□특집□

객체지향 클라이언트/서버 시스템 아키텍쳐

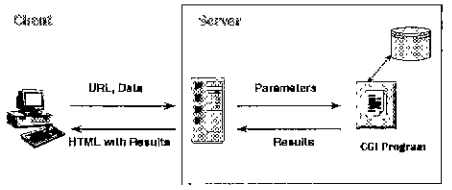
김 수 동† 김 철 진††

◈ 목 차 ◈

1. 서 론

인터넷의 급속한 보급과 대중화로 보다 많은 소프트웨어들이 웹 브라우저 상에서 실행될 수 있는 클라이언트와 CGI 방식의 서버형태로 개발되고 있다. HTML/CGI 방식의 클라이언트/서버 어플리케이션의 구조를 살펴보면 (그림 1)과 같은 구조를 가진다.

클라이언트 컴퓨터는 모든 정보의 검색과 처리를 서버에 요청해야 하고, 서버는 CGI 프로그램을 실행하여 그 결과를 클라이언트 컴퓨터에 보내게 되고 결과는 클라이언트 컴퓨터의 웹 브라우저를 통해 출력되게 된다. 따라서, 클라이언트 컴퓨터는 단지 그래픽 터미널 역할에 불과한 단순 기능만 수행하게 되므로 효율성이 저하되게 된다. 서버는 여러 클라이언트들의 요청들을 받아서 처리해야 하는 모든 기능을 담당하게 되므로 네트워크 트래픽의 집중화와 프로세싱의 과중으로 인한 성능 저하의 문제를 안고 있다.



(그림 1) CGI 기반의 웹 어플리케이션 구조

이와 같은 전통적인 클라이언트/서버 모형의 문제점들을 해결하기 위한 대안책으로 여러 기술들이 지난 2~3년 간에 걸쳐 소개 되고 있다. 대표적으로 Java 언어는 클라이언트 컴퓨터에서의 직접 프로세싱 기능을 가능하게 하며, OMG에서 분산 객체 컴퓨팅의 표준으로 제시한 CORBA는 데이터와 프로세싱이 분산을 가능하게 한다. Java와 CORBA[1]는 모두 객체지향 패러다임(Paradigm)을 채택하고 있어 새로운 객체지향 방식의 클라이언트/서버 모형의 정립과 이를 기반으로 한 시스템 아키텍쳐를 필요로 하고 있다. 본 논문에서는 객체지향 방식의 3계층 클라이언트/서버 모형을 제안하고, 각 계층의 구성 요소들을 정의한 후 8가지의 대표적인 시스템 아키텍쳐들을 소개한다.

† 종신회원 : 숭실대학교 컴퓨터학부 조교수
†† 정 회 원 : 숭실대학교 전자계산학과 석사과정

제안된 아키텍처를 채택한 객체지향 어플리케이션들은 절차적 클라이언트/서버 시스템의 문제점이나 제약 사항들을 해결할 수 있고, 보다 높은 성능과 효율성의 시스템 개발을 지원한다. 본 논문의 구성을 살펴보면, 1장에서는 웹 기반의 절차적 클라이언트/서버 시스템들의 한계점들을 알아보고 2장에서는 절차적 클라이언트/서버 모형의 구성 요소들을 살펴본 후, 3장에서는 객체지향 클라이언트/서버 모형의 구성 요소들을 제시하며, 4장에서는 객체지향 클라이언트/서버 어플리케이션의 8가지 대표적 아키텍처들을 제안하고, 5장에서 사례연구를 통하여 제안된 아키텍처의 활용과정을 살펴본다.

2. 절차적 클라이언트/서버 모형

원격 프로시져 호출(Remote Procedure Call, RPC) 방식과 미들웨어 방식에 근거한 절차적 프로그래밍 방식의 클라이언트/서버 시스템은 3개의 계층으로 구성된다[2]. 사용자와의 인터페이스를 제공하는 사용자 인터페이스 계층, 비즈니스 로직과 트랜잭션을 실행하는 프로세싱 계층, 데이터의 관리 및 처리를 제공하는 데이터 계층으로 구성되어 있다.

2.1 사용자 인터페이스 계층

절차적 클라이언트/서버 시스템에서의 사용자 인터페이스는 윈도우나 그래픽 단말기를 통한 사용자와의 정보 교류를 담당하는 계층이다. C와 같은 절차적 프로그래밍 언어를 사용하여 사용자로부터 입력 데이터를 받고 이를 프로세싱 계층에 전달하며, 프로세싱 계층으로부터 전달 받은 정보나 데이터는 사용자 인터페이스의 API를 이용하여 화면등에 출력한다. 이때, 전달 받은 정보는 사용자가 원하는 형태로 포맷 및 가공되어 진다.

2.2 프로세싱 계층

프로세싱 계층의 소프트웨어 모듈들은 사용자가 원하는 작업이나 트랜잭션 등을 처리하는 계산 중심의 계층이다. 필요한 데이터를 데이터 계층을 통하여 검색한 후 트랜잭션을 수행하며 그 결과를 사용자 인터페이스 계층으로 전송하여 사용자에게 제공한다. 또한, 변경된 데이터를 데이터베이스에 저장하는 요청을 데이터 계층에 보낸다.

2.3 데이터 계층

데이터 계층의 중요한 기능은 어플리케이션에서 사용되는 영구적인 데이터를 저장 및 관리하는 일이다. 대부분의 경우 관계형 데이터베이스 형태로 정보를 저장하며, 필요한 경우 데이터베이스를 여러 사이트에 분산 관리한다.

3. 객체지향 클라이언트/서버 모형

최근 들어 웹 어플리케이션 개발을 위한 객체지향 기술의 유용성이 크게 부각되고 있다. Java와 같은 객체지향 언어를 이용한 웹 어플리케이션의 경우 사용자에게 보다 능동적인 사용자 인터페이스와 트랜잭션 프로세싱을 제공할 수 있게 되고, 애플릿(Applet)과 같은 형태로의 소프트웨어 재사용이 실현되어 효율적인 조립형 소프트웨어 생산이 가능해 지기 때문이다. 본 장에서는 클라이언트/서버 웹 어플리케이션 개발에 활용될 수 있는 객체지향 클라이언트/서버 시스템에서의 구성요소를 살펴본다. 객체지향 클라이언트/서버 시스템은 절차적 클라이언트/서버의 경우처럼 세 개의 계층들로 구성되고 있으나, 각 계층의 구성 요소는 크게 다르다.

3.1 사용자 인터페이스 계층(UI Layer)

객체지향 프로그램에서는 사용자 인터페이스를

제공하는 모듈의 구성단위가 객체이다. Visual C++의 MFC 나 Java의 AWT의 경우처럼 사용자 인터페이스 윈도우, 버튼, 아이콘 등에 해당되는 객체들이며, 이들 객체들은 각 Widget을 표현하는 데이터와 사용자 정보의 입출력 및 화면의 내용을 변경하는 등의 기능성을 함께 가지고 있다.

절차적 클라이언트/서버에서의 사용자 인터페이스 계층이 데이터와 관련 함수들을 별도의 부분으로 취급하는 반면에[2] 객체지향 사용자 인터페이스 계층의 객체들은 관련된 사용자 인터페이스 데이터와 관련된 함수들을 한 덩어리로 가지고 있다[3]. 이때 각 객체는 사용자로부터의 입력을 프로세싱 계층의 해당 객체들에게 전송하며, 프로세싱 계층의 객체들로부터의 트랜잭션 처리 결과는 화면이나 프린트 등에 출력을 한다.

사용자 인터페이스 계층의 객체는 프로세싱 계층에 있는 해당 객체를 참조 형태로 가지게 되며, 프로세싱 계층도 해당 사용자 인터페이스 객체들을 참조하게 된다. 이러한 참조 관계는 객체지향 분석 과정에서 객체 모형도와 이벤트 추적도(Event Trace Diagram)에 표현된다.

3.2 프로세싱 계층(Processing Layer)

객체지향 프로그램에서는 모든 모듈들이 객체로 구성되어 있으므로, 객체지향적인 프로세싱 계층의 정의가 요구된다. 즉, 프로세싱 객체는 사용자 인터페이스 객체로부터 요청받은 트랜잭션을 처리하며, 이를 위하여 데이터 계층의 영구(Persistent) 객체들로부터 관련된 정보를 전송받아 트랜잭션을 실행시킨다. 그 결과는 사용자 인터페이스 계층에 전송되며, 필요시 데이터 계층에 새로운 객체의 상태를 저장한다.

트랜잭션 처리에 필요한 영구 데이터는 데이터 계층에서 전송되어 오므로, 프로세싱 계층의 객체 안에 있는 데이터는 현재 트랜잭션 처리에 필요한 버퍼(Buffer), 임시 변수 및 값을 가지고 있다. 이들 임시 변수의 값들은 데이터 계층내의 영구 객체들이 가지고 있는 값들과는 성격이 구별된다. 따라서, 객체지향 클라이언트/서버 모형에서의 프로세싱 객체는 트랜잭션 처리의 기능성과 임시 데이터를 가지고 있다.

3.3 데이터 계층(Data Layer)

일반적으로 데이터 계층은 도메인에서 발견되는 영구적인 정보나 데이터를 저장 및 관리하는 것인데, 객체지향의 데이터 계층은 객체들로 구성되어 있다. 이들 객체들의 공통된 특징은 객체 내부의 데이터 부분이 지속적으로 저장되어야 할 영구적인 정보라는 점이다.

객체의 두번째 구성요소는 기능 부분인데, 객체 내부의 데이터를 처리하기 위한 기능들이 여기에 포함된다. 데이터 계층의 객체들이 가지고 있는 기능성은 영구 객체들을 사용하는 대부분의 응용 어플리케이션들이 공통으로 필요로 하는 기능들이다. 즉, 영구 객체 내부의 데이터를 처리하기 위해 잘 정의된 필수 기능들로서, 프로세싱 계층의 객체들이 각 응용 어플리케이션에 종속적인 특정 트랜잭션 처리를 위한 함수들을 가지고 있는 것과는 대조적이다.

3.4 절차적 모형과 객체지향 클라이언트/서버 모형의 비교

절차적 클라이언트/서버 모형과 객체지향 클라이언트/서버 모형은 기본적으로 3-Tier 구조를 지닌 3계층으로 구성되어 있다는 점에서 공통점을 지니고 있다. 그러나, 각각의 계층을 구성하는 구성 요소에 대한 시각이 다르기 때문에 계층 내에 있는 각 구성요소의 내용이 크게 다르다.

절차적 방식의 모형은 데이터와 기능 부분이 분리되어 구성된 반면에, 객체지향 방식의 모형은

객체 단위로 각 계층이 구성되어 있다. 절차적 방식에서의 사용자 인터페이스 계층은 데이터와 관련 함수들을 별도의 부분으로 취급하는 반면에, 객체지향 사용자 인터페이스 계층에서는 사용자 인터페이스 객체가 사용자 인터페이스 데이터와 관련된 함수들을 함께 가지고 있다.

절차적 방식에서는 프로세싱 계층이 트랜잭션 처리를 위한 함수들로만 이루어져 있지만, 객체지향 방식에서는 이러한 함수들이 어떤 객체 내의 기능 부분으로 모델링 되고 구현된다.

또한 절차적 방식에서는 대부분의 데이터가 데이터 계층에 속하게 되지만, 객체지향 방식에서는 프로그램 내의 모든 데이터들이 어떤 객체 안의 데이터 부분으로 모델링되고 구현된다. 영구적으로 저장되어야 할 데이터를 지닌 객체들은 객체지향 데이터베이스에 저장되며, 이러한 데이터를 처리하기 위한 함수들이 객체 내부에 포함되어 있다.

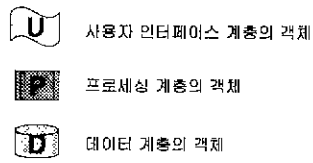
지금까지, 절차적 클라이언트/서버 모형과 객체지향 클라이언트/서버 모형의 구성 요소들에 대한 차이점을 제시하였다. 절차적 방식과 객체지향 클라이언트/서버 모형의 차이점들을 요약하면 <표 3-1>과 같다.

<표 3-1> 절차적 모형과 객체지향 모형간의 비교

항목 \ 모형	절차적	객체지향
단위	모듈	객체
계층간의 의존성	강하다	약하다
계층내의 응집력	약하다	강하다
시스템 처리 속도	늦다	빠르다
네트워크 트래픽	많다	적다
시스템 관리	복잡하다	단순하다
시스템 변경	어렵다	용이하다
재사용성	낮다	높다
유지보수성	어렵다	용이하다
확장성	어렵다	용이하다

객체지향 클라이언트/서버 모형은 이전 절에서 정의한 세개의 계층에 해당되는 객체들을 클라이언트 호스트와 서버 호스트의 어느 곳에 위치시키는지, 또한 어떻게 분할하는 가에 따라서 여러 모형들로 세분화 된다.

객체지향 클라이언트/서버 모형들을 정의하기 이전에 세 개의 계층에 해당되는 객체들의 이름과 아이콘을 (그림 2)처럼 정의한다.



(그림 2) 객체지향 클라이언트/서버 모형의 객체 종류

4. 객체지향 클라이언트/서버시스템 아키텍쳐

객체 지향 클라이언트/서버 모형은 적용 기술에 따라서 여러 가지 형태의 모형들로 나타난다. 본 장에서는 객체 지향 클라이언트/서버 모형을 8가지 형태의 모형들로 분류하며, 각각의 모형별 아키텍쳐 및 장/단점에 대해 제시한다.

본 장에서는 성능이 높고 효율적인 클라이언트/서버 시스템을 구축하기 위해 적용 기술을 이용하여 객체들을 클라이언트와 서버 호스트에 분할하도록 제안한다. 그리고, 본 연구에서는 다음과 같은 가정을 전제로 하고 있다.

본 연구의 접근 방식이 객체지향을 기반으로 하기 때문에 클라이언트와 서버간에 존재하는 각각의 사용자 인터페이스 객체, 프로세스 객체, 그리고 데이터 객체들은 자체적으로 강한 응집도(Cohesion)를 가지고 있다고 본다. 또한 사용자 인터페이스 객체, 프로세스 객체, 그리고 데이터 객체 내부의 상속성이나 집합성으로 인한 결합도는 서로 다른 부류 객체들 사이의 상속성이나 집합

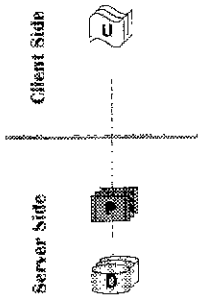
성 보다 높기 때문에 고려하지 않았다.

4.1 모형 #1: Fat Server

4.1.1 아키텍처

Fat Server 모형은 기존의 CGI를 기반으로 한 클라이언트/서버 모형과 유사한 형태의 아키텍쳐를 가진다. 그림 3에서 제시된 것처럼 클라이언트 호스트에는 단지 사용자 객체들 만이 존재하게 되며, 서버 호스트에 프로세스 객체 집합과 데이터 객체 집합이 존재한다.

이 모형은 서버측에서 데이터 저장이나 관리 그리고 모든 프로세싱을 담당하기 때문에 Fat Server 모형으로 명명한다.



(그림 3) Fat Server 클라이언트/서버 모형

클라이언트 측에는 사용자 인터페이스 객체가 존재하여 사용자의 입력 등 사용자가 직접 이용하는 전반적인 내용들의 객체들을 포함하고 있다. 기존 클라이언트/서버 모형에서의 사용자 인터페이스 계층과 다른 점으로는 사용자의 입력에 대한 무결성 검사, 날짜의 자동 입력, 반복 데이터에 대한 자동 입력 등 간단한 사용자 입력에 도움을 줄 수 있는 프로세싱이 행하여 진다는 점에서 그 차이점을 둘 수 있다. 즉 기존 클라이언트/서버 모형의 사용자 인터페이스 계층의 수동적인 형태에서 벗어 날 수 있다. 서버 측에는 프로세스 객체와 데이터 객체가 존재하며, 데이터 객체는 데이터를 저장하고 데이터 처리의 기본적이고 공통적인 오퍼레이션을 수행하는 메소드들을 가진다. 이외 영구적인 상태를 갖는 데이터와 이를 조작하는 메소드들도 함께 보관할 수 있다.

이러한 방법으로 프로세스 객체의 메소드와 데이터 객체의 메소드들로 나누어 놓는 것은 프로세스 객체에 여러 번 정의 할 수 있는 사항을 데이터 객체에 영구 객체화 함으로서 시스템 공통적으로 사용하는 내용은 한번만 정의하여 효율을 높이고자 하는 이유를 바탕으로 하고 있다. 그 예로서 한 데이터 베이스를 두 애플리케이션이 사용한다면 데이터 베이스 내에 공통 오퍼레이션을 정의하여 두 애플리케이션에서 공유하여 사용할 수 있게 되는 것이다.

또한, 본 모형은 데이터 베이스 접근을 행하는 메소드들을 프로세싱만을 행하는 메소드들과 구분하여 놓으므로써 확장성과 보안성을 확보하고 있다. 즉, 사용자의 입력을 받는 계층, 프로세싱이 일어나는 계층, 공통적인 데이터 처리의 공통기능을 수행하는 계층의 3계층으로 구성된다.

Fat Server 모형은 위의 설명한 형태로 구성되어, 모든 프로세싱은 서버에서 발생하게 된다. 오퍼레이션 수행시 클라이언트는 사용자 인터페이스 객체를 통하여 서버측의 프로세스 객체를 호출하게 되고, 프로세스 객체는 데이터 관련 처리 공통 기능을 행하는 데이터 객체의 메소드들과 데이터들을 이용하여 프로세싱을 행하게 된다. 이후 프로세스 객체는 생성된 결과 값을 클라이언트 측의 사용자 인터페이스 객체로 보내줌으로써 오퍼레이션은 끝나게 된다.

4.1.2 장/단점

이 모형의 장점은 애플리케이션 개발시 서버측에서 독립된 시스템 구축과 동일한 형태로 데이터 객체와 프로세스 객체를 쉽게 구성할 수 있으며, 관리 시에도 서버측의 한 곳의 내용을 변경함

으로써 쉽게 프로세스 객체들의 내용을 변경할 수 있는 장점을 가질 수 있다. 그리고 클라이언트 상에서는 사용자 인터페이스 객체에 의해 사용자의 입력에 대한 무결성(Integrity) 검사를 할 수 있고, 동적인 사용자 인터페이스를 제공할 수 있다는 점에서 기존 클라이언트 서버 모형 중 프로세스 계층과 사용자 인터페이스 계층의 분리 모형이 갖는 정적인 사용자 인터페이스의 역할을 극복할 수 있는 장점을 가진다.

이 모형이 지니는 가장 큰 단점으로는 모든 프로세싱이 서버측에서 발생하므로 서버 측에 과부하가 생길 가능성이 높다. 이 모형에서 클라이언트는 단지 서버에 서비스를 요청만 하고, 서버가 처리한 결과 값이 반환 될 때까지 대기해야 하기 때문에 클라이언트는 전반적으로 처리속도가 저하되는 경향을 보일 수 있다.

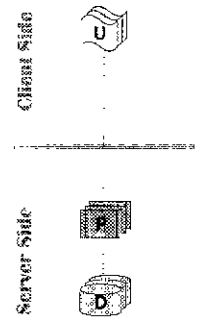
또한 클라이언트에는 사용자 인터페이스 객체들만이 존재하므로 클라이언트의 역할이 트랜잭션 프로세싱을 행하지 않으므로 사용자 인터페이스 객체들만을 나타내는 한정적이고, 수동적인 역할만을 수행하게 된다는 것이다.

4.2 모형 #2: Data Server

4.2.1 아키텍처

이 모형은 서버의 부하를 줄이고 클라이언트 호스트를 활용하기 위한 모형으로서, 그림 4에서 보이는 것처럼 프로세스 객체들을 클라이언트 호스트로 옮김으로써 클라이언트 호스트에서 프로세싱을 수행할 수 있기 때문에 클라이언트가 보다 능동적인 처리를 할 수 있다.

클라이언트를 단순한 터미널로 사용하는 방식이나 Fat Server모형에서와 같이 사용자와의 인터페이스만을 위해 사용하는 방식보다는 클라이언트가 갖고 있는 처리 능력을 보다 더 활용하기 위한 모형이다.



(그림 4) Data Server 클라이언트/서버 모형

이 모형에서 사용자 인터페이스 객체는 클라이언트 측에만 위치하므로 클라이언트 종속적인 인터페이스를 제공한다. 그러므로, 시스템에 공통적인 인터페이스, 예를 들면 시스템 모니터링 또는 시스템 관리 인터페이스를 담당하는 사용자 인터페이스 객체들을 제공하기 위해서는 별도의 방법을 써야 한다. 사용자마다 권한을 부여해서 접근을 제한하는 방법이 좋은 예라 할 수 있다.

프로세스 객체도 사용자 인터페이스 객체와 같이 클라이언트 측에 위치하므로 사용자 인터페이스를 통해 요구된 서비스를 처리하고 계산하는 클라이언트 종속적인 서비스를 제공한다. 이 모형에서는 프로세스 객체가 서버측에 없으므로 시스템 관리 기능과 같이 서버에서 처리해야 할 특정 업무를 프로세스 객체가 처리하지 못하고, 데이터 객체를 통해 시스템에서 데이터를 공유하는 방식으로 제공할 수 있다. 또한, 시스템 관리 기능을 제공하기 위해서는 사용자의 권한을 체크하는 프로세스를 클라이언트 측에 두고 서버에 저장된 데이터를 참조하여 권한을 체크하는 방식을 사용할 수 있다.

데이터 객체는 모두 서버측에 위치하므로, 서버에 종속적이며 모든 클라이언트들에 공통적인 데이터들을 관리하고 데이터의 활용은 모두 클라이언트 측에서 이루어 진다.

이 모형은 객체들이 클라이언트에 집중되어 있다는 점에서 Data Balanced 클라이언트/서버 모형과 상당히 유사하고, 객체가 서버에 집중된 Fat Server 모형과는 대조를 이루는 모형이다.

4.2.2 장/단점

이 모형의 장점은 클라이언트의 처리 능력에 대한 활용성을 높임으로써, 서버의 부하를 줄일 수 있다. 즉, 하드웨어 비용에 비해 계산 성능이 우수한 개인용 컴퓨터의 처리 능력을 활용함으로써, 하드웨어 비용이 많이 소요되는 서버의 사양을 다소 줄일 수 있다. 따라서, Fat Server 모형보다 시스템의 하드웨어 구축 비용을 줄일 수 있으며, 사용자가 많은 시스템인 경우에는 처리에 따른 서버의 부하를 줄이는 효과가 크기 때문에 Fat Server 모형보다 향상된 시스템 성능을 얻을 수 있다.

또 다른 장점으로는 전체적인 시스템 구조를 단순하게 만들 수 있다는 데 있다. RMI나 CORBA같은 기술을 사용하지 않고, 소켓을 사용해 클라이언트의 사용자 인터페이스 객체와 프로세스 객체가 서버의 데이터 객체와 메시지를 주고 받도록 소프트웨어를 구축할 수 있다. 소켓에 비해서 RMI와 CORBA는 여러 가지 부가서비스를 제공하여 소프트웨어의 구조를 복잡하게 하므로, Data Server 모형과 같이 객체의 분산이 단순한 모형에서는 소켓을 사용하도록 한다.

단점들 가운데 하나는 프로세스 객체의 분산이 이루어지지 않는다. 공통적인 비즈니스 프로세스에 대해서는 서버가 처리해 주는 것이 좋으나 이를 지원하지 못한다. 전체 시스템에 대해 관리하는 기능이 미약하거나 지원하지 않는 모형이므로, 중.대형 과제에는 적합하지 않다.

또 다른 단점으로는 데이터 객체의 분산이 이루어지지 않고 있다. 클라이언트마다 갖고 있어야 할 데이터들까지도 서버에 저장함으로써 네트웍 트래픽이 증가하며, 특정 클라이언트에 필요한 데이터를 저장하기가 곤란한 모형이다.

또한 영구적으로 저장된 데이터를 활용하기가 어렵다. 즉, 데이터 서버 모형에서는 서버에 프로세스 객체가 없으므로, 서버의 메소드를 직접 호출하거나 데이터 객체의 레퍼런스(Reference)를 반환 받을 수 없어 이를 직접 클라이언트가 활용하지 못하고 특정 데이터 스트림만을 받을 수 있다. 이 모형에서는 소켓 또는 데이터그램방식을 사용하여 서버의 데이터 객체로부터 서비스를 받는데, 이는 프로그램의 복잡도를 증가 시킨다.

4.3 모형 #3: Process Balanced 클라이언트/서버

4.3.1 아키텍쳐

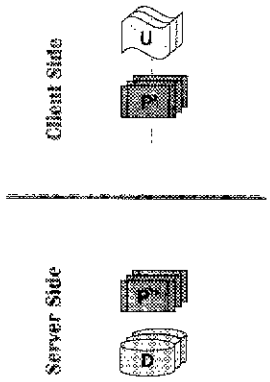
이 모형은 프로세스가 서버 호스트에 치우쳐 있는 전형적인 클라이언트/서버 모형과는 달리 그림 5에 나타난 것처럼 프로세스 객체들이 클라이언트 호스트와 서버 호스트에 분산되어 있는 경우이다.

이 모형은 도메인에서 처리할 대부분의 트랜잭션 부분인 프로세스가 서버 부분에 치우쳐 있는 기존의 전형적인 클라이언트/서버 모형과는 달리 클라이언트 부분과 서버 부분에 분산되어 있는 경우이다. 이러한 모형에서는 클라이언트 부분에 특정적인 프로세스 객체는 클라이언트에 할당하게 되고, 모든 클라이언트들이 공통적으로 사용하는 프로세스 객체는 서버에 할당하게 된다.

클라이언트 부분에는 클라이언트가 서버에 서비스를 요청하거나 또는 요청된 서비스 결과를 보여주는 데 필요한 사용자 인터페이스 객체(U)들과 해당 클라이언트에 종속적인 계산을 처리하는 프로세스 객체(P)들이 존재한다.

서버 부분에는 특정 클라이언트에 종속적이기보다는 모든 클라이언트들이 공통적으로 필요로 하는 계산이나 기능을 담당하는 프로세스 객체(P)

들과 프로세스 객체들이 계산을 하는 데 필요한 데이터 객체(D)들이 존재한다.



(그림 5) Process Balanced 클라이언트/서버 모형

클라이언트 부분에 놓이는 프로세스 객체들은 특정 클라이언트에 종속적인 결과를 보내준다. 즉, 다수의 클라이언트들이 서버에 접속하는 경우 클라이언트마다 해당 클라이언트 프로세스 객체에 서비스를 요청하면, 계산된 결과가 각각의 클라이언트에 국한되는 것이다. 예를 들어, 인터넷 상에서 브라우저를 통해서 사용자가 상품을 구매하는 경우에 있어서, 사용자가 구매하고자 하는 상품들의 개수를 선택하면 이에 대한 처리를 서버에 있는 프로세스 객체(P)가 담당하는 것이 아니라, 클라이언트에 있는 프로세스 객체(P)가 담당하여 계산된 결과를 클라이언트에 보내준다. 따라서, 여러 사용자가 구매하고자 하는 상품의 개수가 다를 수 있기 때문에 각각의 클라이언트들이 요청한 서비스들을 서버 부분에 있는 프로세스 객체가 처리하는 것보다 각각의 클라이언트 부분에 있는 프로세스 객체들이 처리함으로써 네트워크 트래픽을 줄일 수 있을 뿐만 아니라 처리 속도 또한 빨라지게 된다.

이와 달리, 서버 부분에 놓이는 프로세스 객체들은 모든 클라이언트들이 공통으로 필요로 하는 프로세스 객체들이다. 모든 클라이언트들에 동일한 기능들을 각각의 클라이언트에 프로세스 객체들을 두는 것보다는 하나의 서버에 두는 것이 처리 속도나 자원의 효율적인 사용을 가져온다. 예를 들어 전자 상거래와 같은 시스템에서 주문이나 결제와 같이 트랜잭션 처리를 담당하는 프로세스 객체들을 각각의 클라이언트에 두기 보다는 서버 한 곳에 둠으로써 여러 클라이언트들이 동시에 물품 주문이나 결제를 할 수 있다. 따라서 시스템의 병행성이 향상되고 클라이언트의 자원 또한 효율적으로 사용할 수 있게 된다.

이 모형에서 프로세스 객체들이 분산된다고 해서 객체들이 완전히 독립적으로 분리되는 것이 아니라 메시지 전송을 통해 객체들 간의 상호작용이 이루어지게 된다. 즉, 클라이언트 부분에 있는 사용자 인터페이스 객체나 프로세스 객체가 서버 부분에 있는 프로세스 객체나 데이터 객체와 통신을 하게 되는 것이다. 예를 들어, 클라이언트 부분에 있는 프로세스 객체가 계산에 필요한 데이터를 서버에 있는 데이터 객체로부터 가져 오는 경우나, 클라이언트의 프로세스 객체가 계산을 하는 도중에 모든 클라이언트들이 공통적으로 사용하는 서버에 있는 프로세스 객체에게 서비스를 요청할 수도 있다.

4.3.2 장/단점

이 모형이 지니는 장점 중의 한가지는 네트워크 트래픽이 감소하게 된다는 것이다. 이는 프로세스 객체들이 서버 부분에 치우쳐 있지 않고 클라이언트와 서버에 분산되어 있기 때문에, 클라이언트가 필요로 하는 기능을 네트워크를 통하지 않고 클라이언트 자체에서 실행될 수 있기 때문이다.

다른 장점으로는 시스템 처리 속도가 향상된다는 것이다. 모든 객체들이 서버에 집중되어 있기 보다는 프로세스 객체들이 클라이언트와 서버에

분산되어 있기 때문에 클라이언트와 서버 간의 메시지 전달 횟수가 서버에 집중되어 있을 경우보다 적어진다. 기존의 전형적인 클라이언트/서버 모형에서는 프로세스 객체들이 분산되어 있는 것이 아니라 프로세스와 데이터 객체들이 모두 서버에 포함되어 있어서, 클라이언트가 필요로 하는 서비스들이 모두 네트워크를 통해서 서버에 전달이 되기 때문에 클라이언트와 서버간의 메시지 전달이 자주 발생하므로 시스템 처리 속도가 느리다. 그러나 이 모형에서는 일부 프로세스 객체들이 클라이언트에서 수행될 수 있어서 클라이언트들이 공통으로 필요로 하는 서비스들만 네트워크를 통해 서버에 전달이 되기 때문에 클라이언트와 서버 간의 메시지 전달이 적게 발생한다.

또한 기존의 클라이언트/서버 모형에 비해 서버의 부담이 줄어든다는 것이다. 모든 프로세스 객체와 데이터 객체들이 서버 부분에만 집중되어 있지 않고 여러 클라이언트들이 공통으로 필요로 하는 프로세스 객체들과 데이터 객체들만 서버에 놓이기 때문이다.

데이터 서버 모형과 같이 클라이언트에 프로세스 객체가 위치하므로 클라이언트가 능동적으로 처리될 수 있다. 이는 클라이언트에 종속된 프로세스 객체들이 있음으로 클라이언트에서 일부 처리 기능들을 수행할 수 있기 때문이다. 기존의 클라이언트/서버 모형에서는 클라이언트가 단지 서버에 서비스만을 요청하기 때문에 매우 수동적인 그래픽 터미널에 불과했지만, 이 모형에서 제시된 것처럼 클라이언트에서도 일부 기능들을 수행할 수 있도록 함으로써 클라이언트의 자원들을 보다 효율적으로 이용할 수 있게 되는 것이다.

이 모형이 지니는 단점 가운데 하나는 프로세스 객체들이 적절하게 클라이언트와 서버에 분할되지 않으면 시스템 성능 저하를 초래할 수 있다. 즉, 클라이언트에 놓여야 할 객체들이 서버에 놓인다거나 혹은 서버에 놓여야 할 프로세스 객체들이 클라이언트에 놓이면 잦은 메시지 전송으로 인해 시스템 처리 속도가 상당히 저하되게 된다.

이 모형에서는 프로세스 객체들은 분산되어 있지만, 모든 클라이언트들이 필요로 하는 데이터 객체들은 여전히 서버 부분에 모두 놓여 있기 때문에 서버의 부담이 클라이언트에 비해서 크다고 할 수 있다. 그리고, 클라이언트에 프로세스 객체들이 있지만 이 프로세스 객체들이 필요로 하는 데이터 객체들은 서버에 있기 때문에 데이터 전송은 네트워크를 통해서 이루어지게 된다.

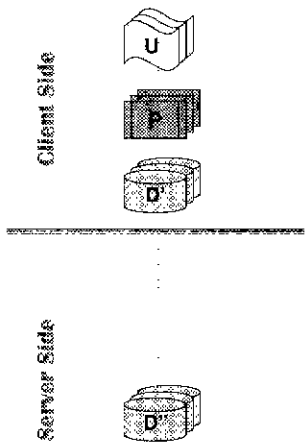
4.4 모형 #4: Data Balanced 클라이언트/서버

이 모형은 데이터 서버 모형에 추가로 데이터 객체가 클라이언트와 서버 호스트에 분산되어 있는 경우로서, 그림 6에 나타난 것처럼 클라이언트 호스트에 있는 지역 데이터베이스에 일부 데이터 객체들이 놓이게 되고 나머지는 서버 호스트에 있는 데이터 베이스에 놓이게 된다.

즉, 클라이언트 부분에는 사용자 인터페이스 객체(U), 프로세스 객체(P), 그리고 클라이언트에 종속적인 일부 데이터 객체(D')가 할당되고 서버 부분에는 모든 클라이언트들이 공통으로 사용하는 데이터 객체(D'')만이 할당되어 있는 모형이다.

이 모형에서 클라이언트에 존재하는 데이터 객체들은 클라이언트에 종속적인 데이터 객체들이다. 즉, 각각의 클라이언트가 지역 데이터 베이스를 갖는 것이다. 예를 들어 클라이언트에 종속적인 데이터 객체는 해당 클라이언트 사용자의 시스템 사용 기록 또는 사용자 등록 정보 같은 것이 이에 해당된다. 반면에 서버에 존재하는 데이터 객체들은 모든 클라이언트들이 공통적으로 필요하는 데이터 객체들이다. 예를 들어 이러한 객체들은 인터넷 상의 전자 상거래와 같은 시스템인 경우에는 상품에 대한 정보 등이 해당된다.

이외에도 클라이언트에 할당되는 데이터 객체들에는 멀티미디어 데이터 객체들도 해당된다. 이는 데이터 크기가 크기 때문에 많은 네트워크 트래픽을 요청하는 데이터들을 클라이언트에 CD-ROM과 같은 형태로 클라이언트 데이터베이스(Local Database)에 분산시킴으로써 네트워크 트레픽을 줄이기 위함이다.



(그림 6) Data Balanced 클라이언트/서버 모형

이 모형은 데이터 서버 모형과 유사하다. 데이터 서버 모형과의 차이점은 데이터 객체가 서버에 집중되어 있지 않고, 클라이언트와 서버에 분산되어 있다는 것이다. 따라서, 클라이언트에 모든 프로세스 객체들이 존재하기 때문에 데이터 서버 모형처럼 실행이 클라이언트에서 이루어지게 되며 해당 클라이언트들이 각각 지역 데이터베이스(Local Database)에 클라이언트에서 필요한 데이터 객체들을 갖고 있는 형태이다. 따라서, Fat Server 모형에서는 객체들이 서버에 집중되어 있는 반면에 이 모형은 객체들이 클라이언트에 집중되어 있는 형태이다.

4.4.1 장/단점

이 모형의 장점은 서버 측면에서 볼 때 서버의 부담이 줄어 들고, 모든 데이터 객체들을 서버에 집중시키기 보다는 클라이언트에 필요한 데이터 객체들은 클라이언트에 할당하고 여러 클라이언트들이 공통으로 사용하는 데이터 객체들은 서버에 할당한다.

다음 장점으로는 클라이언트 호스트의 처리 속도가 빨라진다. Data Server나 Process Balanced 클라이언트/서버 모형에서는 프로세스 객체가 클라이언트에 존재하지만 처리에 필요한 데이터 객체들이 모두 서버에 집중되어 있기 때문에 네트워크를 통해서 데이터 객체들을 전송 받는 데 걸리는 시간으로 인해 속도가 저하되게 된다. 반면에 본 모형을 적용하는 경우는 클라이언트에서 처리되는 프로세스 객체가 필요로 하는 데이터 객체들이 클라이언트에도 존재하기 때문에 네트워크를 통한 경우보다 처리 속도가 빨라진다.

또다른 장점으로는 네트워크 트래픽이 감소한다. 클라이언트에 프로세스 객체와 일부 데이터 객체들이 존재 하기 때문에 네트워크를 통한 클라이언트와 서버 간의 메시지 전송 횟수가 줄어들게 된다.

이 모형의 단점으로는 클라이언트의 부하가 크다. 데이터들을 처리하는 모든 프로세스 객체들이 클라이언트에 집중되어 있음으로 서버의 부담이 줄어든 반면에 클라이언트의 부하가 커지게 된다.

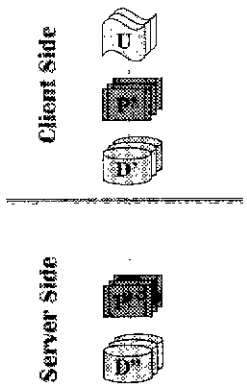
다른 단점으로는 시스템 효율성이 떨어진다. 모든 프로세스 객체들이 클라이언트에만 집중되어 있음으로 인해서 공통된 기능들을 클라이언트들이 각각 갖기 때문에 시스템의 병행성이 떨어지고 클라이언트 자원의 낭비를 가져올 수 있다.

또다른 단점으로는 서버에 있는 데이터 객체들을 많이 사용하는 경우는 클라이언트에 있는 프로세스 객체가 네트워크를 통해서 서버로부터 객체들을 로딩(Loading)해야 하기 때문에 네트워크 트래픽이 증가된다. 따라서, 이러한 모형을 적용하는 경우는 프로세스 객체들이 서버에 있는 데이터들을 가능한 적게 참조하는 시스템에 적합하다.

서버가 매우 수동적이게 된다. 이 모형은 기존의 클라이언트/서버 모형과는 달리 클라이언트는 상당히 능동적인 반면에 서버가 매우 수동적인 경우이다. 따라서, 서버는 단지 데이터 객체들만을 지니고 있기 때문에 클라이언트로부터의 요청이 들어오는 경우에만 데이터 객체들을 제공하게 되고, 클라이언트들로부터의 요청이 없는 경우는 거의 휴무상태(Idle State)에 있게 된다. 이러한 경우는 서버가 지니고 있는 자원들을 낭비하게 되며, 오히려 시스템의 불균형도 초래하게 된다. 그러나 이 모형은 클라이언트들이 공통으로 필요로 하는 데이터 객체들이 많고, 각각의 데이터 객체들의 크기가 크거나 복잡한 경우들로 구성된 경우에 적합하다.

4.5 모형 #5: Data/Process Balanced 클라이언트/서버

이 모형은 그림 7에 보이는 것처럼 프로세스와 데이터 객체가 모두 분산되어 있는 형태를 가진다. 클라이언트의 프로세스 및 데이터 객체는 서버의 프로세스 객체나 데이터 객체와 독립적으로 운영된다. 따라서, 이 모형은 여러 클라이언트가 서버의 기능을 공유하면서, 각각의 클라이언트는 독립된 자료를 처리할 수 있다.



(그림 7) Data/Process Balanced 클라이언트/서버 모형

이 모형은 프로세스와 데이터가 모두 분산되어져 있는 이전에 소개된 Process Balanced 클라이언트/서버 모형과 Data Balanced 클라이언트/서버 모형의 혼합된 형태를 가지고 있다. 클라이언트의 프로세스/데이터 객체는 서버의 프로세스/데이터 객체와 독립적으로 운영되어진다. 따라서 이 모형은 여러 클라이언트가 서버의 기능을 공유하며, 각각의 클라이언트는 다른 클라이언트와 독립된 자료 처리를 목적으로 한다. 클라이언트와 서버간에 프로세스를 분산함으로 모든 연산이 한 곳에서만 일어날 경우 생길 수 있는 과부하를 막을 수 있으며, 각각의 클라이언트가 필요로 하는 공통된 연산을 서버가 수행하여 모든 클라이언트가 이용할 수 있게 함으로 해서 자원의 낭비를 막고, 성능을 높일 수 있다. 또한 데이터를 분산하여 모든 클라이언트가 공유하거나 이용해야 하는 데이터는 서버에 위치시켜 정보의 중복을 막을 수 있으며, 각 클라이언트가 유지하는 정보들은 클라이언트에 위치시켜, 각 클라이언트의 상태 등을 자율적으로 저장할 수 있고 보안에 관련된 자료도 각각의 클라이언트에서 처리할 수 있게 한다.

사용자 인터페이스 객체(U)는 주로 클라이언트의 프로세스 객체(P)와 메시지를 주고 받으며 서버와의 통신을 위해 서버의 프로세스 객체(P)와 메시지를 주고 받는다. 전자의 경우에는 주로 클라이언트에 의존된 작업을 수행하게 된다. 예를 들면 클라이언트가 처리할 수 있는 계산이나 클라이언트의 데이터에 접근하는 작업 등이 이 경우에 속한다. 후자의 경우에는 모든 클라이언트에게 영향을 주는 계산이나 서버 객체의 상태를 변경 시킬 때, 그리고 서버의 데이터에 접근할 때 주로 사용되어진다. 클라이언트의 데이터 객체(D)는 클라이언트의 사용자 인터페이스 객체(U)나 프로세스 객체(P)에 의해서 접근되어진다. 따라서 클라이언트의 데이터 객체

(D)는 주로 클라이언트에 관한 정보나 클라이언트에서만 사용되어지는 정보만을 가지게 된다. 이와는 반대로 서버쪽의 데이터 객체(D)는 모든 클라이언트가 사용하기 때문에 공통적인 정보를 가지고 있다. 이 데이터는 주로 서버의 프로세스 객체(P)가 처리하여 각 클라이언트에게 전달한다.

4.5.1 장/단점

이 모형의 장점은 프로세스와 데이터 객체를 분산하여 놓으므로 클라이언트나 서버 어느 한쪽에만 작업이 치우치지 않아 작업부하를 방지할 수 있으며, 지역 데이터를 사용함으로 네트워크 부하가 줄어든다.

또한 분산된 프로세스 객체는 각기 다른 작업을 병행으로 수행할 수 있어 다중 처리를 필요로 하는 작업에 적합하다. 데이터 객체를 분산하여 저장함으로 데이터를 유지보수하는데 용이하다.

단점으로는 클라이언트와 서버 사이에서 어떠한 객체를 어느 곳에 위치시켜야 하는가를 결정해야 하는 분할(Partitioning) 문제가 발생한다. 만약 분할이 제대로 이루어지지 않으면 오히려 성능을 저하시키고, 자원을 낭비하는 결과를 초래할 수 있다.

5. 사례 연구

제시된 클라이언트/서버 모형 중 Process-Balanced 클라이언트/서버 모형을 이용해 실제 프로세스 객체가 어떻게 분산되는지를 연구해본다. 본 사례 연구는 Java RMI(Remote Method Invocation)을 이용해 두개의 서버 호스트로부터 원하는 문자열을 애플릿에 출력하기 위해 필요로 하는 객체의 함수를 호출하여 결과값을 반환 받는다[4]. 이때 완전한 다중 서버를 구현할 수 없기 때문에 우회적인 방법으로 다중 서버를 구현한다.

```
public interface Hello extends java.rmi.Remote {
    String sayHello() throws java.rmi.RemoteException,
    String sendData() throws java.rmi.RemoteExcepuon.
}
```

(예제 1) Interface 코드

위의 (예제 1)은 원격 객체(Remote Object)의 함수를 나타내기 위한 인터페이스 코드로 sayHello()와 sendData()를 다음에 나올 서버 코드인 HelloImpl에서 구현한다.

(예제 2)는 lucifer라는 호스트 이름을 가지고 있는 서버에 위치한 코드로서 (예제 1)에서 선언한 원격 함수를 정의하고 있고 또한 다른 서버(서버 호스트 이름 : casper)에 위치한 프로세스 객체의 함수를 호출하고 있다. 서버 casper의 함수 sayHello2()를 호출하여 그 결과값을 sendData()의 반환값으로 클라이언트 애플릿에서 간접적으로 받을 수 있다.

```
public class HelloImpl extends UnicastRemoteObject
                implements Hello
{
  private String name = "";
  static String message = "";
  public String sayHello() throws RemoteException {
    return "Server 1 World ";
  }
  public String sendData() throws RemoteException {
    return message;
  }
  public static void main(String args[])
  {
  System.setSecurityManager(new RMISecurityManager());
  try {
    HelloImpl obj = new HelloImpl("HelloServer");
    Naming.rebind("//lucifer.soongsil.ac.kr/HelloServer",obj);
    System.out.println("HelloServer bound in registry");
  } catch (Exception e) {
    System.out.println("HelloImpl err: " + e.getMessage());
    e.printStackTrace();
  }
  try {
    Hello2 obj2 = (Hello2)Naming.lookup("rmi://casper.soongsil.ac.kr/HelloServer2");
    message = obj2.sayHello2();
  } catch (Exception ex2) {
    System.out.println("HelloImpl2 exception: "
                    + ex2.getMessage());
    ex2.printStackTrace();
  } } }
```

(예제 2) Server Implementation 코드

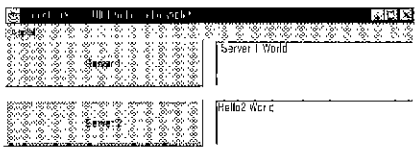
(예제 3)은 서버 lucifer에 위치한 클라이언트 애플릿 코드로 sayHello()와 sendData() 둘다 같은 서버에 위치한 함수이지만 sendData()에서 반환되는 값은 casper라는 서버에서 처리된 결과값을 받은 함수이다.

```
public class HelloApplet extends java.applet.Applet {
  String message = " ";

public boolean action(Event e, Object o)
  {  if (e.target == b1)
     { try {
          message = " ";
          Hello obj = (Hello)Naming.lookup("rmi://lucifer.soongsil.ac.kr/HelloServer");
          message = obj.sayHello();
          t1.setText(message);
       } catch (Exception ex) {
          System.out.println("HelloApplet exception: " + ex.getMessage());
          ex.printStackTrace();
       }
       return true;
     }
  else if (e.target == b2)
  {  try {
          message = " ";
          Hello obj2 = (Hello)Naming.lookup("rmi://lucifer.soongsil.ac.kr/HelloServer");
          message = obj2.sendData();
          t2.setText(message);
       } catch (Exception ex2) {
          System.out.println("HelloApplet2 exception: " + ex2.getMessage());
          ex2.printStackTrace();
       }
     return true;
     }
  return true;
  }
}
```

(예제 3) Client Applet 코드

(그림 8)의 결과 처리 화면은 클라이언트 애플릿에 나타난 화면으로 Server 1(서버 lucifer)의 버튼을 눌렀을 때, 애플릿은 lucifer의 sayHello()를 호출하고 Server 2의 버튼을 눌렀을 때는 서버 lucifer를 통해 서버 casper의 sayHello2()를 호출한다.



(그림 8) Process-Balanced 처리 결과 화면

본 실험을 통해 클라이언트와 서버간에 프로세스의 원활한 분산이 이루어 졌으며, 필요한 경우에만 서버에 종속적인 프로세스 객체를 호출하므로 네트워크 트래픽을 줄일 수 있었고 코드의 재사용성을 높일 수 있었다. 그리고 다중 서버 간에 분산되어있는 프로세스 객체를 완전하지는 않지만 간접적으로 구현됨을 보였다.

위에서 제시된 각각의 모형들이 취할 수 있는 기능을 비교 해본 결과, 프로세스 분산에 가장 적합한 모형은 CORBA의 ORB를 사용한 모형이 가장 효율적임을 보였다. 서버 대 서버의 통신이 ORB를 통해 원하는 서버로부터의 객체를 호출할 수 있으며 다중 서버로부터 동시에 원하는 객체의 함수를 이용할 수 있다. RMI도 부분적이지만 프로세스 분산을 위한 분산 구조에 적합함을 보였다.

6. 결 론

지금까지 절차적 방식의 클라이언트/서버 모형이 지니는 제약사항과 문제점에 대해 살펴보았고, 이를 해결할 수 있는 새로운 인터넷 기반의 객체지향 클라이언트/서버 모형 및 대표적인 시스템 아키텍처들을 제시하였다. 본 논문에서 제시한 클라이언트/서버 모형들은 도메인 특성에 따라 다양하게 적용할 수 있으며, 객체지향 패러다임을 적용함으로써 재사용성을 향상시킬 수 있을 뿐만 아니라 시스템 변경 및 유지보수를 용이하게 한다.

참고문헌

[1] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.

[2] Rosenberry, M., Kenney, D. and Fisher, G., *Understanding DCE*, OReilly & Associates, Inc., 1992.

[3] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice-Hall International, 1991.

[4] Wutka, Mark, *Hacking Java: The Java Professionals Resource Kit*, Que Corporation, 1997.

김 수 동

1984년 미조리 주립대학교, (전산학 학사)
1998년 The University of Iowa, (전산학 석사)
1991년 The University of Iowa, (전산학 박사)
1991년-1993년 한국통신 연구개발단 선임연구원
1994년-현대전자 소프트웨어연구소 책임연구원
1995년-현재 숭실대학교 컴퓨터학부 조교수
관심분야 : 객체지향 개발방법론, 분산형 웹 어플리케이션, 전자상거래 시스템

김 철 진

1996년 경기대학교 전산학과 (학사)
1996년-현재 숭실대학교 전자계산학과 석사과정
관심분야 : 객체지향 사용자 인터페이스, 분산 어플리케이션 설계기법