# Activation 메커니즘 기반의 RMI API 설계 및 구현

조희남[0] 윤기현 정명조 이철훈
충남대학교 컴퓨터공학과 병렬처리연구실
(hncho[0], ghyoon, mjjung, chlee)@ce.cnu.ac.kr

## Design and Implementation of RMI Using Activation

Hee-Nam Jo[0] ,Gi-Hyun Yoon, Myoung-Jo Jung, and Cheol-Hoon Lee
Parallel Processing Laboratory Dept. of Computer-Engineering, Chung Nam National Univ..

**요 약**

객체 지향 분산 시스템에서 어느 한 객체가 메모리를 차지하게 되면 그 객체를 Active 하다고 하며 그 반대를 Passive 하다고 말한다. Activation 이란 Passive 한 객체를 Active 하게 해주는 전환 과정을 뜻한다. 자바의 RMI(Remote Method Invocation)는 Activation 메커니즘을 JDK1.2.x 버전부터 채택해 사용하고 있는데, Activation 메커니즘이 지원되지 않는 JDK1.1.X 의 RMI 에 Activation 메커니즘을 덧붙이는 형태로 API 를 제공하고 있기 때문에 그 빠대는 Activation 메커니즘 기반이 아니므로 그 효율성 및 성능이 떨어진다 볼 수 있다. 본 논문은 이를 개선 Activation 메커니즘 기반의 API 를 설계 구현한다.

## 1. 서 론

분산 시스템을 지원하는 통신 메커니즘에는 소켓과 RPC(Remote Procedure Call)방식이 있는데, 소켓의 경우 일반적인 분산시스템에서도 유연성있게 사용되지만 에러가 나기 쉽고 그 구현이 매우 까다로운 단점을 지니고 있다. 소켓과 다른 통신 메커니즘인 RPC 의 경우에서는 객체 지향 분산시스템에서는 사용할 수 없는 단점을 지니고 있다. 따라서 객체 지향 분산 시스템에 적절한 통신 메커니즘이 필요한데, 그 중 하나가 자바의 RMI(Remote Method Invocation)이다.

분산 객체 시스템의 주요 특징 중 하나는 객체의 지속성이다. 이러한 특징에 의해서 객체 지향 분산시스템에서 파생된 수 많은 불필요한 객체들은 시스템의 자원을 차지하게 되고 이로인해 귀중한 메모리를 낭비하게 된다. 따라서 시스템에 필요한 객체는 계속 메모리에 상주하도록 하고, 그렇지 않은 객체는 메모리에서 제거하여 관리하다가, 차후 필요하게 되면 다시 메모리에 로딩하는 메커니즘이 필요하다. Activation 메커니즘이 이러한 기능을 하며, 시스템의 자원을 확보한 객체를 Active 객체, 그렇지 못한 객체를 Passive 객체라 부르고, Passive 객체에서 Active 객체로의 변환 과정을 Activation 이라 한다.

JDK1.1.X 까지, 자바의 RMI 는 Activation 메커니즘을 지원하지 못 하였으나, JDK1.2.X 버전부터 Activation 메커니즘을 지원하고 있다. 그러나 자바의 Backward Compatibility 특성상 JDK1.2.X 버전부터 RMI 의 빠대는 줄곧 Activation 메커니즘이 지원되지 않는 RMI 에 두고 있기 때문에 그 효율성 및 성능 저하가 생긴다고 볼 수 있다.

본 논문은 이를 개선하기 위해 처음부터 Activation 메커니즘이 지원되는 RMI 의 API 를 설계 및 구현하는데 초점을 맞추고 있다.

논문의 구성은 다음과 같다. 2 장에서는 Activation 메커니즘 기반의 API 구성에 필요한 Activation 모델의 종류와 RMI 에 대한 전반적인 설명을하며, 3 장에서는 본 논문에서 구현하고자 하는 Activation 메커니즘 기반의 RMI API 에 대한 설계 및 구현에 대해 기술하고, 마지막으로 4 장에서는 결론 및 향후과제를 기술한다.

## 2. 관련 연구

Activation 의 종류는 Eager activation, Lazy activation 그리고 Split activation 이렇게 3 가지 종류가 있다. 각각은 장단점이 있으며, 본 연구는 Lazy activation 모델을 사용한다.

RMI 는 3 개의 계층으로 이루어져 있으며, 클라이언트의 요청에 대해 *rmi* 데몬은 적절한 서버를 구동시킨다.

### 2.1 Eager activation

어느 한 객체가 Active 되면 그 객체가 참조하는 모든 객체들도 다같이 Active 하게 된다. 이러한 Eager activation 의 이점은 일단 자기자신을 참조하는 다른 객체가 Active 하게 되면 동시에 자기자신도 Active 하게됨으로, 자기자신이 언제 active 해야 하는지를 고려할 필요가 없다. 그러나 이 모델은 교착상태(Deadlock)가 발생하는 문제점을 지니고 있다. 즉 자기자신을 참조할 경우와 상호참조할 경우 모두 교착상태가 된다는 것이다. 또한 한번에 너무 많은 객체들이 Active 하게 될 경우 메모리의 과다사용 및 전혀 사용하지 않는 객체들까지도 Active 하게 만들어져서 불필요한 메모리의 낭비가 발생하게 된다.

### 2.2 Lazy activation

이 모델은 클라이언트측에서 처음으로 객체를 이용하고자 할 때, 그 객체를 Active 하게 해주는 방식이다. 따라서 클라

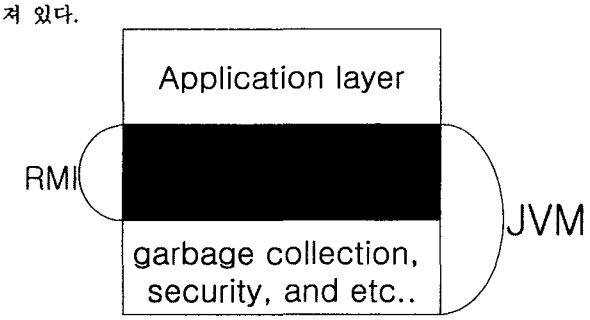
이언트가 그 객체의 함수를 호출하기 전까지 그 객체의 참조는 Fault 한 상태로 남아 있게 되며 이를 Fault block 이라고 부른다. 이 메커니즘의 이점은 Eager activation 모델에서 발생된 교착상태 문제점을 해결할 수 있으며, 또한 함수 호출시에만 객체가 active 하게됨으로 자원을 효율적으로 사용할 수 있다. 그러나 그 반대로 함수 호출시에만 그 객체를 사용할 수 있다는 점이 오히려 단점이 될 수 있으며, Passive 한 객체들을 관리해야 하는 단점이 생긴다. Passive 한 객체들을 분리 및 관리해야 한다는 것은 하나의 관리자를 더 두거나, 이와 대등한 연산을 취해 주어야 한다는 것을 의미한다.

### 2.3 Split activation

Split activation 모델은 서버측과 클라이언트측으로 분리하여, Lazy activation 모델을 사용하는 클라이언트측 Fault block 을 서버측 Fault block 이 제어하는 방식으로, Eager activation 모델과 Lazy activation 모델의 중간적인 형태이다. 이 모델은 Eager activation 이 지니는 교착상태 문제를 해결하고 Lazy activation 모델의 단점들도 해결한다. 그러나 Lazy activation 이 갖고 있는 시스템의 효율적 사용면에서는 떨어지며, 실행시에는 Activation 과정의 제어를 서버측으로 이전해야만 한다.

### 2.4 RMI 의 구조

RMI 의 구조는 [그림 1]과 같이 3 개의 계층으로 이루어져 있다.




[그림 1] RMI의 계층구조

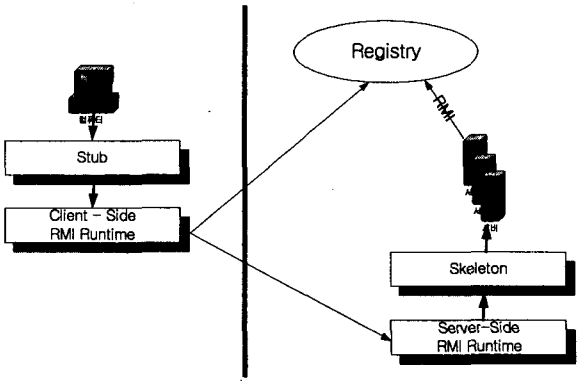
가장 상위 계층인 Stub 및 Skeleton(JDK1.2.X 버전부터 Stub 으로 통일됨) 계층은 메서드호출을 그 하위 계층으로 전달하며, 결과 값을 응용계층(Application layer)으로 전달한다. 메서드 호출의 파라미터들은 이곳에서 Mashalling 되거나 Unmarshalling 된다.

그 하위 계층인 Remote Reference 는 양자간 통신 중 연결이 끊어졌을 경우 다시 시도하는 역할을 수행하며 Marshalling 된 파라미터들을 그 다음 계층으로 전달시킨다.

가장 마지막 계층인 Transport 계층은 연결설정을 하는 부분으로써, 이 계층은 자원의 효율적 활용을 위해서 소켓을 재사용하게 된다.

[그림 2]는 분산 시스템에서 Registry 를 사용한 RMI 응용 프로그램의 동작과정을 보여주고 있다.

클라이언트 컴퓨터 안에서 작동되고 있는 RMI 런타임(Run time)은 서버측 Registry 에서 원하는 객체가 존재하는지를 찾게되고, 그와 관계된 Skeleton reference 를 전달받아 서버측 RMI 런타임과 연결하여 원하는 작업을 수행하게 된다.

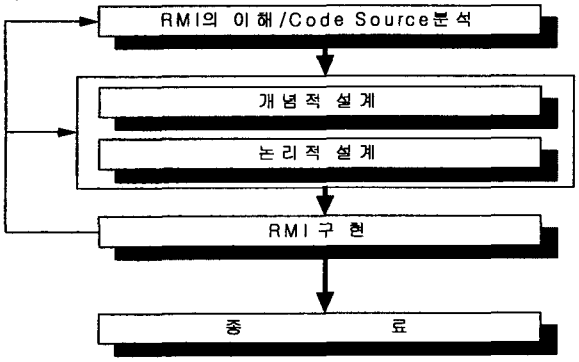



[그림 2] RMI 동작과정

## 3. RMI 의 설계 및 구현

### 3.1 RMI 의 설계

RMI 의 설계는 [그림 3]과 같은 과정을 통하여 이루어 졌다.




[그림 3] RMI의 설계과정

프로그램 중심적 사고방식이 아닌 자료중심적 객체 지향적인 방법으로 RMI 에 대하여 심도있고 체계적인 구현을 위한 개발방법론으로써 각각의 단계는 다음과 같다.

가) Code source 분석 및 RMI 이해단계
중요 특징이 부각되고, 위험부담을 최소화해주며 이해가 쉽고 문서화가 가능한 Data flow 방식을 사용하여 구현하고자 하는 내용을 개체화하는 단계

나) 개념적 설계
도출된 자료들에 대한 개체간의 연관성의 깊이를 추출하는 단계

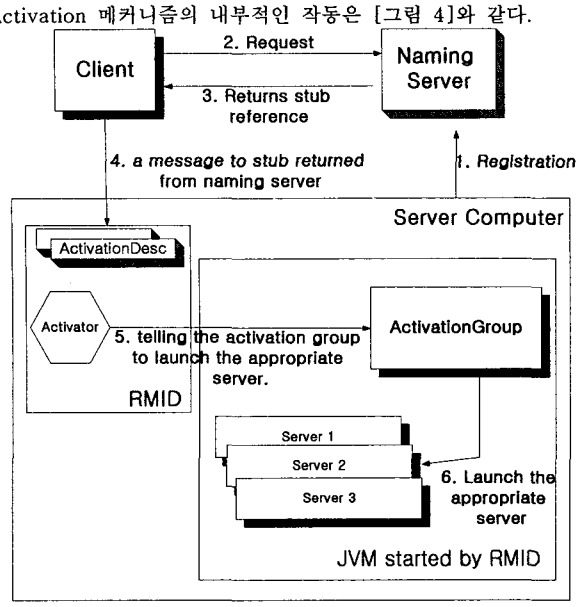
다) 논리적 설계
RMI 구현을 위한 개체내 속성들의 강약관계 설정 및 함수적 종속성을 파악하는 단계

각 단계에서 들어난 문제점에 대해서는 전 단계로 거슬러 올라가 수정 및 정정하여 다시 아래 단계로 내려가는 방식을 선택하였다.

### 3.2 RMI 의 내부동작

RMI 설계과정을 통하여 작성된 문서를 토대로 작성된

Activation 메커니즘의 내부적인 작동은 [그림 4]와 같다.




[그림 4] Activating the server

첫째, 서버 컴퓨터는 Naming 서버에게 자신의 존재를 알리기 위해 Stub reference 를 Naming 서버에게 전달한다. 이는 클라이언트 컴퓨터가 서버 컴퓨터를 찾을 때 발생되는 Bootstrap 문제를 해결해 준다.

둘째, 클라이언트 컴퓨터는 Naming 서버에 자신이 찾고자 하는 서버 컴퓨터를 문의하며, Naming 서버는 클라이언트 컴퓨터가 원하는 서버 컴퓨터를 찾아 Stub reference 를 결과값으로 전달한다.

셋째, 서버 컴퓨터내의 rmi 데몬에 의해 작동되고 있는 Activator 는 클라이언트 컴퓨터로부터 메서드 호출를 접수하여 현재 서버 컴퓨터 안에서 작동되고 있는 자바가상머쉰(Java Virtual Machine : JVM)중 클라이언트 컴퓨터의 메서드 호출과 관계된 JVM 이 존재하는 지를 검사한다. 만약 그러한 JVM 이 없으면 새로운 JVM 을 생성시키고, 새로이 생성된 JVM 안에서 클라이언트 컴퓨터가 필요로하는 객체를 Active 하게 만든다. JVM 안의 객체들은 ActivationGroup 에 의해 관리되는데, 클라이언트 컴퓨터의 메서드 호출과 관계된 객체역시 ActivationGroup 에 의해 active 하게 된다.

넷째, Active 된 객체는 적절한 연산을 수행하게 되며, 그 결과값은 직렬화(Serialization)과정을 거쳐 Marshalling 된 후 클라이언트 컴퓨터에게 전달되게 된다.

현재 Activation 메커니즘 기반 RMI API 는 구현중에 있다.

## 4. 결론 및 향후과제

JDK1.1.X 이후 버전으로 넘어오면서 RMI 에 적용된 Activation 메커니즘은 자원의 효율적인 활용면에 있어서는 그 중요성이 매우 크다 할 수 있다. 그러나 일부분으로써의 Activation 메커니즘의 흡수는 성능 저하의 문제를 지니고 있다고 볼 수 있다. 본 논문은 이러한 점을 개선하여 Activation 메커니즘 기반의 RMI 의 API 구현에 대해 제시하였다. 그러나 본 논문에서의 Activation 메커니즘 기반의 RMI API 의 구현이 현재 완료되지 않았기 때문에, JDK1.1.X 이후 버전과의 상호 성능 테스트 비교에 대한 일반적인 기준을 제시하지 못하고 있다. 따라서 구현이 완료되면 이에 대한 연구가 필요할 것으로 생각된다. 또한 본 논문에서 채택된 Lazy activation 모델과 나머지 두 모델간의 구현상의 차이점 또한 앞으로 연구되어야 할 것이다.

### 참 고 문 헌

[1]**William Grosso, *Java™ RMI*,2002**
[2]**Ann Wollrath, Geoff Wyant, and Jim Waldo ,*Simple Activation for Distributed Objects*, November 1995**
[3] **Bill Venners, *Inside the Java Virtual Machine*,1999**
[4] **Sun Microsystems, Inc., *Java™ Remote Method Invocation Specification*, Revision 1.50, JDK 1.2, October 1998**
[5] **Sun Microsystems, Inc., *Java™ Object Serialization Specification*, Revision 1.4.3, JDK™ 1.2 Beta4,September 30, 1998**
[6] **Douglas Dunn, *Java™ Rules*,2001**
[7] **Elliotte Rusty Harold, *Java™ secrets*, 1997**