동기화 모델의 서버측 기반 클래스는 다음과 같이 정의되어 있다.

```
// 동기화 모델의 클라이언트측 기반 클래스
// 다른 서버측 기반 클래스들의 최상위 기반 클래스.
class serviceStub
{
private:
  static service_t SVCS[MAX_SERVICE];

  static void dispatcher(struct svc_req *, SVCXPRT *);
  void initiaeServer( );

protected:
  static int registered;
  static u_long programNo;
  static u_long versionNo;
  static struct timeval tval;

  serverStub(u_long, u_long);
  serverSuub( ) { };
  static service_t *getService(u_long);
  void createUDP(void (*)(struct svc_req *, SVCXPRT *));
  void cerateTCP(void (*)(struct svc_req *, SVCXPRT *));
  void setTimeValue(u_int, u_int);

public:
  void addService(serverStub*, u_long, size_t, xdrproc_t, sdrproc_t, stat_svc_t));
  void runService( );
};
```

서버 프로그램은 실행 초기에 앞에서 정의한 기반 클래스로부터 계승받은 파생 클래스를 사용하여 서비스 제공 객체를 생성하는데 기반 클래스의 생성자는 서비스 제공 객체를 초기화하는 과정에서 initiateServer( ), addService( ) 멤버 함수들을 호출한다. initiateServer( ) 함수는 서버 프로그램의 정보를 protmapper에 등록하므로써 서버 프로그램을 초기화한다. 이 과정에서 dispatcher( ) 멤버 함수가 protmapper에 등록되는데 이 멤버 함수는 클라이언트의 요구에 따라 클라이언트로부터 주어진 인자값들을 실인자로 하는 서비스 멤버 함수와 XDR 함수들을 서비스 데이타 베이스에서 찾아서 실행한 후 그 결과를 되돌려 주는 역할을 수행한다. addService 함수는 서비스 데이타 베이스에 서비스 함수와 이 함수의 인자 전달과 결과 반환시에 필요로 하는 XDR 함수들을 등록한다. 초기화가 끝난 후에 서버 프로그램은 서비스 요구를 기다리게 되는데 이 때 runService( ) 멤버 함수를 호출하고 서버 프로그램은 대기상태로 들어간다.

(3) 다중 처리 서버 모델의 클라이언트측 기반 클래스

다중 처리 서버 모델의 프로그램은 그림 3.5와 같이 여러개의 클라이언트 프로그램으로부터 동시에 발생하는 서비스 요구에 응답하기 위해서 서버 프로그램이 자 프로세스를 생성하도록한다. 즉 부 프로세스는 클라이언트의 요구가 발생하는 즉시 자 프로세스를 생성한 후 다른 클라이언트의 요구를 검출하기 위해 대기 상태에 들어가게 된다.

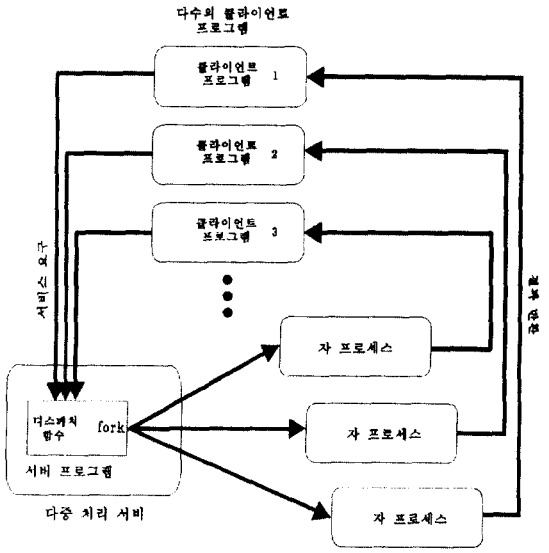


그림 3.5 다중 처리 서버 모델의 서비스 호출
Fig. 3.5 Service call for multi-tasking server model

다중 처리 서버 모델의 경우 클라이언트측 기반 클래스는 clientStub 클래스를 계승받으므로 서비스 요구 객체의 초기화 과정은 앞에서 설명한 동기화 모델의 경우와 동일하다.

(4) 다중 처리 서버 모델의 서버측 기반 클래스

multiServerStub 클래스는 serverStub 클래스로부터 계승되는데 serverStub 클래스는 서버 프로그램의 다중 처리를 고려하지 않기 때문에 이것이 가능하도록 dispatcher( ) 함수를 새로 정의해야 한다. 따라서 multiServerStub 클래스의 멤버 함수인 dispacher( )는 클라이언트로부터의 요구가 발생하면 이를 처리하는 서버 프로그램의 자 프로세스를 생성하고 dispatcher( ) 함수를 끝내도록 작성되어 있다.

다중 처리 서버 모델에 있어서 서비스 제공 객체의 초기화 과정 역시 동기하 모델의 경우와 동일하다.

(5) 비동기화 모델의 클라이언트측 기반 클래스

비동기화 모델에세 서버 프로그램은 클라이언트 프로그램으로부터의 요구가 발생하는 즉시 무의미한 값을 클라이언트 프로그램에게 반환하여 클라이언트 프로그램의 대기상태에 놓이지 않도록 한다. 그 다음 서버 프로그램은 클라이언트와의 연결을 끊은 상태에서 서비스 함수를 실행하고 그 결과는 클라이언트 프로그램내에 정의된 수신 함수를 호출하여 반환하므로써 비동기를 이루도록 한다. 그림 3.6에서 서버 프로그램이 클라이언트 프로그램에게 수신 함수를 호출할 때에는 클라이언트 프로그램이 서버와 같은 역할을 수행하게 된다. 즉, 서버 프로그램과 클라이언트 프로그램이 각각 서버와 클라이언트의 기능을 함께 포함하도록 해야 한다.

비동기화 모델에 있어서 클라이언트측 기반 클래스의 가장 큰 특징은 클라이언트 프로그램이 서버 프로그램과 동일한 기능을 수행하도록 한다는데 있다. 즉 비동기화 모델에 있어서 클라이언트 프로그램은 서버 프로그램에게 서비스를 요구하고 그 결과를 기다리지 않고 즉시 다음 코드를 실행하기 때문에 서버로부터의 결과를 수신할 수 있는 수신 함수가 별도로 필요하게 된다. 이 수신 함수는 서버 프로그램이 실행한 서비스의 결과를 클라이언트에게 반환하도록 하는 역할을 수행한다. 즉 서버 프로그램이 실행한 후 그 결과값을 클라이언트에게 반환하고자 할 때 클라이언트 프로그램내의 수신 함수를 호출하는데 서비스의 결과값을 인자로하여 수신 함수를 호출하게 된다. 따라서 서비스 요구 객체내에 있는 임의의 서비스 진입 함수는 자신이 호출한 서비스의 결과값을 수신하는 하나의 수신 함수와 쌍을 이루도록 해야 한다.

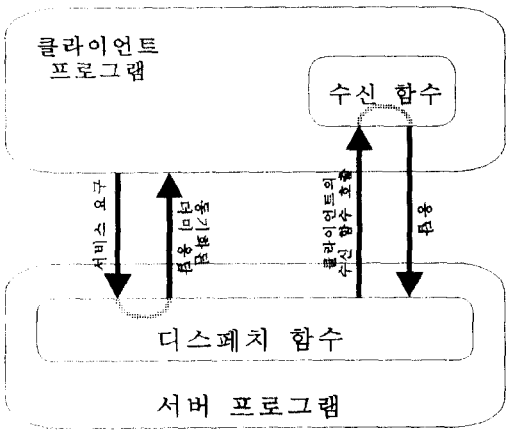


그림 3.6 비동기화 모델의 서비스 호출
Fig. 3.6 Service call for asynchronous model

```
// 비동기화 모델 및 비동기화 다중 처리 서버 모델의 클라이언트측 기반 클래스.
// clientStub로 부터 계승받음.
class asyncClientStub:public clientStub
{
private:
  staic int registered;
  static u_long localPNo;
  static u_long localVNo;
  int busy;
  char *serverName
  u_long serverPNo
  u_long serverVNo
  int ts;

protected:
  asyncClientStub(u_long, u_long, char *, u_long, u_long)
  asyncClinetStub( ) {};

public:
  void regiserReciever(u_long, xdrproc_t,stat_svc_t);
  void callService(u_long, char *, char *);
};
```

asyncClientStub 클래스를 통한 서비스 요구 객체의 초기화 과정은 앞에서 설명한 동기화나 다중 처리 서버의 경우와 약간의 차이를 갖는다. XDR 함수를 등록하는것은 같지만 부가적으로 수신 함수를 등록하는 과정이 필요하다. 기반 클래스인 asyncClientStub의 멤버 함수 registerReciever( )는 수신 함수를 등록하는데 사용된다. 그리고 이를 계승받는 파생 클래스에는 서비스 진입 함수와 함께 결과를 수신하는 수신 함수를 포함해야 한다.

비동기화 모델은 호출과 결과의 반환이 동기를 이루지 못하므로 서비스 호출시 주의해야 한다. 즉 서비스를 호출했다고해서 결과가 그 다음에 곧바로 나타나는 것이 아니므로 결과의 반환이 언제 있을지 알 수 없다.

(6) 비동기화 모델의 서버측 기반 클래스

비동기화 모델의 서버측 클래스 역시 서버 프로그램이 서비스를 수행한 후 그 결과를 클라이언트 프로그램에게 반환하기 위해 클라이언트 프로그램의 수신 함수를 호출할 수 있도록 해야 한다. 따라서 서버측 클래스도 클라이언트측 클래스와 유사한 멤버들을 포함하고 있어야 한다

```
// 비동기화 모델의 서버측 기반 클래스.
// serverStub로부터 계승받음.
class asyncServerStub:public serverStub
{
private:
  staic void dispatcher(sttruct svc_req *, SVCXPRT *);
  void innitiiateServer( );

protected:
  static u_long clinetPNo;
  static u_long localVNo;
  static argAndResult xdrFuncs[MAX_XDR];

  asyncServerStub(u_long, u_long, u_long, u_long)
  asyncServerStub( ) { };

public:
  void addXdr(u_long, xdrproc_t, xdrproc_t);
};
```

비동기화 모델에 있어서 서비스 제공 객체의 초기화 과정은 다른 모델의 서버측 객체 초기화 과정과 유사하나 다른 모델의 경우처럼 서비스 함수만을 등록하는 것이 아니라 자체적으로 보유하고 있는 XDR 데이타 베이스에 클라이언트 프로그램으로부터 수신 함수를 호출할 때 필요한 XDR 함수들을 등록하는 차이점이 있다.

(7) 비동기화 다중 처리 서버 모델의 클라이언트측 기반 클래스

이 모델의 클라이언트 프로그램은 비동기화 모델의 경우와 동일한 역할을 수행한다. 즉 서버 프로그램의 다중 처리 여부에 관계없이 클라이언트 프로그램은 서비스를 요구할 수 있기 때문이다.

(8) 비동기화 다중 처리 서버 모델의 서버측 기반 클래스

비동기화 다중 처리 서버 모델은 앞에서 설명한 비동기화 모델을 서버 프로그램이 다중 처리 가능하도록 확장한 모델이다. 따라서 이 모델의 서버측 클래스는 비동기화 모델의 특성을 그래도 유지하면서 앞에서 설명한 다중 처리 서버 모델의 경우와 같이 dispatcher( ) 멤버 함수를 다중처리 작업에 적합하도록 수정하였다.

3.3 ROCL을 이용한 클라이언트/서버의 구현

서버 프로그램내의 파생 클래스들은 서비스를 실제로 수행하는 멤버 함수들을 포함한다. 따라서 이러한 서비스 함수들은 private, portected, public중 어디에도 있을 수 있으며 원격 호출이 아닌 서버 프로그램내의 호출(local call)인 경우에도 사용될 수 있다. 이러한 서버 프로그램내의 호출인 경우에는 C++의 접근허용 규칙(scope resolution)이 그대로 유지되지만 클라이언트 프로그램에 의한 원격 호출시에는 클라이언트 프로그램에서 서버 프로그램내의 서비스 함수들이 갖는 접근범위를 알 수 없게 된다.

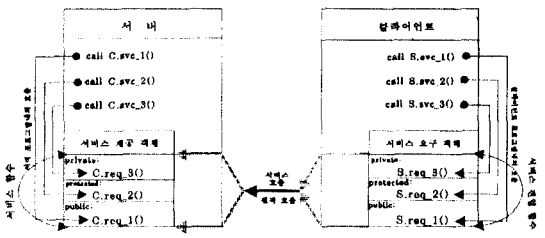


그림 3.7 응용 프로그램내의 멤버 함수 호출과 원격 서비스 호출
Fig. 3.7 Member function and remote service's calls in application program

따라서 본 논문에서는 앞에서와 같은 문제를 해결하기 위하여 그림 3.7에서와 같이 클라이언트 프로그램내의 파생 클래스가 서버측의 파생 클래스의 서비스 함수와 동일한 접근규칙을 갖는 서비스 진입 함수들을 포함하도록 하였다. 그림 3.8은 ROCL을 이용한 클라이언트/서버 모델의 프로그램을 개발하는 대략적인 흐름을 보여주고 있다.

그림 3.8에서 DCL_DYN_SERVICE와 DCL_TRANSLATOR는 서비스 함수와 XDR 함수를 각각의 데이타 베이스에 등록하도록 하는 매크로 함수이다. 전자는 DCL_DYN_SERVICE(1, int, int, &Rtrace Server::die_1)와 같은 형태로 호출되는데

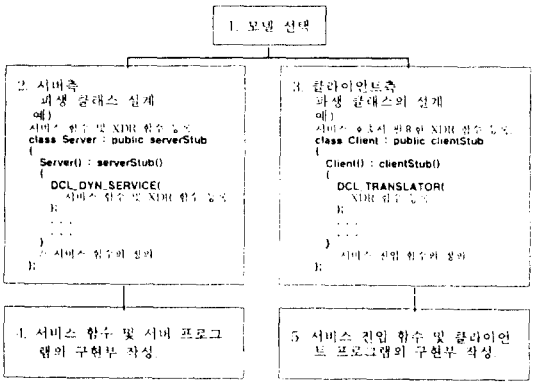


그림 3.8 ROCL을 이용한 프로그램 개발 절차
Fig. 3.8 Application program development procedure using ROCL

첫번째 인자는 서비스 함수의 인식 번호이며 두번째와 세번째는 서비스 함수의 인자값과 결과값에 해당하는 데이타 형으로서 해당 XDR 함수를 등록하게 된다. 마지막 인자는 서비스 제공 객체내의 멤버 함수인 서비스 함수를 등록하도록 한다. 그리고 후자의 경우 DCL_TRANSLATOR(1, int, int)와 같은 형태로 호출되는데 DCL_DYN_SERVICE의 앞에서부터 세개의 인자들과 동일한 의미를 갖는다.

Ⅳ. 응용 예

본 논문에서는 클라이언트/서버 모델의 특성을 잘 나타내는 전형적인 예로서 ROCL을 이용한 응용 프로그램인 RDIR(Remote DIR)과DIRT(Distributed Image Ray Tracer)를 들기로 한다.

RDIR은 ROCL이 제공하는 네가지 모델을 적용하기에 적절한 예로서 클라이언트 프로그램의 요구에 따라 서버 프로그램이 실행중인 원격 호스트의 디렉토리를 출력하도록 한다. RDIR은 ROCL의 세부 모델들을 모두 적용하여 각 모델의 특성에 따라 작성되었으며 각 모델에 대한 클래스 정의는 4.1절에서 소개한다.

또한 ROCL이 네가지 모델외에서도 적용될 수 있음을 보이기 위해 DIRT를 예로 들었다. DIRT는 독립된 호스트들에서 실행하는 다수의 서버 프로그램들과 통신하기 위해서 클라이언트 프로그램의 다중 처리가 요구된다.

4.1 RDIR(Remote Directory List)

RDIR은 ROCL에서 제공하는 동기화, 다중 처리 서버, 비동기화와 비동기 다중 처리 서버 모델에 따라 설계, 구현하였다. RDIR의 각 모델에 해당하는 클래스 정의는 다음과 같고 각 클래스로부터 생성되는 객체는 3.2절의 초기화 과정에 의해 초기화 된다.

(1) RDIR의 동기화 모델 클래스

다음의 클래스 정의에서 client_1 클래스는 RDIR의 동기화 모델을 정의하기 위한 클래스로서 clientStub 클래스를 기반으로 한다. 이 클래스는 서버의 호스트 이름 sv, 프로그램 번호 p와 버전 v를 인자로하는 생성자 client_1을 호출하여 서비스 요구 객체를 생성한다. 또한 서비스 진입 함수로서 readdir_res *readdir_1(nametype *)를 포함하고 있다. 이 함수는 서버 프로그램의 서비스 함수에게 디렉토리 이름을 전달하고 그 리스트를 반환받는 역할을 한다.

그리고 service_1 클래스는 serverStub 클래스를 기반으로하는 서버측 클래스이다. 생성자인 servece_1은 서버 프로그램의 번화 p, 버전 v를 받아들여 서비스 제공 객체를 생성한다. 그리고 서비스 함수로서 readdir_res *readdir_1(nametype *)를 포함하고 있다. 이 함수는 클라이언트의 요구에 해당하는 디렉토리 리스트를 구성하여 반환한다.

```
// ROCL의 클라이언트측 기반 클래스인 clientStub로부터 계승.
class client_1:public clientStub
{
public:
  // sv:서버 호스트 이름
  // p:서버의 프로그램 번호
  // v:서버의 버전
  client_1(char * sv, u_long p, u_long v)):clientStub(sv,p,v)
  {
  DCL_TRANSLATOR(1, nametype, readdir_res)
  }
  // 서비스 진입 함수
  readdir_res *readdiir_1(nametype *);
};

// ROCL의 서버측 기반 클래스인serverStub로부터 계승.
class service_1:public serverStub
{
public:
  // p:서버의 프로그램 번호
  // v:서버의 버전
  service_1(u_long p, u_long v)):serverStub(p,v)
  {
  DCL_DYN_SERVICE(1, nametype, readdir_res, &service_1::readdir_1)
  }
  // 서비스 함수
  readdir_res *readdiir_1(nametype *);
};
```

(2) RDIR의 다중 처리 서버 모델 클래스

이 모델은 다수의 클라이언트 요구에 응답하기 위해 서버 프로그램이 다중 처리되는 경우로서 클라이언트측과 서버측의 파생 클래스 들은 ROCL의 다중 처리 서버 모델의 기반 클래스인 multiClientStub와 multiServerStub로부터 계승받도록 한다.

(3) RDIR의 비동기화 모델 클래스

다음은 RDIR의 비동기화 모델에 해당하는 클래스 정의이다. 이 모델의 경우에 있어서 서비스의 호출과 결과의 반환이 비동기적으로 발생하므로 서버 프로그램은 서비스를 반환하기 위해 클라이언트 프로르램에서 정의된 서비스 수신 함수를 호출할 수 있도록 해야 한다. 이를 위해서 서비스 제공 객체는 3.2절의 (6)에서 설명한 XDR 함수의 등록 과정을 수행한다.

다음의 client_1과 service_1 클래스 정의에서 사용되는 DCL_ASYN_TRANSLATOR와 DCL_ASYN_DYN_SERVICE는 3.3절의 DCL_DYN_SERVICE와 DCL_TRANSLATOR와 유사한 역할을 수행하는 매크로 함수이다.

DCL_ASYN_TRANSLATOR는 마지막 인자에 서버로부터의 서비스 결과를 수신하는 수신 함수를 등록하도록 되어있다. 그리고 DCL_ASYN_DYN_SERVICE는 인자들의 형태가 DCL_DYN_SERVICE와 동일하지만 사용되는 의미에 차이가 있다. 즉, DCL_DYN_SERVICE의 경우에는 두번째와 세번째 인자가 서비스 함수의 인자와 결과에 대한 데이타 형만을 의미하지만 DCL_ASYN_DYN_SERVICE에서는 세번째 인자가 수신 함수를 호출할 때 사용되는 인자의 데이타 형도 의미한다.

```
// ROCL의 클라이언트측 기반 클래스인 asyncClientStub로부터 계승
class client_1:public asyncClientStub
{
public:
  // p:클라이언트의 프로그램 번호
  // v:클라이언트의 버전
  // sv:서버 호스트 이름
  // rP:서버의 프로그램 번호
  //rV:서버의 버전
  client_1(u_long p, u_long v, char * sv, u_long rP, u_long rV)
            :asyncClientStub(p,v,sv,rP,rV)
  {
    DCL_ASYN_TRANSLATOR(1, nametype, &client_1::recv_readdir_1);
  }
  // 서비스 진입 함수
  void readdir_1(nametype *);
  // 서비스 결과를 수신하는 함수
  static void recv_readdir_1(dir_type *);
};

// ROCL의 서버측 기반 클래스인 asyncServerStub로부터 계승
class service_1:public asyncServerStub
{
public:
  // p:서버의 프로그램 번호
  // v:서버의 버전
  // rP:클라이언트의 프로그램 번호
  //rV:클라이언트의 버전
  service_1(u_long p, u_long v, u_long rP, u_long rV)
            :asyncServerStub(p,v,rP,rV)
  {
    DCL_ASYN_SERVICE(1, nametype, &Service_1::recv_readdir_1);
  }
  // 서비스 함수
  dir_type *readdir_1(nametype &);
};
```

(4) RDIR의 비동기 다중 처리 서버 모델 클래스

이 모델의 클래스 정의는 앞에서 설명한 비동기화 모델의 클래스와 유사하며 단지 서버측의 기반 클래스로서 asyncMultiServerStub를 계승하는 차이만이 있을 뿐이다.

4.2 DIRT(Distributed Image Ray Tracer)

DIRT는 클라이언트/서버 모델을 기반으로한 Image Ray Tracing 프로그램이다. 그러나 ROCL에서 제공하는 기본 모델외의 형태인 다중 처리 클라이언트 모델을 기반으로 하고 있다. 이 모델은 그림 4.1에서와 같이 하나의 클라이언트 프로그램이 지역적으로 분리된 다수의 서버들과 통신하기 위해 다중 처리하는 모델로서 클라이언트 프로그램은 서버 프로그램들에게 처리 대상 데이타들을 분배하고 각 서버 프로그램들은 독립된 실행을 한 후 그 결과를 클라이언트 프로그램에게 되돌려주게 된다.

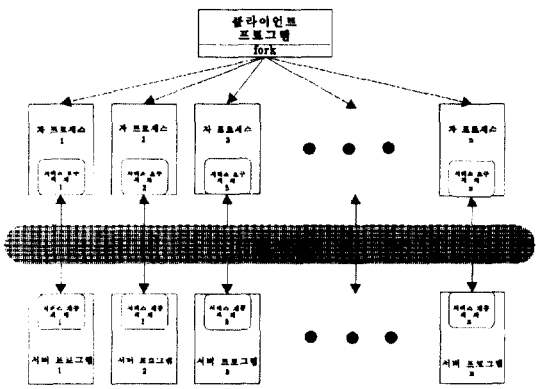


그림 4.1 DIRT의 실행 형태
Fig. 4.1 DIRT execution

구현된 DIRT는 그림 4.2에서 처럼 크게 GUI

(Graphic User Interface)부분, 실제로 이미지를 처리하고 그 결과를 클라이언트 프로그램으로 반환하는 서비스 제공 객체 부분과 서비스 제공 객체에게 처리할 데이타를 넘겨주고 그 결과를 전달받는 서비스 요구 객체 부분으로 구성됨을 알 수 있다.

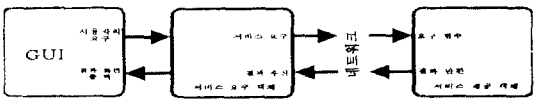


그림 4.2 DIRT의 구성
Fig. 4.2 Construction of DIRT

GUI 부분은 OSF/Motif를 사용하여 작성되었으며 그림 4.3과 같은 이미지 처리에 필요한 데이타들을 대화식으로 입력하고 처리된 이미지를 출력하는 사용자 인터페이스를 제공한다.(3,4,8) DIRT의 서비스 제공 객체와 서비스 요구 객체는 동기화 모델을 기반으로 하여 설계되었다.

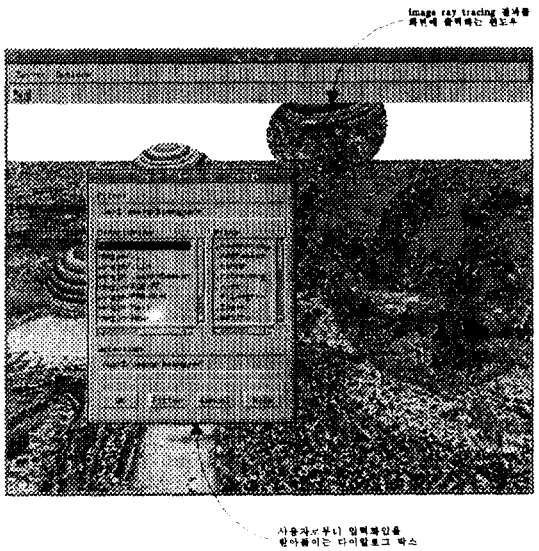


그림 4.3 DIRT의 GUI와 이미지 처리 결과
Fig. 4.3 GUI for DIRT and image processing results

본 논문에서는 7대의 Sun 워크스테이션이 LAN으로 연결되어있는 분산 환경하에서 각각의 호스트에 DIRT의 서버 프로그램들을 적재하고 그 중 하나의 호스트에서 클라이언트 프로그램을 실행하는 실험을 하였다. 그 결과 그림 4.4와 같은 성능 그래프 얻을 수 있었다. 그림 4.4에서 서버 갯수의 증가에 따라 따라 처리 시간이 단축됨을 알 수 있는데 7개째에서는 오히려 통신부하(communication overhead)로 인한 처리 시간의 증가를 보이고 있다. ROCL은 개발자의 필요에 따라 DIRT와 같이 독립된 서버 프로그램과 하나의 클라이언트로 구성된 변형된 형태의 분산 프로그램에서도 사용될 수 있으며 이러한 경우 단일 처리기들의 처리 능력을 극대화한 병렬처리도 가능하다.(9)

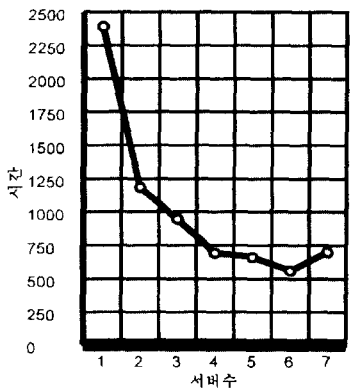


그림 4.4 DIRT의 성능 분석 그래프
Fig. 4.4 Performance graph of DIRT

V. 분석 결과

ROCL은 기본적으로 기존의 RPC에서 제공되는 동기화된 클라이언트/서버 모델을 기반으로 하고 있다. 그러나 그 외에 세가지의 기본 모델들을 더 추가하고 각 모델에 대한 클래스 선언을 통해 클라이언트/서버 모델의 객체 지향 응용 프로그램을 개발할 수 있도록 하였다. 그리고 ROCL을 이용한 응용 프로그램에 있어서 기존의 RPC로 작성된 경우 보다 더욱 축약된 코드로 작성되며 Ⅳ장에서 제시한 예의 경우 프로그램의 라인수가 약 30% 이상이 감소되는 결과를 가져왔다. 따라서 일반적인 클라이언트/서버 모델에 대해서도 이와 유사한 결과를 얻으리라고예상된다. 또한 ROCL은 C++ 언어의 객체 지향 기법에 의한 프로그래밍을 지원하므로 일관된 프로그래밍 형태와 프로그램의 설계, 유지 및 보수의 편이성 등을 제공한다. 표 5.1은 C++ 클래스 라이브러리인 ROCL을 사용한 프로그램의 특성들을 기존의 C로 작성한

표 5.1 ROCL과 기존 RPC간의 특성 비교
Teble 5.1 Comparison of ROCL and conventional RPC

프로그램의 특성	ROCL	기존 RPC
복잡성	서비스 대상만을 설계하면 되므로 설계의 복잡도가 줄어든다.	서비스 대상 뿐아니라 스터브 부분에 대한 설계가 필요하다.
오류 발생	오류가 없는 스터브 부분의 코드가 클래스의 멤버들로 지원되므로써 개발자의 부담을 덜어준다. 즉, 스터브 부분과 관련된 서비스 제공 객체와 서비스 요구 객체의 클래스 구현 부분은 응용 프로그래머에 의한 수정이 필요없으므로 이 부분은 재컴파일 과정없이 사용할 수 있다.	분산 프로그램의 스터브 부분의 코드를 응용 프로그램 개발자가 직접 작성해야하고 서비스 함수의 추가나 삭제와 같은 수정이 발생할 때마다 재컴파일해야 하는 부담이 있다.
모듈화	서버와 클라이언트로 분리하여 개발할 수 있으므로 클래스 단위의 모듈화가 가능하다.	C 언어와 같은 구조화 언어에서 사용되므로 모듈 프로그래밍 가능하다.
이해도	스터브 부분이 객체에 의해 캡슐화 되었으므로 이에 대한 이해는 필요치 않으며 응용 프로그램에 관련된 부분에 관심을 집중할 수 있다.	작성되는 프로그램의 성격에 따라 스터브 부분과 함께 응용 프로그램 전반에 대한 이해가 필요하다.
코드 크기	스터브 부분에 대한 코드가 응용 프로그램에서 없어지므로써 순수한 서비스 부분의 코드만이 남는다.	응용 프로그램마다 스터브 부분의 코드가 삽입되므로 전체 코드량이 늘어난다.
객체 지향 기법	스터브를 서비스 제공 객체와 서비스 요구 객체로 지원한다.	C++와 같은 객체지향 언어에서 지원하는 클래스 라이브러리가 아니므로 함수의 호출 형태로서만 지원할 수 있다.
다양한 서버/클라이언트 모델의 지원 (동기,비동기,다중처리,비동기 다중처리)	ROCL에서 제공하는 기반 클래스를 계승하는 것으로 특정 서버/클라이언트 모델을 선택할 수 있다.	프로그래머가 응용 프로그램에서 필요로 하는 모델을 직접 구현해야한다.

RPC 프로그램의 경우와 비교하여 분석한 결과이다.

Ⅵ. 결 론

분산 프로그래밍은 이를 지원하는 프로그래밍 환경이 필요한데 기존의 RPC 시스템들은 분산 프로그래밍을 위한 좋은 개발 환경을 제공하고 있다. 그러나 이러한 RPC 시스템들의 API들은 C 또는 Pascal과 같은 명령형 언어의 라이브러리로서 제공되므로 C++ 언어와 같은 객체 지향 언어에서 이용할 경우 개발자가 직접 분산 프로그래밍을 위한 객체를 설계, 구현해야 하거나 C++ 응용 프로그램내의 함수 호출 형태로 삽입되므로써 일관성있는 프로그래밍을 작성하기 어렵고 따라서 다른 응용 프로그램의 개발시 재사용할 수 없는 결과를 초래한다. 또한 서버와 클라이언트의 관계에 있어서 서비스 함수가 어느 서버로부터 제공되는지 명확하게 드러나지 않는다. 따라서 서버와 클라이언트를 좀 더 세분화한 모델에 따라 객체라고 하는 추상화된 형태로 재정의하고 서버와 클라이언트간의 관계를 객체간의 관계로 정의 하므로써 좀 더 명확한 관계가 성립될 수 있다.

본 논문에서는 클라이언트/서버 모델의 응용 프로그램을 개발하는데 있어서 서버와 클라이언트를 각각 객체의 관점에서 프로그래밍할 수 있는 환경을 제공하기 위해서 ROCL 이라는 C++ 언어의 클래스 라이브러리를 설계, 구현하였다. 따라서 서비스를 제공하는 객체와 서비스를 요구하는 객체로 그 관계가 명확하게 드러나며 응용 프로그램의 개발자는 이들 객체에 대한 이해없이 서비스만을 설계의 대상으로 할 수 있도록 한다.

제안된 ROCL은 클라이언트/서버 모델을 세분화한 각 모델의 기반 클래스를 제공한다. 이 기반 클래스들은 응용 프로그램에서 계승하여 사용하도록 하였다. ROCL을 사용한 예로서 RDIR은 ROCL에서 제공하는 각 세부 모델들에 대해 모두 적용해본 경우로서 ROCL의 특성을 잘 보여주는 전형적인 예라고 할 수 있다. DIRT는 ROCL에서 제공하는 모델외에도 ROCL이 적절하게 이용될 수 있음을 보여준다.

그러나 실제 분산 프로그램의 개발에 있어서 DIRT의 경우와 다양한 형태의 모델들에 있을 수 있으며

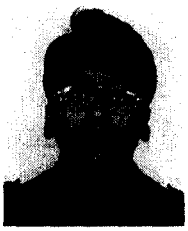
이러한 모델들에 대한 분류와 객체의 설계 및 구현에 관한 더 많은 연구가 필요한다.

참고문헌

1. Andrew S. Tanenbaum, "Modern Operating Systems", Prentice-Hall, 1992.
2. Bjarne Stroupstrup, "The C++ Programing Language", Addison Wesley, 1991.
3. Dan Heller, "Motif Programming Manual Vol. 6", O'Reilly & Associates, 1991.
4. Douglas A. Young "Object-Oriented Programming with C++ and OSF/Motif", Prenice-Hall, 1992.
5. Englewood Cliffs, NJ, "UNIX System V Release 4 Programmer's Guide:Networking Interfaces", AT&T, 1990.
6. John Bloomer, "Power Programming with RPC", O'Reilly & Associates, 1991.
7. Keith E. Gorlen, Sanford M. Orlow and Perry S. Plico, "Data Abstraction and Object-Oriented Programming in C++", John Wiley & Sons, 1990.
8. Mark J. Sebern, "Building OSF/Motif Applications", Prentice-Hall, 1994.
9. Muuss, M.J. "RT & REMRT:shared memory parallel and network distributed ray-tracing programs", USENIX Association Foruth Computer Graphics Workshop, 1987, pp. 86-97
10. Padovano, "Networking Applications on UNIX system V Release 4", Prentice-Hall, 1993.
11. Roger Jenning, "Database Developer's guide with Visual Basic 3", SAMS publishing 1994.
12. Stephen G. Kochan and Patrick H. Wood, "UNIX Networking", Hayden Books, 1989.
13. Sun Microsystems, Inc., "RPC:Remote Procedure Call, Protocol Specification, Version 2", RFC 1057, 1988.
14. Sun Microsystems, Inc., "XDR:External Data Repesentation Standard", RFC 1014, 1987.
15. 문송천, "분산 및 병렬 처리 개론", 정훈출판사, 1991.

崔 英 根 (Young Keun Choi)

1980년 : 서울대학교 사범대학 수학교육과 이학사
1982년 : 서울대학교 계산통계학과 이학석사
1989년 : 서울대학교 계산통계학과 이학박사
1983년~현재 : 광운대학교 이과대학 전산과 부교수
※ 관심분야 : 프로그래밍 언어, 병렬 컴퓨터, 병렬 프로그래밍, 객체지향 프로그래밍 언어 및 설계, 분산 처리 등



趙 光 濟 (Kwang Je Cho)

1993년 : 광운대학교 이과대학 전자계산학과 이학사
1995년 : 광운대학교 이과대학 전자계산학과 이학석사
1995년~현재 : 핸디·소프트 기술연구소 근무
※ 관심분야 : 병렬 프로그래밍, 객체 지향 프로그래밍 언어 및 설계, 분산 처리 등