네트워크 프로그램용 메소드 동적 호출 컴포넌트 개발

Development of a Method Dynamic Invocation Component for Network Program

신봉준* · 정문상** · 홍순구***

* 동아대학교 석사

** 동아대학교 경영정보학과 교수

*** 동아대학교 경영정보학과 교수

요 약

많은 기능을 수행하는 네트워크 프로그램은 그 기능 만큼의 명령어들과 명령어 인자들을 주고 받게 된다. 수신된 명령어에 대한 처리는 "IF~ELSE" 같은 순차적인 비교구문을 사용하거나 자바 RMI같은 원격 메소드 호출방식을 사용하고 있다. 그러나 많은 명령어들을 매번 순차적인 방식으로 비교하는 것과 원격 메소드를 설계하는 방식은 그 구현 및 유지보수에 많은 어려움을 유발하고 있다. 본 논문의 목적은 명령어 수신부와 실행부에 대한 컴포넌트를 개발하여 프로그램 개발 및 유지보수에 들어가는 노력을 줄이고 프로그램의 수행성능을향상시키기 위한 컴포넌트 개발에 있다.

1. 서론

1.1 연구의 목적 및 필요성

1990년대 이후 인터넷의 급속한 보급과 전자상거래의 확산으로 기업의 업무처리는 네트워크에 크게 의존하고 있고 휴대폰을 이용한 모바일 뱅킹 등 네트워킹 기술은 우리의 생활과도 밀접한 관계를 맺고 있다. 이에 따라 현재 컴퓨터에서 사용되는 프로그램들 중 문서 작성 및 특별한 경우를 제외하고는 대부분의 프로그램들이 네트워크 기능을 제공하고 있으며 최근에 출시되는 워드 프로세스 프로그램들도 인터넷이나 네트워크 기능을 기본적으로 제공하고 있는 추세이다. 이처럼 우리가 흔히 사용하는 네트워크 프로그램들은 사용자 편의를 위한 많은 기능들을 포함하고 있다. 이와 같은 사용자 및 시장의 요구에 따라 네트워크 프로그램이 다양한 기능을 제공하기 위해서는 기능 하나하나에 맵핑되는 메소드 및 필드들이 구현되어야만 하기에 그 수 또한 지속적으로 늘어나고 있다. 이렇게 구현되는 코드는 그 수가 많아질수록 개발에 필요한 시간과 노력이 늘어나게 되며 특히 프로그램의 트리거 역할을 담당하는 명령어 수신부와 실행부는 더욱 그러하다. 명령어의 수에 따라 늘어나는 코드의 길이는 프로그램 개발 뿐만 아니라 유지보수, 업그레이드 및 디버깅시에 개발자 및 운영자들을 힘들게 하는 원인이 된다. 특히 예전에는 하나의 프로그램을 개발하고 그에 따르는 업그레이드의 속도가 빠르지 않았지만 현재 프로그램들의 특성은 급변하는 사용자들의 요구에 맞추어 업그레이드가 자주 일어나기 때문에 명령어들의 추가 및 삭제가 빈번하게 발생하고 있다. 이때 기존의 프로그램들은 명령어 비교구문이 있는 명령어 수신부와 실제 실행 메소드가 있는 명령어 실행부에 대한 수정 작업이 이루어 져야 하며 이는 프로그램 실행 중지, 프로그램 수정, 프로그램 컴파일, 프로그램 실행이라는 절차를 거쳐야 한다. 중단 없는 서비스를 제공해야 하는 서버 프로그램의 입장에서 프로그램 수정 시 마다 이런 절차를 거쳐야 한다면 업무수행에 지장을 초래하게 된다.

본 논문은 현재 일반적으로 구현되고 있는 IF~ELSE" 비교구문이나 "SWITCH~CASE" 비교구문 방식을 사용하여 수신된 명령어를 비교구문의 처음부터 시작하여 끝까지 해당 명령어와 같은 구문이 있는지를 검사하여 그에 따른 실행 메소드를 호출하는 방식이 아닌 수신된 명령어 인자를 바이너리 서치를 통해서 실행 메소드를 동적으로 호출하는 컴포넌트를 개발하는 것이다. 기존의 방식은 처음 프로그램 개발 시 명령어 인자의 수만큼 비교구문을 작성하고 프로그램 수정 시, 명령어를 수신하여 비교를 하는 명령어 비교구문과 비즈니스 로직을 구현하여 실제 실행 메소드를 포함하는 명령어 실행부, 두 부분에 대한 수정을 거쳐 다시 컴파일을 해야 한다. 기존의 방식에 비해 프로그

램 개발 시 리플렉션을 사용하는 명령어 수신부를 작성하고 이후 프로그램을 수정할 때는 변동된 비즈니스 로직을 반영할 명령어 실행부만 고쳐주면 된다. 이는 프로그램 코드에서 호출될 메소드 코드를 명시적으로 코딩을 하지 않고 프로그램 실행 시 늦은 바인딩(late binding)을 통한 동적 호출을 구현함으로써 비교부분과 실행부분, 두 부분에 대한 수정을 실행부분으로만 한정 할 수 있게 된다.

1.2 연구의 범위 및 방법

본 연구에서는 네트워크 프로그램 중 프로그램 특성상 많은 명령어들을 처리해야 하는 서버 사이드 및 분산 프로그램을 연구대상으로 한다. 개발 프로그램 언어로는 동적 바인딩과 리플렉션(reflection) 기법이 사용 가능한 자바를 사용해서 바이너리 서치와 리플렉션을 이용하여 명령어 인자에 상응하는 메소드를 동적으로 호출하는 컴포넌트와 커맨드 패턴의 개발을 목적으로 한다. 컴포넌트 개발 후 이의 활용 가능성을 시험하기 위해 기 사용되고 있는 네트워크 프로그램과 새로운 컴포넌트를 추가한 프로그램의 수행결과를 비교분석 한다.

본 논문은 2장에서 네트워크 프로그램의 개념 및 기존 네트워크 프로그램에서 사용하고 있는 명령어 처리방식 및 한계점에 대해서 알아본다. 3장에서는 컴포넌트 개발을 위한 기본이 되는 기술들에 대해서 알아보고 그 구현에 대해서 서술한다. 4장에서는 기존의 네트워크 프로그램과 새로운 컴포넌트의 수행성능과 프로그램 유지보수의 용이성에 대해서 비교한다. 5장에서는 본 연구의 활용성과 적용분야에 대해서 알아보고 한계점에 대해 논의한다.

2. 기존 프로그램의 한계점

네트워크 프로그램은 둘 이상의 컴퓨터들이 통신을 한다는 것을 전제로 하고 있다. 각각의 컴퓨터들이 서비스 요청 혹은 서비스 제공의 입장에서 서로간에 규약(protocol)을 정해서 정해진 통신을 하게 된다. 이때 서비스 제공자의 입장에서 수신된 명령어가 규약에 있는 명령어인지를 검사를 하고 존재한다면 그것에 맞는 서비스를 제공하게 된다. 명령어 검사의 방식으로 "IF~ELSE" 비교구문, "SWITCH~CASE"비교구문이 있으며, 다른 방식으로는 RMI(Remote Method Invocation), RPC(Remote Procedure Call)방식 등이 있다.

2.1 기본 "if ~ else ~ "구문 및 "switch case" 구문

주위에서 많이 사용되고 있는 서비스 중 Command 라인 형식으로 제공되는 기본적인 FTP 서비스에서 필수 명령어 인자는 42개이다. 여기에 GUI형식으로 좀더 부가적인 기능을 추가한다면 명령어 인자의 수는 비례해서 증가하게 된다.

● 비교부

비교부는 수신되는 명령어 인자와 파라미터가 정의되어 있는 명령어가 있는지, 있다면 그에 따른 작업을 수행하게 하는 일차적인 부분이다. 기존의 네트워크 프로그램에서 수신되는 명령어 인자와 파라미터를 비교하기 위해서는 대부분 "if ~ else"구문을 사용하거나 비슷한 역할을 하는 "switch ~ case"구문을 사용하게 된다. 앞에서 예로든 FTP 프로그램에서 들어오는 명령어 인자를 비교하기 위해서는 최소 42개의 "if ~ else"구문이나 "switch case" 구문을 필요로 하게 된다. 하나의 명령어 인자에 대해서 42개에 달하는 순차적인 비교검색은 명령어 인자가 상위에 있는 비교구문에서 일치가 되면 문제가 되지 않지만 하위에 있는 명령어에 대해서는 순차적으로 모두 비교를 해야 되는 비효율적인 작업이 되며 비교작업이 계속 될수록 수행성능에 영향을 주게 된다. 여러 클라이언트들이 접속하고 계속적인 명령어를 주고받아야 하는 상황에서 비교부에서 일어나는 지속적인 병목현상은 프로그램 전체적인 속도를 떨어지게 하는 요인이 된다.

명령어와 파라미터가 그렇게 많이 사용되지 않는 규모가 작은 프로그램에서는 명령어와 파라미터에 대해서 하나씩 비교하는 방식이 효율적일 수도 있으나 많은 명령어를 처리해야 하는 서버 사이드 프로그램에서는 명령어가 들어올 때 마다 순차적인 비교방식을 사용하는 것은 비효율적이다. 특히 기존 네트워크 프로그램 개발 시 자주 사용될 것으로 예상되는 명령어와 파라미터들에 대해서는 비교구문의 상위에 위치하게 해서 가능한 한 비교검색의 속도를 높이려는 시도가 많았지만 이와 같은 시도도 계속되는 프로그램의 유지보수와 기능의 추가에서 프로그램 코드 수정 시 코드의 변화 및 가독성의 문제를 더 복잡하게 만드는 원인이 되고 있다.

● 호출부

호출부는 명령어 인자와 파라미터에 맵핑되는 메소드를 호출하는 부분이다. 구조적 방법론에서 CBD(Component Based Development)까지 대부분의 프로그램 개발에서 특정 함수나 객체의 메소드를 구현하고 구현한 메소드를 사용하기 위해서는 호출부에서 명시적으로 함수나 메소드의 이름을 지정해야만 한다. 이와 같은 접근법은 명령어 인자와 파라미터를 사용하지 않는 다른 프로그램에서는 문제가 되지 않지만 많은 명령어 인자와 파라미터를 비교하고 그에 따른 메소드를 호출해야 하는 네트워크 프로그램에서는 그 수만큼 메소드 호출코드를 작성해야 하는 문제가 발생한다.

● 실행부

실행부는 클라이언트가 요구한 서비스를 수행하는 부분이다. 네트워크 프로그램에서 서비스를 수행하는 실질적인 코드인 수행부는 비교검색이 이루어지는 코드 내 혹은 다른 객체에서의 메소드로 구현이 된다. 구현된 메소드를 이용하기 위해서는 명시적인 호출이 이루어져야만 메소드 실행이 가능하다. 이와 같이 단순히 하나의 명령어 인자와 파라미터를 바꾸기 위해서도 프로그램 코드의 호출부와 실행부 등 여러 부분을 고려하여 프로그램 코드를 작성해야 하는 관계로 여러 부분으로 분산된 코드에 대해서 하나의 잘못된 부분을 찾기 위해서는 많은 시간과 자원이 소모되는 원인이 된다. 특히 많은 프로그래머들이 고민하는 부분들이 프로그램 코드에서 사용될 변수와 메소드 이름의 선정에 많은 시간을 소비하는 것이 현실이다. 특정 프로그램 언어에서는 대소문자를 구분해서 개발자들에게 이와 같은 고민을 조금은 덜어주고 있지만 명령어 인자와 파라미터를 정의하고 그것에 맞는 변수 및 메소드들의 이름을 정의하는 것은 아직도 개발자들에게 많은 시간을 소비하게 하는 원인이다. 이것은 초기 개발 시에 국한되지 않고 프로그램의 코드가 늘어날수록 개발자들이 잘못 선언된 변수명과 메소드명을 찾기 위해서 모니터 앞에서 시간을 보내는 근본적인 원인이 된다 [Horton,2001,Adatia,2002].

2.2 자바 RMI

자바 RMI(Remote Method Invocation)는 JRMP(Java Remote Method Protocol)에 의해 동작한다. 자바는 자바 객체의 직렬화에 크게 의존하기 때문에, 객체를 일종의 스트림(stream)으로 마샬링(marshaling)한다. 이러한 자바객체의 직렬화(Java Object Serialization)는 자바의 특징가운데 하나이기 때문에, 자바 RMI 서버 객체와 클라이언트객체가 모두 자바일 경우에만 사용할 수 있다. 각각의 자바 RMI 서버 객체는 현재 사용하고 있는 JVM(Java Virtual Machine)의 외부에서 서버 객체에 접근할 수 있는 인터페이스를 정의한다. 이렇게 정의된 인터페이스를 통해 서버 객체에 의해 제공되는 서비스 메소드가 결정된다. 클라이언트가 서버 객체에 처음 접근할 때 RMI는 서버 기계에서 작동하는 RMIRegistry 라는 일종의 명명 메커니즘(naming mechanism)을 이용하는데, 이를 통해 현재 사용 가능한 서버 객체의 정보를 얻을 수 있다. 자바 RMI클라이언트가 자바 RMI 서버 객체에 대한 객체 레퍼런스를 획득한 뒤에는 마치 같은 클라이언트 어드레스 공간에서 메소드를 호출하는 방법과 같은 방식으로 서버 객체의 메소드를 이용할 수 있다.

RMI는 구조상 두 개의 자바 클래스, 인터페이스와 인터페이스를 구현한 클래스를 필요로 한다. RMI를 작성하려면 먼저 인터페이스를 작성한 후 인터페이스에 대한 구현 클래스를 작성한다. 그리고 RMI 컴파일러(rmic)를 통해 인터페이스를 구현한 클래스를 컴파일한다. 그러면 스텁(stub, 클라이언트에 다운로드한 클래스), 스켈레톤(skeleton, 서버에서 클라이언트의 요청에 대해 처리하는 클래스)이 만들어진다. 즉, 스텁과 스켈레톤은 클라이언트와 RMI 구현 객체 사이에 중계 역할을 한다. 그 후 RMI 객체를 RMI 레지스트리(RMI Registry)에 등록해 클라이언트의 요청에 대해 수행할 수 있다. 클라이언트에서는 위에서의 어떤 과정이 벌어지는지 알 필요 없이 인터페이스의 메소드를 호출한다.

이렇듯 자바는 RMI라는 자체 ORB(Object Request Broker)를 가지고 네트워크 프로그램작성에 다른 가능성을 제공하지만 초기 RMI는 자바 플랫폼 환경하에서만 사용해야 하는 제약을 가지고 있다. RMI는 자바언어의 자체적인 특성에 많이 의존하기 때문에 자바 객체 직렬화를 구현하여야만 한다. 이후 썬(SUN)에서 JDK1.2를 발표하면서 CORBA의 IIOP를 표준 프로토콜로 지원을 하였다. 개발자는 자바의 RMI 객체를 다른 언어에서 작성한 분산객체와의 통신도 가능하게 되었다. 현재 다른 언어를 지원하는 분산객체기술이 연구

되고 있으나 자바를 응용한 제품들이 가장 경쟁력이 있으며 널리 받아들여지고 있다[Lange,Oshima,1998].

네트워크 프로그램 작성시 여러 편의성을 제공하는 자바 RMI이지만 실제 네트워크에 배치한다고 가정하면, 인터페이스를 설계하는 것에서부터 시작하여, 인터페이스 구현객체 소스코드 작성, 컴파일, RMI컴파일러(rmic)를 이용해 스텁(stub), 스켈레톤(skeleton)객체 생성, 해당되는 객체가 클라이언트에서 접근이 가능하도록 classpath 관리 및 배치, 해당 원격객체의 system naming registry에의 등록, 서버측 애플리케이션 시작, 클라이언트측 애플리케이션 시작 등 상당히 복잡한 과정을 거치게 된다[서건원,2002].

또한 현재의 RMI는 누구나 확장할 수 있게 돼 있기보다는 아직은 블랙박스에 가깝다고 할 수 있다. RMIC로 생성하는 스텁에서 내부적으로 사용하는 부분을 빼고 나면 상대적으로 고수준의 API만을 제고 하고 있을 뿐이다. 문제는 프로토콜 자체를 TCP 소켓이 기반하지 않는 다른 것으로 대체하고 싶은 경우에는 다른 방법이 없다. 또한 RMI에 기반한 솔루션을 개발할 때 항상 부딪히는 문제인 인증에 관련된 것이다. 즉 원격 객체에서 그에 접속하는 기계가 아닌 사용자에 대한 인증에 따라 상대가 누구인지 알고 대응해야 하는데 아직 이런 문제에 대해서 충분한 해결모델을 제시하지 못하고 있다[김도형,2000,차상현,1998].

위의 표에서 보듯이 기존의 네트워크 프로그램들은 저마다의 한계점을 가지고 있다. 비교구문을 통한 프로그램 작성은 직관적인 코드작성이 가능하지만 네트워크 명령어 인자에 수에 비례하여 코드 가독성이 떨어지게 된다. RMI를 통한 방법은 클라이언트가 필요한 기능을 로컬에 있는 메소드처럼 호출할 수 있는 장점이 있지만 자바 플랫폼에 종속적이고 설계 및 유지보수에 많은 어려움이 따른다.

3. 바이너리 서치를 통한 메소드 동적 호출 컴포넌트 개발

3.1 관련기술

3.1.1 바이너리 서치(Binary Search) 트리 알고리즘

바이너리 서치 트리는 삽입, 삭제 그리고 탐색 연산을 지원하는 자료 구조로 잘 알려져 있다. 바이너리 서치 트리에서 왼쪽 자식 노드의 값은 부모 노드의 값보다 작고 오른쪽 자식 노드의 값은 부모 노드의 값보다 크다.

바이너리 서치 트리의 탐색의 연산은 매우 간단하다. 특정 값 v를 찾을 경우, 바이너리 서치 트리의 루트 노드에서 시작하여 v가 노드의 값보다 작으면 그 노드의 왼쪽 자식 노드로 이동하고 노드의 값보다 크면 오른쪽 자식 노드로 이동하면 된다. 이 과정을 v를 찾을 때까지 반복하면 된다[Horowitz,1993].

3.1.2 커맨드 패턴(Command Pattern)

커맨트 패턴은 요청을 객체로 표현하여 클라이언트가 요청을 파라미터화 할 수 있도록 해준다. 각각의 명령어를 객체로 표현하기 때문에 신규

[표] 기존 네트워크 프로그램 특징 및 한계점

구분	If~else / switch~case	RMI
비교	순차적인 방식으로 비교구문의 처음부터 끝까지 조건이 일치하는 것이 나올 때까지 비교를 수행한다.	비교가 필요없음.
호출	비교구문안에서 명시적으로 함수나 메소드의 이름을 지정해야만 한다.	클라이언트가 필요한 기능의 메소드를 로컬에서 호출하듯 서버의 메소드를 호출한다.
실행	실제 작업을 수행하는 실행구문을 서버에 작성한다.	실제 작업을 실행하는 구문을 서버에 작성한다.
설계 및 유지보수	명령어 수만큼 비교구문을 작성하고 비교구문안에 메소드 호출코드를 삽입한다. 명령어가 많아질수록 코드의 가독성이 떨어지게 된다.	인터페이스를 설계, 인터페이스 구현객체 소스코드 작성, 컴파일하고 그리고 RMI컴파일러(rmic)를 이용해 스텁(stub), 스켈레톤(skeleton)객체 생성, 해당되는 객체가 클라이언트에서 접근이 가능하도록 classpath 관리 및 배치, 해당 원격객체의 system naming registry 에의 등록, 서버측 애플리케이션 시작, 클라이언트측 애플리케이션 시작 등 상당히 복잡한 과정을 가진다.

명령어에 대한 행위는 다른 명령어들에 영향을 미치지 않고 쉽게 신규 객체를 추가 할 수 있다. 또한 기존 명령어의 행위에 대한 수정이나 추가가 필요한 경우에도 다른 명령어에 아무런 영향을 주지 않고 쉽게 수정 및 추가를 할 수 있다. 커맨드 패턴은 '메소드 호출'이라는 개념 자체를 객체에 담아 메소드 호출이라는 개념을 객체로 다룰 수 있게 해준다[Adatia,2002]. 따라서 커맨드 패턴을 이용하면 메소드 호출 자체를 다른 메소드로 전달하고 어딘가에 보관할 수 있으며 다형성을 이용하여 서로 다른 메소드 호출을 하나의 묶음으로 처리할 수 있게 해준다.

3.1.3 리플렉션(Reflection)

Java 언어의 중요한 기능으로 리플렉션("introspection"이라고도 함)을 들 수 있다. 리플렉션은 속성과 관련하여 Java 클래스를 질의하고, 주어진 오브젝트 인스턴스의 이름에 따라 여러 메소드들과 필드들을 사용할 수 있게 한다[자바 개발자 포럼]. 원래의 객체지향 언어 개념(OOP, Smalltalk)에서 Class는 메모리에 인스턴스화 되지 않는 개념상의 정보일 뿐이다. 그러나 Java는 Class정보 자체를 인스턴스화 하여 메모리에 적재한 뒤, 그 메타 정보를 실시간으로 이용할 수 있게 한다. 이러한 메타정보 시스템은 C#의 manifest에 의해 더 발전하게 된다. 클래스 정보 자체를 정의한 자바클래스가 바로 java.lang.Class 이다. java.lang.Class 의 인스턴스는 ClassLoader라는 또다른 클래스 인스턴스에 의해 생성된다. ClassLoader는 classpath라는 특정 위치 배열에 매핑되어 그 위치에 존재하는 클래스 정의 파일들(*.class)을 읽어서 메모리에 동적으로 적재하게된다. *.class 파일에 들어있는 내용은 클래스 정의인데, 특별한 포맷이 있다. *.class가 반드시 파일일 필요는 없으며 network을 통해 받는 byte의 연속(stream)이어도 되고 메모리상에서 즉석으로 만들어낸 byte 배열이여도 된다. 자바는 클래스정보를 인스턴스화하여 가질 수 있어 리플렉션(reflection)이라는 개념이 가능하게 되었다.

본 논문에서 제시하는 리플렉션을 사용하면 초기에 정의하는 명령어 인자와 파라미터의 이름을 그대로 변수와 메소드명으로 사용할 수 있기에 초기 변수 및 메소드 네이밍에 소비되는 부차적인 개발시간을 줄일 수 있으며 좀 더 직관적으로 코드를 구현할 수 있게 된다.

3.2 구현

앞에서 밝혔듯이 기존의 네트워크 프로그램은 들어오는 명령어 인자에 대한 실행 메소드 호출부분이 하드코딩형식으로 만들어져 명령어들에 대한 추가 및 삭제, 프로그램 수정 시 명령어 비교부분에 대한 수정, 실행 메소드 호출부분에 대한 수정, 마지막으로 실제 실행되는 메소드에 대한 수정이 이루어졌다. 이렇게 여러 부분에 대한 수정작업은 프로그램 수행 시 예상치 못한 오류를 발생시킬 가능성이 커지게 된다. 이런 이유로 명령어를 수신하고 명령어에 맵핑되는 메소드 호출부분을 하나의 컴포넌트로 만들고 실행 메소드들만 모아 하나의 컴포넌트로 만들어 둘 사이의 관계를 느슨하게 한다면 명령어 추가/삭제 및 프로그램 수정 시 실행 메소드 부분만 수정을 하여 바로 사용할 수 있어 여러 부분에 거쳐 수정을 하는 작업보다 효율적으로 작업을 완수할 수 있으며 직관적인 수정이 가능해진다. 특히 비교구문의 처음부터 끝까지 순차적으로 비교를 수행하는 방식이 아닌 바이너리 서치를 통한 빠른 메소드 검색은 프로그램 수행성능을 향상 시킨다.

3.2.1 비교호출부

기존 프로그램에서 명령어 인자와 파라미터의 수만큼 비교구문을 작성하고 그에 따른 메소들 호출구문을 작성하던 방식은 프로그램이 지원하는 기능이 많아질수록 프로그램 코드가 비례하여 늘어났지만 리플렉션을 사용하면 명령어 인자와 파라미터의 수에 관계없이 두 줄로 유지될 수 있다. 특히 명령어비교가 많을수록 순차적인 방식으로 비교를 하는 기존 프로그램 보다 명령어에 맵핑된 메소드를 검색하는 시간이 빨라지게 된다.

서버 프로그램이 실행될 때 실행 메소드들을 포함하는 클래스의 메소드들을 메소드 이름과 메소드 객체로 나누어서 배열에 할당한다. 실행부를 포함하는 클래스의 메소드들은 작성될 때 무작위로 구현이 된다. 자바의 바이너리 서치 기능을 사용하기 위해서는 바이너리 서치에 사용될 배열의 내용이 우선 정렬된 상태여야만 가능하다[강규영,2003]. 이를 위해서 메소드 이름을 포함하는 배열을 메소드명으로 오름차순하여 다시 할당한다. 이후 메소드 객체를 포함하는 배열을 메소드 이름으로 정렬된 배열의 순서에 맞게 다시 할당한다.

명령어 인자 수신부는 들어온 명령어 인자를 커맨드 패턴에 맞게 가공한다. Arrays 클래스의 정

적 메소드 바이너리 서치 메소드는 2개의 인자(메소드 이름 배열,명령어 인자)를 비교하여 명령어 인자와 일치하는 메소드명이 있으면 배열의 인덱스를 반환한다. 만약 일치하는 메소드명이 없으면 -1을 반환하게 된다. 반환된 인덱스 값으로 메소드 객체 배열의 메소드를 호출하게 된다.

3.2.2 실행부

기존의 프로그램이 비교검색이 이루어지는 코드 내에 실행 메소드를 구현하지 않고 다른 클래스에 메소드를 구현하였다면 별다른 변환없이 메소드 각각에 대해서 예외처리(Throws Exception)를 해주면 실행부 컴포넌트로 바로 사용할 수 있다. 실행 메소드만으로 구성된 클래스는 개발자가 다른 부수적인 코드에 신경써야하는 것을 줄이게 만들고 프로세스 로직에 좀더 집중할 수 있도록 도와줄 수 있다.

4. 개발된 컴포넌트 평가

본 논문에서 제안한 메소드 동적 호출 컴포넌트는 프로그램 수정 및 유지보수라는 정성적 측면과 프로그램 수행성능이라는 정량적 측면에서 평가를 할 수 있다.

정성적 측면에서 본 논문의 컴포넌트를 사용하면 외부 인터페이스를 통해서 바이너리 서치 명령어 수신부를 사용할 수 있다. 명령어 수신부는 프로그램이 수행해야 하는 명령어에 비례하여 비교구문이 늘어나는 기존의 코드에 비해 단 두 줄의 명령어 비교 및 호출코드로 프로그램 수행이 가능하다. 비교 구문안에 명시적으로 메소드호출을 지정해야 하는 작업은 프로그램 코드 작성을 어렵게 하며 프로그램 에러의 원인이 되기도 한다. 특히 자주 호출되는 명령어 인자에 대해서는 프로그램 로그 분석을 통해서 비교구문의 상위에 위치를 시켜야 하는 방식에 비해서 프로그래머가 신경을 쓸 부분이 없어졌다. 이를 통해 프로그램 개발자 및 분석가는 프로세스 로직에 좀 더 집중을 할 수 있으며 코드 분할에 따른 기능 추가 및 디버깅이 좀더 수월해진다. 그리고 프로그램 수정 시 기존의 프로그램들과는 달리 프로그램의 중지 없이 명령어 실행부만 수정을 하여 바로 사용할 수 있으며 명령어들의 추가 및 삭제가 이루어져도 명령어 수신부의 수정 없이 명령어 실행부만 바꾸어 주면 된다. 이는 갈수록 짧아지는 프로그램의 수명주기 및 다양한 시장의 요구에 신속하게 대응할 수 있는 방편이 되며 순차적인 비교방식에 비해서 바이너리 서치를 통한 방식이 프로그램의 수행성능을 향상시키게 된다.

바이너리 서치와 리플렉션 기능을 사용한 동적 메소드 호출 컴포넌트는 프로그램 수행시간의 단축 및 프로그램 코드의 간략화, 분할을 통한 직관적인 프로그램 작성이 가능해진다. 이는 실시간으로 많은 통신을 하는 서버 사이드 프로그램 뿐만 아니라 기능이 다양해 지고 있는 채팅, VOIP의 소프트스위치 엔진 및 협상 에이전트 같은 통신 프로그램에 적용이 가능하다. 특히 협상 에이전트는 그 특성상 네트워크의 부하를 줄이는 통신을 하면서 다양한 작업을 수행하게 되는데 이때 에이전트에 대한 기능의 추가 및 삭제 시 전체적인 프로그램의 다운로드를 통해서 프로그램을 수정하는 방식을 사용하고 있다. 이런 방식은 전체적인 프로그램의 수정뿐만 아니라 사소한 코드의 수정 시에도 같은 방식이 이루어지며 비효율적인 방법이라고 할 수 있다. 본 논문의 컴포넌트는 프로그램의 비즈니스 로직이 있는 명령어 실행부가 분리되어 있어 수정이 필요한 부분만 다운로드하여 바로 사용할 수 있어 네트워크 활용율을 높일 수 있으며 프로그램의 중지 없이 계속적인 실행이 가능하다. 이는 비단 에이전트 프로그램에만 국한되지 않고 잦은 업데이트를 수행하는 네트워크 프로그램이나 서버/클라이언트 프로그램에서 활용이 가능하다.

이상과 같은 정성적 부분에 대한 비교를 표로 정리하면 다음과 같다.

[표] 기존프로그램과 동적호출의 정성적 비교

비교	IF~ELSE	RMI	동적 메소드 호출
명령어 수신부 수정여부	○	○	×
명령어 수신부 컴파일여부	○	○	×
명령어 호출부 수정여부	○	○	×
명령어 호출부 컴파일여부	○	○	×
명령어 실행부 수정여부	○	○	○
명령어 실행부 컴파일여부	○	○	○

새로운 컴포넌트의 성능을 분석하기 위해서 더미프로그램을 개발하였다. 더미프로그램 서버는 총

세 부분으로 이루어지는데 바이너리 서치와 리플렉션 코드를 포함하는 명령어 비교호출부(TestCMD.java), 명령어와 매핑된 실행 메소드를 포함하는 명령어 실행부(TestCMDArgs.java), 그리고 명령어 수신부 및 호출부를 실행시키는 main 함수를 포함하는 프로그램 시작부분(TestCom.java)으로 구성 된다. 클라이언트측 더미 프로그램은 쓰레드 형식으로 서버측에 명령어를 전송하는 명령 전달부(TestThread.java)와 스레드를 동시에 몇 개씩 실행시킬지를 결정하는 프로그램 시작부분(TestClient.java)으로 구성된다. 클라이언트는 쓰레드 시작시 "STRT"라는 명령어를 전송한후 10000번의 루프를 돌면서 50개의 메소드 호출 명령어 중 하나를 전송하게 된다. 10000번의 전송이 끝나면 쓰레드 종료를 알리는 "ENDS"라는 명령어를 전송하고 프로그램을 종료하게 된다. 서버 프로그램은 "STRT"라는 명령어를 수신하였을 때의 시간을 저장하고 클라이언트가 전송한 명령어에 해당하는 메소드 호출을 반복하면서 "ENDS"라는 명령어가 수신되었을 때 의 시간차를 계산해서 출력하게 된다.

다음과 같은 실험환경에서 프로그램을 실험하였다.

[표] 실험환경

구분	OS	CPU	Memory	Complier
더미 서버	Window 2003	PIV-2.53 GHz	1G	JDK 1.4.2
더미 클라이언트	Window XP	PIV-2.4G Hz	512M	JDK 1.4.2

[표] 순차적 메소드 호출에 따른 처리시간

단위 : msecs

시도 횟수	기존 IF~ELSE에 의한 호출	동적 호출
1	750	735
2	750	750
3	781	750
4	750	797
5	734	750
6	797	750
7	750	734
8	750	750
9	750	750
10	750	812
11	843	750
12	781	781
합계	9186	9109
평균	765.5	759.08

앞에서 살펴본 GUI형식이 아닌 Command 라인 형식의 FTP 프로그램의 기본 명령어 인자가 42개이고 서버프로그램에 비해서 기능이 맞지 않은 리눅스에서 사용되고 있는 채팅프로그램 XChat 1.6.3의 프로그램의 명령어 인자가 41개인 것을 감안하여 더미 서버에서 수행되는 명령어 인자를 50개로 설정을 하였다.

[표] 마지막 메소드 호출에 따른 처리시간

단위 : msecs

시도 횟수	기존 IF~ELSE에 의한 호출	동적 호출
1	812	750
2	860	797
3	765	828
4	766	765
5	766	750
6	766	750
7	875	766
8	750	750
9	875	766
10	766	766
11	859	750
12	890	797
합계	9750	9235
평균	812.5	769.58

[표] 순차적 메소드 호출과 마지막 메소드 호출시의 처리시간차

단위 : msecs

평균	기존 IF~ELSE에 의한 호출	동적 호출	차이
순차적 메소드 호출	765.5	759.08	6.42
마지막 메소드 호출	812.5	769.58	42.92
차이	47	10.5	36.5

순차적인 메소드호출시에 평균적으로 "IF~ELSE" 비교구문 방식이 765.5(MS)가 소모되었고 동적 메소드 호출 방식이 759.08(MS)가 소모되었다. 이는 일반적인 비교구문 방식에 비해 동적 메소드 호출 방식이 근소하게 성능이 높다는 것을 보여주고 있다. 하지만 마지막 메소드에 대한 호출시간은 기존 비교구문 방식이 812.5(MS)이고 동적 메소드 호출 방식이 769.58(MS)로 수신된 명령어와 명령어 인자를 "IF~ELSE" 비교구문의 처음부터 끝까지 비교를 해야 하는 기존방식의 시간보다 더 빠른 수행능력을 보여주고 있다. 특히 순차적 메소드 호출과 마

지막 메소드 호출에 대한 "IF~ELSE" 비교구문의 시간차이는 47(MS)나 나오는데 반해 동적 메소드 호출방식은 10.5(MS)가 나와 프로그램 수행성능에 대한 편차가 심하지 않다는 것을 확인할 수 있다.

5. 결론

소프트웨어 산업의 규모는 급격한 팽창을 거듭하고 있다. 이에 따르는 소프트웨어 개발 기술의 발전 속도는 하드에어 발전 속도에 비해 뒤떨어지는 실정이다. 많은 소프트웨어 프로젝트들은 개발 기간의 지연, 예산 초과, 사용자가 원하는 고품질의 소프트웨어를 생산하지 못함으로 인해 실패하는 경우가 많다[정원호,2003].

본 논문에서는 네트워크 프로그램의 특성상 많은 명령어들과 명령어 인자들을 주고 받는 명령어 수신부와 실행부에 대한 컴포넌트를 개발하였다. 기존 프로그램은 명령어 추가 및 삭제 시 "IF~ELSE" 구문에 추가적인 명령어 비교구문을 수정해야 하고 비교구문안에 호출될 메소드 이름을 추가해야 한다. 마지막으로 실제 메소드 부분에 대한 구현을 하는 것이 기존의 방식인데 반해 동적 메소드 호출방식은 비교호출부에는 어떠한 수정작업 없이 실행 메소드 부분에 대해서만 수정을 하면 된다. 이것은 기존의 방식은 프로그램 수정 시 명령어 수신부, 메소드 호출부, 명령어 실행부에 대한 적어도 3부분 이상에 대한 작업을 하고 그것을 컴파일하는 작업을 필요로 하지만 동적 메소드 호출은 명령어 실행부 하나에 대해서만 수정을 하고 컴파일 작업을 필요로 한다. 실행되는 프로그램 코드의 양이 많아 질수록 기존의 방식은 비효율적인 작업의 양이 많아지는 반면 필요한 코드만 수정을 필요로 하는 컴포넌트 방식은 프로그램 개발자나 시스템 분석가에게 핵심업무에 집중할 수 있도록 도움을 준다. 그리고 프로그램 수행성능은 명령어에 대한 순차적인 비교방식이 아닌 바이너리 서치를 사용하여 명령어에 해당하는 메소드를 바로 검색하여 호출할 수 있어 비교구문의 처음부터 끝까지 순차적으로 비교하는 방식보다 더 빠른 수행시간을 보여준다.

본 논문의 한계점으로는 바이너리 서치 기능을 자바가 기본으로 지원하는 방식을 그대로 사용하고 있어 검색에 있어 최적화 되어 있다고 말하기 힘들다. 검색에 대한 좀더 빠른 수행을 할 수 있는 로직 개발이 필요하다. 그리고 리플렉션 기능은 프로그램 실행시 약간의 오버헤드가 존재하게 된다. 그래서 명령어 수에 따른 검색시간과 리플렉션의 오버헤드의 트레이드 오프가 어디까지 되는지에 대한 연구가 향후 과제로 남겨져 있다.

참고문헌

1) 김도형, 자바의 현재와 미래를 가늠하다, 마이크로 소프트웨어, 201-207 (2000)
2) 강규영, 자바 리플렉션에 대한 고찰, 마이크로 소프트웨어, 233, 215-221 (2003)
3) 서건원, 에이전트를 이용한 분산컴퓨팅 환경 구현, 서울産業大學校 (2002)
4) 정원호, 분산 응용을 위한 이동 에이전트 프로그래밍 시스템, 자연과학 논문집, Vol.9 (2003)
5) 차상현, 3계층 클라이언트/서버 구조에서 코바/자바 ORBs와 HTTP/CGI의 성능 비교, 산업기술논문집, Vol.11 No.2 (2000)
6) A.W.Brown and K.C Wallnau, The Current State of CBSE, IEEE Software Sept/Oct, 37-46 (1998)
7) Chad Darby, Java Networking, worx, 75-92 (2002)
8) Clemens Szyperski, Component Software, Addison-Wesley (1998)
9) Danny B. Lange and mitsuru Oshima, Programming and Deploying Java Mobile Agent with Aglets, Addison Wesley (1998)
10) Eliss Horowitz, Sartaj Sahni, and Susan Anderson-Freed, Fundamental of Data Structure in C, Computer Science Press (1993)
11) Iver Horton, Beginning JAVA2, wrox (2001)
12) Rahim Adatia, professional EJB, wrox, 891-894 (2002)
13) Wojtek Kozaczynaki, Composite Nature of Component, ICSE99, 73-77 (1999)