제33회 한국정보처리학회 춘계학술발표대회 논문집 제17권 제1호 (2010. 4)

자바서비스 접근성 향상을 위한 비스텁 방식의 원격자바메서드 호출기법

최우정*, 최재현*, 박제원* 이남용*
*숭실대학교 컴퓨터학과
e-mail:cwj011@nate.com, uniker80@empal.com, jwpark5656@hotmail.com
nylee@ssu.ac.kr

A Method of Invocation of Java Services for High Accessibility by not Using Static Stub (JRSC: Java Remote Service Call)

Woo-Jeong Choi*, Jae-hyun Choi*, Jae-Won Park*, Nam-Yong Lee*
*Dept of Computer Science and Engineering, Soong-sil University

요 약

최근 분산자바기술의 발전에 따라 다양한 애플리케이션의 단위기능들은 어디서나 호출 가능한 서비스로 정의되고 있으며, 또한 이러한 서비스들은 새로운 애플리케이션을 구성하기 위한 단위기능으로 사용되고 있다. 특히 RMI, CORBA 등은 가장 널리 활용되고 있는 원격 메서드 또는 서비스 호출 프레임워크로 분산 자바애플리케이션 개발을 위한 중요한 기술이라 할 수 있다. 그러나 이러한 기술들은 모두 원격메서드 또는 서비스의 호출을 위해 클라이언트사이드에서 스텁을 활용하는 기술로, 스텝의 정적생성 및 동기화 필요성으로 인해 다소 사용자 편의성 및 서비스의 접근성이 저해되는 측면이 있다.

따라서 본 논문에서는 이러한 스텁의 생성이 필요 없는 비스텁 방식의 효율적인 원격자바메서드 호출기법을 제시한다. 이 기법을 활용할 경우, 사용자 측면에서 별도의 스텁 없이 원격지의 서비스를 자유롭게 호출할 수 있어 사용자의 편의성 및 서비스 접근성을 크게 향상시킬 수 있으며, 원격 서비스 변경에 따른 동적 서비스 재구성 또한 지원할 수 있어 보다 유연한 애플리케이션 개발이 가능하다.

1. 서론

최근 고성능 컴퓨터의 보급과 인터넷의 대중화는 분산 컴퓨팅 기술의 발전을 주도하게 되었다. 또한 소프트웨어 개발 기술 역시 절차적인 방식에서 객체지향, 그리고 분산 객체 방식으로 변화하고 있다.[1][2] 특히 인터넷의 활성화는 기존 로컬에서만 제공되었던 서비스들이 위치를 가리지 않고 네트워크만 구성되어 있다면 어디서든지 사용가능한 형태로 진화하고 있다. 이러한 분산 애플리케이션을 개발하기 위해서는 흔히 소켓 프로그래밍을 사용하였다. 하지만 이러한 소켓 프로그래밍은 개발자가 통신 프로토콜에 대한 충분한 이해가 필요로 하며, 통신상에서 발생되는 많은 오버헤드를 직접 관리해야 한다. 이러한 소켓 프로그래밍은 사용자 측면에서 상당한 부담이 될 수 있으며 생산성 또한 저해되는 요인이 된다.[3] 이를 보완하고 사용자들이 보다 쉽게 분산 애플리케이션을 개발하기 위한 연구가 활발히 진행되고 있다. 특히 썬마이크로시스템즈에 의해 개발된 Java의 등장은 개발자들에게 분산 애플리케이션 개발을 위한 보다 낳은 환경을 제공해주었으며, 대표적인 기술로 CORBA와 RMI가 있다. CORBA와 RMI는 가장 널리 활용되고 있는 원격 메서드 또는 서비스 호출 프레임워크로 분산 애플리케이션을 개발하는 중요한 기술이라 할 수 있다. 하지만 이러한 기술들은 모두 원격 메서드 또는 서비스의 호출을 위해서 클라이언트사이드에서 스텁(Stub)의 정적생성 및 동기화 필요성으로 인해 다소 사용자 편의성 및 서비스의 접근성이 저해되는 측면이 있다.

따라서 본 논문에서는 이러한 별도의 스텁 생성이 필요 없는 비스텁 방식의 효율적인 원격자바메서드 호출기법(JRSC:Java Remote Service Call)을 제시한다. 이 기법을 활용할 경우, 사용자 측면에서 별도의 스텁 생성 없이 원격지의 서비스를 자유롭게 호출할 수 있어 사용자의 편의성 및 서비스 접근성을 크게 향상시킬 수 있으며, 원격 서비스 변경에 따른 동적 서비스 재구성 또한 지원할 수 있어 보다 유연한 애플리케이션 개발이 가능할 것으로 기대한다.

2. 자바기반의 분산객체기술

분산객체기술의 핵심은 로컬에 존재하는 메소드 또는 서비스가 어디에 있던지 원격에서 호출하여 사용할 수 있도록 하는 것이다. 이러한 분산 애플리케이션을 개발하기 위해 지원하는 기술로 CORBA와 RMI가 대표적이다.

2.1 CORBA

CORBA(Common Object Request Broker Architecture)[5]는 서로 다른 애플리케이션간을 연결하는 소프트웨어 버스의 역할을 하는 모듈로 원격 시스템에 존재하는 서비스의 메서드를 마치 로컬에 존재하는 메서드처럼 호출하게 하는 하부 구조를 제공한다.[4] 특히 IDL(Interface Definition Language)를 제안함으로서 원격 객체에 접근을 위한 인터페이스의 정의단계와 구현단계를 분리하였으며 이는 이미 개발된 컴포넌트나 다른 언어와의 통합이 용이하다. CORBA의 구성요소로는 IDL Stub, 동적 호출인터페이스(Dynamic Invocation Interface), 코어와 인터페이스로 이루어진 ORB(Object Request Broker), 객체 어댑터(Object Adapter), Skeleton 등이 있다.

CORBA와 Java와의 연동을 위해서는 Java 애플리케이션과 CORBA간의 인터페이스를 제공하는 IDL 프로그래밍이 필요하다. IDL로 작성된 서버 객체에 대한 인터페이스는 JavaIDL 컴파일러에 의해 Stub 및 Skeleton 코드로 매핑된다. 생성된 Stub과 Skeleton 코드를 기반으로 클라이언트와 서버코드를 작성한 후 Java 컴파일러를 통해 바이트 코드로 변환한다.

2.2 RMI

RMI(Remote Method Invocation)[6]는 다양한 언어를 통합하려는 CORBA와는 다르게 Java만을 위한 분산객체 기술이다. RMI의 기본 아이디어는 객체를 생성할 때 내부에 정의된 메서드를 다른 JVM(Java Virtual Machine)에서 호출할 수 있도록 허용하는 것으로 자바 객체를 위한 원격 함수 호출(RPC) 메커니즘을 제공한다. RMI의 메커니즘은 원격 메서드 호출을 위해서 공통적으로 Poxy Stub과 Skeleton Stub을 제공한다. 이로써 통신을 담당하는 하부 구조의 투명성을 보장한다.

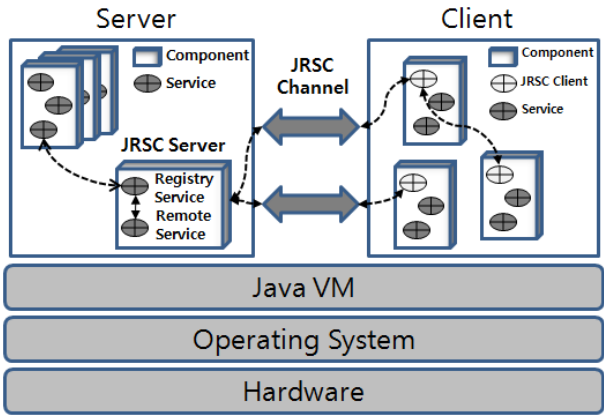
RMI 역시 CORBA의 IDL과 같이 원격객체의 인터페이스를 정의하여야 하며, 인터페이스에서는 클라이언트에 의해 호출될 메서드들을 나열하는데 반드시 java.rmi.Remote 인터페이스를 상속해야 하고 java.rmi.RemoteException과 각 오퍼레이션마다 적절한 예외를 선언해야 한다. 원격 객체가 정의되면 RMI 컴파일러를 통하여 Stub과 Skeleton 코드를 생성하고 이를 기반으로 서버와 클라이언트를 기존의 Java 프로그래밍과 같이 구현한다.

3. 비스텁 방식의 원격자바메서드 호출기법

2절에서 살펴본 CORBA와 RMI는 분산 애플리케이션을 개발하기 위해서 별도의 스텁객체를 생성하여야 했다. 이는 클라이언트사이드를 개발하기 위해서 서버사이드에서 생성된 스텁객체를 제공받아야 하며 이를 기반으로 클라이언트를 구현하게 된다. 서버사이드에서 공개하고자 하는 메서드 또는 서비스의 인터페이스가 정의되지 않는다면 클라이언트는 구현될 수 없으며 서비스의 형태가 변형되었을 때 스텁객체를 재생산하여 클라이언트사이드에 제공하여야 한다. 이러한 부분은 사용자가 분산 애플리케이션을 개발함에 있어서 편의성 및 생산성을 저해 하는 요소가 될 수 있다. 따라서 본 논문에서 비스텁 방식의 원격자바 호출기법을 제시한다.

3.1 JRSC(Java Remote Service Call)

JRSC는 자바 기반의 메서드를 원격에서 호출 및 연동하기 위한 프레임워크이다. 개발자는 JRSC를 통해 특정 서비스를 외부에 공개할 수 있으며, 공개된 서비스는 원격지에서 자유롭게 호출 및 연동이 가능하다. 특히 기존에 발표된 CORBA와 RMI와 같이 별도의 스텁을 생산하지 않으며 단순 코드 추가만으로 서비스를 공개하고 호출 가능하다.



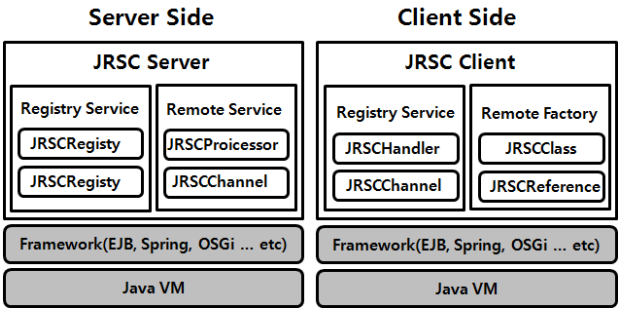
(그림 1) JRSC의 개요

(그림1)은 JRSC의 개요를 설명하고 있다. JRSC는 서버사이드에는 컴포넌트 형태로, 클라이언트사이드에는 라이브러리형태로 제공된다. 서버에는 레지스트리 서비스와 원격서비스로 구성되며 레지스트리 서비스는 다양한 형태의 서비스들에 대한 원격 프록시(Proxy)를 등록하고 관리하는 역할을 담당하고 원격서비스는 서버에서 공개된 서비스들에 대한 호출을 처리하기 위한 역할을 담당한다.

3.2 JRSC의 아키텍처

JRSC는 별도의 프레임워크의 사용 여부에 상관없이 자바환경이면 구동이 가능하며, 서버사이드의 JRSCServer와

클라이언트 사이드의 JRSCClient로 구성된다. JRSCServer은 외부에서 호출 가능한 원격객체인 프록시와 원격객체를 관리하는 레지스트리, 그리고 객체 간에 송수신되는 메시지를 처리하는 채널 및 프로세서로 구성된다.



(그림 2) JRSC 아키텍처

반면 JRSCClient는 원격의 서비스객체를 생성하는 Factory, 로컬에서 이루어지는 호출을 원격 호출로 변환해 주는 핸들러와 채널로 구성된다.

3.3 JRSC의 동작메커니즘

JRSC의 서버사이드 즉, JRSCServer는 JVM또는 기타 프레임워크 내에서 구동된다. 이는 서버에 설치되었거나 설치되는 서비스에 대한 모니터링을 수행하며 서비스 중 외부에 공개하려는 서비스에 대해 원격객체를 생성하고 이를 서비스레지스트리에 등록 한다.

서버사이드에서 특정 서비스를 외부에 공개하기 위해서는 서비스 객체를 생성한 후, 해당 객체를 인터페이스 클래스 기반으로 레지스트리에 등록을 하여야 한다. (그림3)은 서버사이드에서 서비스를 등록하는 과정이다.

```
// Create service object
RemoteService service = new RemoteServiceImpl();

// Set general properties
Dictionary props = new Properties();
props.put("category", "service");

// Set JRSC Interface property
props.put("RemoteInterface",RemoteService.class.getName());
```

(그림 3)JRSC 서비스 등록

생성된 서비스 객체에 대해서 RemoteInterface를 이용하여 원격인터페이스를 지정하고 해당 원격인터페이스에 지정된 메서드들을 원격지에서 JRSCClient를 이용하여 자유롭게 호출이 가능하다.

(그림 4)는 서버에서 공개한 서비스에 대한 인터페이스이다. 이를 원격지에서 호출하기 위한 과정은 (그림 5)와 같다. 클라이언트에서는 우선적으로 원격객체의 생성을 담당하는 JRSCFactory 객체를 생성하여야 한다. 이를 이용하여 원격객체인 JRSCClass 객체를 생성 할 수 있다. 생성된 서비스 객체에 대한 메서드를 getMethod를 통해서 핸들러를 등록하고 invoke를 통해서 원격지의 메서드 호출 할 수 있다.

```
package remote.service;

public intergace RemoteService{

        public String getServiceInfo();
        public void setServiceInfo(String ID);

}
```

(그림 4) 서버사이드 서비스 인터페이스

```
public static void main(String[] args){

  // Create remote class
  RemoteReferenceFactory factory = new RemoteReferenceFactory("203.253.23.194");
  RemoteClass rc =factory.getRemoteClass("remote.service.RemoteService");

  if ( rc != null )
  {
    // Get remote method handle
    RemoteMethod getService = rc.getMethod("getServiceInfo");
    RemoteMethod setService = rc.getMethod("setServiceInfo");

    int i = 0 ;
    while ( true ) {
      try {
        String serviceInfo;
        // invoke remote method to server
        serviceInfo = getService.invoke();
        setService.invoke("ID0012");
      } catch (Throwable e) {
        e.printStackTrace();
      }
    }
  }
}
```

(그림 5) 클라이언트사이드 서비스 호출

4. CORBA, RMI, JRSC의 개발과정 비교

CORBA와 RMI, 그리고 본 논문에서 제시하는 JRSC 모두 분산 애플리케이션을 개발하기 위한 기술로 원격의 서비스를 자유롭게 호출할 수 있도록 지원한다. 이번 절에서 각각의 개발과정을 비교함으로서 JRSC의 사용자 편의성 및 생산성에 대해 검증한다.

<표 1> CORBA, RMI, JRSC 개발단계비교

	CORBA	RMI	JRSC
개발 단계	1.Interface 정의 2.Stub/Skeleton 생성 3.Service 구현 4.Client 구현	1.Interface 정의 2.Service 구현 3.Sub 생성 4.Client 구현	1.Interface 정의 2.Service 구현 3.Client 구현
스텁 사용 유무	사용	사용	미사용

<표 1>은 CORBA와 RMI, 그리고 JRSC의 개발단계를 비교한 것이다. 세 기법 모두 초기에 인터페이스를 정의한다. 하지만 CORBA의 경우 정의한 인터페이스에 대한 Stub 과 Skeleton을 "IDL2Java"라는 별도의 컴파일러를

사용하여 생성이 선행되어야 한다. 생성된 Skeleton을 이용하여 서버사이드 즉, Service를 구현하고 Stub을 이용하여 Client를 구현하게 된다. 또한 RMI는 정의한 Interface에 대한 Service를 구현하고 이를 기반으로 RMI컴파일러를 이용하여 Stub을 생성하여야 한다. 생성된 Stub을 이용하여 Client를 구현하게 된다. 하지만 본 논문에서 제시하는 JRSC는 Interface 정의 후, Service를 구현하고 별도의 Stub생성 과정 없이 바로 Client를 구현한다. 기존의 방식은 Stub 또는 Skeleton이 생성되지 않는다면 서버 또는 클라이언트 구현이 불가능하다. 하지만 JRSC는 Interface정의만 된다면 서버와 클라이언트가 동시에 구현할 수 있다는 장점이 있다. 또한 별도의 컴파일러가 필요한 CORBA와 달리 Java 설치만으로 쉽게 개발이 가능하다. 또한 개발도중 서비스가 변경되어 수정이 불가피 할 때 CORBA나 RMI는 Stub생성 과정을 다시 수행하여야 하며 이를 클라이언트에게 제공하여야하는 불편이 있다. 하지만 JRSC는 클라이언트의 코드만 수정하게 되면 서버의 변경에 쉽게 대처할 수 있는 장점이 있다.

5. 결론 및 향후 연구

본 연구를 통해서 스텁을 생성하지 않는 비스텁 방식의 원격자바메서드 호출 기법을 제안하였다. 본 연구를 통해 기존 분산 애플리케이션 개발에 활용된 기법에서 스텁을 생성하는 과정을 생략함으로서 개발과정을 간소화 하였다. 또한 사용자는 개발시간을 단축할 수 있으며 자바 환경이면 어떠한 프레임워크 환경에서도 통합 운영될 수 있도록 구현하였다. 따라서 JRSC를 사용함으로써 분산 애플리케이션을 개발하기 위한 편의성 및 생산성 향상을 기대 할 수 있으며 원격 서비스에 대한 접근성 또한 크게 향상될 것으로 기대하고 있다.

향후 연구로는 성능향상의 측면에서 서비스의 호출에 따른 오버헤드로 인한 성능저하에 대한 연구가 진행될 것이다. 또한 보안문제에 대한 연구와 다양한 환경을 통합하기 위한 연구도 추가적으로 진행될 것이다.

참고문헌

[1] 방승준, 안진호 “소켓 및 RMI 기반 자바 메시지 전달 시스템의 구현 및 성능평가” 2007.
[2] 한국전산원 “분산객체기술표준연구” 1998.
[3] 김분희, 이상윤 ,김식 “RMI 분산처리시스템의 분석 및 구현” 1998.
[4] 현무영, 김식, 이상윤 “RMI와 CORBA 환경하의 객체 번역 시스템의 설계 및 구현” 1999.
[5] John Siegel, 'CORBA Fundamentals and Programming', John Wiley & Sons, 1996.
[6] Sun Microsystems, JDK 1.1 Documentation <URL:http://java.sun.com/>