<JSTORM>

파일처리와 네트워크 프로그래밍 - 2

중앙대학교 컴퓨터공학과
자바 동호회
JSTORM
http://www.jstorm.pe.kr

Document Information

Document title:	파일처리와 네트웍 프로그래밍 - 2
Document file name:	파일처리와네트웍프로그래밍2_장대원.doc
Revision number:	<1.0>
Issued by:	<장대원> dwjang@jkdsoft.co.kr
	정리 : <윤준호> junoyoon@orgio.net
Issue Date:	<2000/8/23 >
Status:	final

Content Information

Audience	마이크로 소프트웨어 독자
	자바의 RMI 프로그래밍에 대하여 알고 싶어하는 초보 프로그래머
Abstract	RMI 프로그래밍
	- 간단한 RMI 프로그램을 제작해 봄으로써 RMI의 사용법을 익힌다.
Reference	
Benchmark information	

Document Approvals

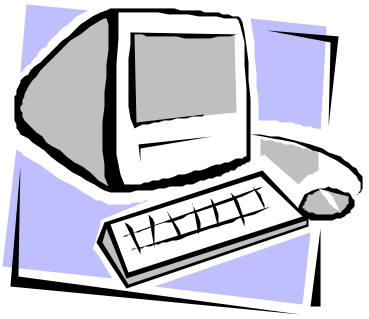
고문	Signature	date
지위	Signature	date

.

Revision History

Revision	Date	Author	Description of change

Table of Contents



파일/소켓/RMI (2)

들어가는 말

이번 시간에는 자바를 더욱 객체 지향적이게 하는 통신 시스템인 RMI(Remote Method Invocation)에 대해 이야기를 해보겠습니다. 지난 시간에 다루었던 파일과 소켓에 대한 내용도 무척 중요한 것이었지만 이미 시중에서 풍부한 자료를 구할 수 있고 또, 다른 언어로 프로그래밍 경험이 있는 사람이라면 벌써 파일이나 소켓에 대한 내용은 자주 다뤄봤을 것으로 생각되니 추가로 더 다루지는 않겠습니다. 자신이 부족하다 싶은 독자들은 시중의 단행본이나 튜토리얼 등으로 보강해 두기 바랍니다.

RMI가 자바를 더욱 객체 지향적이게 한다고 했는데 유닉스 상에서 RPC(Remote Procedure Call) 프로그래밍 경험을 갖고 있는 분들은 RPC와 비교해 RMI의 객체 지향적 특성을 쉽게 파악할 수 있을 것입니다.

우선 둘 다 프로그래머를 네트웍으로부터 어느 정도 격리시켜준다는 공통점을 가지고 있습니다. 그래서 RMI와 RPC 프로그래밍 모두 프로그래머가 네트웍에 대해 많은 신경을 쓰지 않고도 프로그램을 개발할 수 있도록 해줍니다. 개념적으로 본다면 네트웍에 대해 메쏘드나 함수 수준의 추상화된 레이어를 제공한다고 볼 수 있는 것이지요.

RMI와 RPC 프로그래밍 방식의 차이점은 그 이름에서 알 수 있습니다. RMI는 (객체의)메쏘드 호출이 가능한 방식을 말하며 RPC는 단순히 다른 컴퓨터에 존재하는 프로그램의 프로시져 수행을 위한 것입니다. RPC에 객체 지향이라는 옷을 입힌 것이 RMI라 볼 수 있습니다.

RMI 방식의 프로그래밍은 소켓 방식의 프로그래밍과 비교해 볼 때도 많은 장점을 가지고 있습니다. 소켓 프로그래밍은 클라이언트와 서버간에 응용프로그램 레벨의 프로토콜 정의를 필요로 합니다. 응용프로그램 레벨이라는 것은 네트웍을 통해 보낼 메시지와 받는 메시지에 대한 인코딩(encoding)과 디코딩(decoding) 프로토콜을 프로그래머가 직

접 정의하고 구현해야한다는 것입니다. 이러한 작업은 매우 번거로울 뿐만 아니라 에러를 양산할 확률도 높습니다. RMI를 사용하면 자바의 기본 데이터 타입 또는 객체단위의 메시지 송수신이 별도의 프로토콜 없이 가능하므로 프로그래머는 프로그램의 로직에 집중할 수 있습니다. (RMI 내부적으로는 독자적인 프로토콜을 사용합니다. 최근 버전과 이후의 버전에서는 CORBA와의 호환성을 위해 IIOP 프로토콜을 지원합니다.)

독자가 RPC나 소켓 프로그래밍에 대한 경험이 없더라도 걱정할 것은 전혀 없습니다. RMI 방식의 통신 프로그래밍이 무척 쉽기 때문이지요. 사실 RMI 방식의 프로그래밍은 초기 설정 부분이 조금 생소해서 그렇지 나머지는 지금까지 독자가 해 오던 프로그래밍 방식과 다를 것이 거의 없습니다. 그러한 점을 고려해 필자도 RMI의 초기 설정 부분을 집중적으로 다룰 계획이니 독자 여러분들도 초기 설정 과정을 눈 여겨 잘 살펴두기 바랍니다.

RMI에 대한 본격적인 이야기를 시작하기 전에 마소 Jr. 6월호의 "**원격 통신, RMI와 CORBA 맛보기**"를 독자 여러분들이 한번 씩 살펴보고 오기를 권합니다. 필자도 RMI에 대해 기초부터 다룰 생각입니다만 필자의 글에서 자세히 다루지 못하는 개념적인 부분이 아주 잘 설명되어 있으므로 미리 한 번 보고 오는 것도 좋은 생각입니다.

헬로 RMI 프로그램으로 RMI 맛보기.

필자는 간단한 "헬로 RMI" 프로그램을 만들어 두고 RMI 프로그래밍이 필요할 때마다 매번 그 틀로 사용합니다. RMI의 초기 설정 과정이 거의 루틴(Routine)화 되어 있기 때문에 서로 다른 프로그램일지라도 거의 비슷비슷한 구조로 되어 있습니다. 독자들도 매번 처음부터 프로그램을 구현하지 말고 루틴화 되어 있는 부분이 있다면 자신만의 저장 창고를 만들어 보관해 두었다가 필요할 때 꺼내 쓰도록 하세요. 훨씬 편하게 프로그래밍을 할 수 있습니다. 좀 능력이 되는 분들은 자신만의 라이브러리를 구축해 두는 것도 좋은 생각입니다. 그럼 필자의 헬로 RMI 프로그램을 소개해 드리겠습니다.

RMI 프로그램의 개발 또는 실행 단계

1. 서버 측 원격 인터페이스를 개발한다.
2. 서버 측 원격 인터페이스를 구현(implement)하는 구현 클래스를 개발한다.
3. 서버 측 애플리케이션을 개발한다.
4. 클라이언트 애플리케이션에서 서버 측의 원격 객체에 대한 레퍼런스를 얻어온다.
5. 서버 측 원격 객체의 메쏘드를 호출한다.

* RMI 프로그램의 컴파일과 실행 방법은 마소 Jr. 6월호의 "**원격 통신, RMI와 CORBA 맛보기**"를 참조하세요.

<리스트 1>에 프로그램의 서버측 인터페이스와 구현 부분이, <리스트 2>에 클라이언트 측 인터페이스와 구현 부분이 나와 있습니다. 그럼 먼저 RMI의 서버는 어떻게 만들어지는지 살펴볼까요?

서버 부분을 살펴보기 전에 RMI에 대한 사전 지식이 없는 독자를 위해 RMI 방식의 프로그래밍에 대해 간략히 살펴보는 것이 좋겠군요. RMI는 서로 다른 컴퓨터에 존재하는(정확히는 서로 다른 가상 머신에 존재하는) 객체의 메쏘드를 호출하는 것으로 통신이 이루어집니다. 이때 메쏘드는 보통의 다른 메쏘드와 마찬가지로 매개변수나 리턴 값 심지어는 예외(Exception)까지도 가질 수 있습니다.

메쏘드를 호출하는 객체를 "(서비스)요청객체", 메쏘드가 호출되는 객체, 즉 메쏘드를 처리 또는 구현하는 객체를 "(서비스)구현객체"라고 부를 수 있습니다. 그런데 요청 객체는 구현객체에 대한 직접적인 레퍼런스(Reference)를 가지고 작업하지 않습니다. 특별한 인터페이스를 요청객체와 구현객체의 사이에 놓고 바로 이 인터페이스를 통해서만 메쏘드를 호출하게 됩니다.(<그림 1>) 이 인터페이스는 구현 객체가 구현(implements) 하고 있기 때문에 요청객체는 인터페이스를 통해 구현객체에 접근할 수 있습니다.

RMI 서버 만들기

RMI 서버를 만들기 위해서 가장 먼저 고려해야할 것은 서버측 인터페이스를 정의하는 것입니다. 이 인터페이스를 특별히 원격 인터페이스 (Remote Interface)라고 부르는데 클라이언트는 이 인터페이스를 통해서만 실제의 구현객체에 접근할 수 있다고 했습니다. 헬로 RMI 프로그램의 HelloServer.java에는 다음의 두 메쏘드만이 정의되어 있습니다. 따라서 헬로 RMI 프로그램의 클라이언트는 이 두 메쏘드만을 서버객체에 호출할 수 있는 것이지요.

```
public void registerClient(HelloClient client) throws RemoteException;
public void printMessage(String msg) throws RemoteException;
```

두 메쏘드 모두 RemoteException이라는 예외를 발생시킬(throws) 수 있는데 원격 인터페이스에 정의되는 어떤 메쏘드든 바로 이 RemoteException을 정의해 두어야합니다. 네트웍을 통해 통신이 이루어지기 때문에 예외적인 상황이 발생할 확률이 매우 높습니다. 예를 들어 연결이 끊어졌다든지 네트웍 오류가 발생했다든지 하는 경우가 있을 수 있습니다. 그래서 RMI에서는 이러한 예외를 프로그래머가 명시적으로 처리할 수 있도록 모든 원격 메쏘드는 무조건 RemoteException을 처리하도록 하고있습니다.

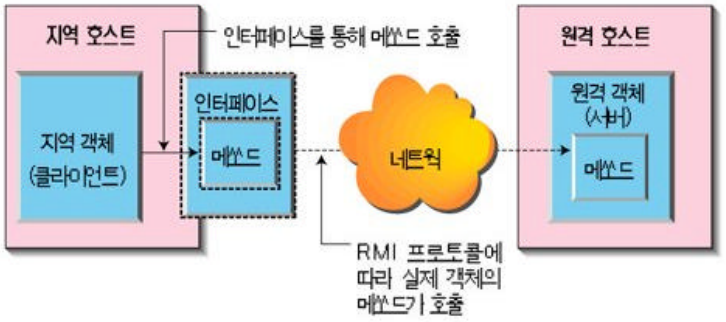
다음 단계는 원격 인터페이스를 구현하는 클래스를 정의하는 것입니다. <리스트 1>의 HelloServerImpl 클래스가 바로 구현 클래스입니다. 구현 클래스는 UnicastRemoteObject 클래스를 상속하는 것이 보통입니다. UnicastRemoteObject가 서버 측에서 필요한 몇 가지 유용한 기능을 제공하고 있기 때문이지요. 그렇지만 UnicastRemoteObject를 상속하

지 않고도 원격 구현 클래스를 정의할 수 있습니다. 클라이언트 부분을 설명할 때 다른 방법에 대해 이야기하겠습니다.

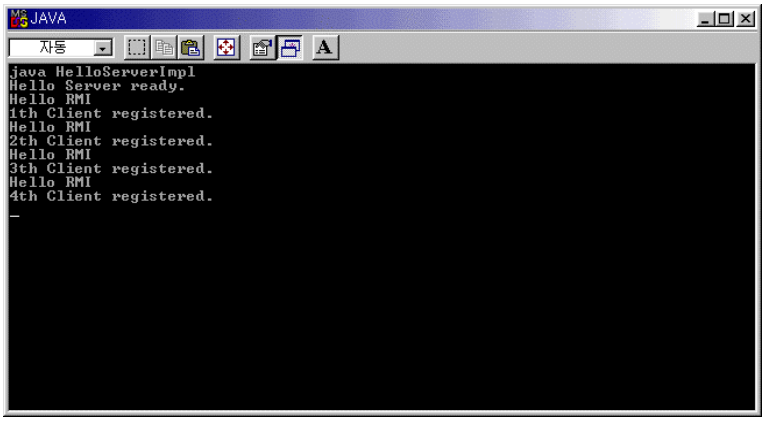
<리스트 1>의 [1번]을 잠깐 볼까요? 수퍼 클래스인 UnicastRemoteObject 생성자를 호출하고 있군요. 익스포트한다는 것은 다른 가상머신에 존재하는 클라이언트가 자신의 메쏘드를 호출할 수 있도록 자신의 스펙(명세)을 공개한다는 것입니다. 원격 객체가 되려면 필수적으로 자신을 익스포트하는 과정이 필요합니다. UnicastRemoteObject의 생성자는 내부적으로 자신을 익스포트하는 코드를 포함하고 있습니다. 따라서 명시적으로 UnicastRemoteObject의 생성자를 호출해 주어야겠습니다. UnicastRemoteObject의 생성자를 사용하지 않고 익스포트하는 방법은 클라이언트 부분에서 살펴보겠습니다.

이번엔 [2번]을 보도록 하겠습니다. Naming이라는 클래스에는 원격 객체를 호스트 컴퓨터에 등록하는 몇 가지 메쏘드가 존재합니다. 원격 객체를 등록한다는 것은 클라이언트가 호스트 컴퓨터에 접속한 다음 그 원격 객체를 찾아낼 수 있도록 호스트 컴퓨터의 등록부에 원격 객체를 등록한다는 것입니다. 이렇게 원격 객체의 등록과 관계되는 메쏘드는 몇 가지 되지만 rebind 메쏘드를 사용하면 대부분 무난히 등록에 성공할 수 있습니다. 등록할 때에는 원격 객체를 식별할 수 있는 이름과 원격 객체를 매개변수로 주면 됩니다.

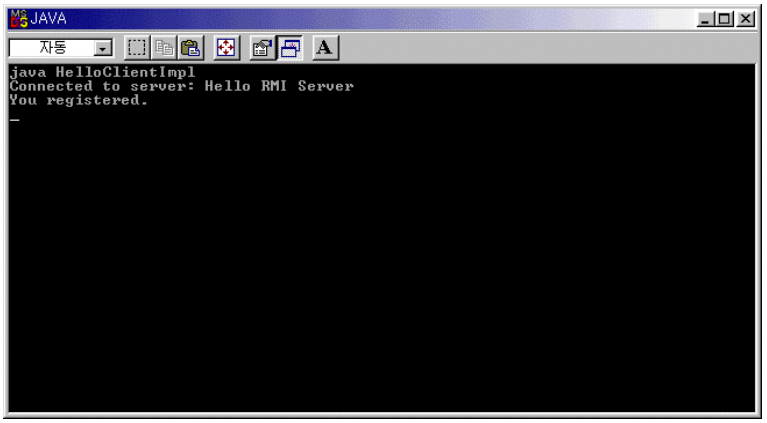
서버 측 메쏘드의 실제 구현 방법은 중요한 사항이 아니니 크게 신경 쓰지 않아도 좋습니다. 단순히 자신에게 접속하는 클라이언트를 등록(register)하거나 클라이언트의 요청에 따라 화면에 메시지를 출력하는 내용이 포함되어 있을 뿐입니다. 독자 여러분들은 RMI 서버를 만드는 과정이 어떤 식으로 진행되는지에 대해 초점을 맞춰주기 바랍니다. 그럼 이번엔 클라이언트 측을 살펴보겠습니다.



<그림 1> 자바의 원격 메쏘드 호출



<화면 1> 헬로 RMI의 서버 측 프로그램 실행화면



<화면 2> 헬로 RMI의 클라이언트 측 프로그램 실행화면

<리스트 1> 헬로 RMI의 서버 측 프로그램

```
// HelloServer.java : 서버 인터페이스
import java.rmi.*;

public interface HelloServer extends Remote {
   // 서버측 컴퓨터에 바인딩 될 서버의 이름
   public static final String SERVER_NAME
         = "Hello RMI Server";
   // 클라이언트를 등록한다.
   public void registerClient(HelloClient client)
         throws RemoteException;
   // 메시지를 출력한다.
   public void printMessage(String msg)
         throws RemoteException;
}
```

```
// HelloServerImpl.java : 서버 구현
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class HelloServerImpl extends UnicastRemoteObject
    implements HelloServer {

  // 접속한 클라이언트들을 담아둔다.
  private Vector m_clients = new Vector();

  public HelloServerImpl() throws RemoteException {

    // 서버 객체를 익스포트(export)한다.
    super();                                          <--- [1번]
  }

  public void registerClient(HelloClient client)
      throws RemoteException {

    m_clients.addElement(client);
    System.out.println(m_clients.size()
        + "th Client registered.");
    try {
      client.printMessage("You registered.");         <--- [3번]
    } catch (RemoteException ex) {}
  }

  public void printMessage(String msg) throws RemoteException {
    System.out.println(msg);
  }

  public static void main(String args[]) {
    try {
      HelloServerImpl server = new HelloServerImpl();
      Naming.rebind(HelloServer.SERVER_NAME, server);   <--- [2번]
      System.out.println("Hello Server ready.");
    } catch (Exception ex) {}
  }
}
```

RMI 클라이언트 만들기

클라이언트에서 살펴 봐야할 내용은 두 가지가 있습니다. 서버에 접속하여 서버 측 원격 객체에 대한 레퍼런스를 얻어오는 방법과 얻어온 원격 레퍼런스를 이용해 원격으로 메쏘드를 호출하는 방법 이렇게 두 가지를 살펴보겠습니다.

<리스트 2>의 [1번]에서 클라이언트가 서버에 접속하여 서버 측 원격 레퍼런스를 가져오는 방법을 볼 수 있습니다. Naming 클래스의 lookup() 메쏘드에 구현 객체의 URL을 매개변수로 넘겨주면 서버에 접속한 다음 구현 객체에 대한 레퍼런스를 얻어올 수 있습니다. 이 때 매개변수로 사용된 URL은 <그림 2>와 같은 형식을 따르게됩니다. 특별히 프로토콜과 포트번호는 생략할 수 있는데 생략될 경우 RMI가 디폴트로 정의하는 프로토콜과 포트번호가 사용됩니다. 독자도 알겠지만 호스트 주소 부분의 "localhost"는 127.0.0.1을 나타냅니다.

```
m_server = (HelloServer)Naming.lookup("//" + "localhost" + "/"
      + HelloServer.SERVER_NAME);
```

이렇게 원격 레퍼런스를 얻어온 다음엔 보통의 메쏘드 호출과 마찬가지로 원격 메쏘드를 호출할 수 있습니다. 명시적으로 RemoteException을 처리해야 한다는 것 말고는 다른 메쏘드 호출과 다른 점이 별로 없습니다. ([3번] 참조)

원격 객체가 되는 또 다른 방법

지금까지 RMI의 기본 틀이 될 수 있는 헬로 RMI 프로그램으로 RMI 프로그램이 어떤 과정과 방법을 통해 만들어질 수 있는지 기본적인 내용을 모두 살펴보았습니다. 앞에서 서버 객체를 원격 객체로 만든 다음 클라이언트가 원격 객체의 메쏘드를 호출하는 방식을 설명하였는데 원격 객체를 만드는 또 다른 방법이 존재합니다. 원격 객체라는 것이 일반적으로 서버 객체를 가르키는 경우가 많지만 때로는 클라이언트 객체가 원격 객체가 될 필요도 있습니다. 원격 객체라는 용어를 엄밀히 정의하자면 다른 가상 머신에 존재하여 원격 메쏘드 호출이 가능한 객체를 말합니다. 보통 클라이언트가 서버 객체의 메쏘드를 호출하는 경향이 있기 때문에 서버 객체가 원격 객체가 되는 경우가 많습니다만 서버도 클라이언트 객체의 메쏘드를 호출해야할 상황이 있을 수 있습니다. 따라서 클라이언트 객체도 원격 객체일 필요가 있을 수 있습니다. 그럼 클라이언트 즉 서버가 아닌 객체를 원격 객체로 만드는 방법을 알아보겠습니다.

원격 객체를 만드는 과정의 앞부분은 RMI 서버를 만드는 과정과 동일합니다. 먼저 원격 인터페이스를 정의한 다음 이 인터페이스를 구현하는 구현 클래스를 만드는 것이지요. 원격 인터페이스는 java.rmi.Remote 인터페이스를 상속하여 만들어야 합니다. 이렇게 구현 클래스까지 만들어 놓았다면 이번엔 구현 객체를 수동으로 익스포트 해야합니다. <리스트 2>의 [2번]을 보세요.

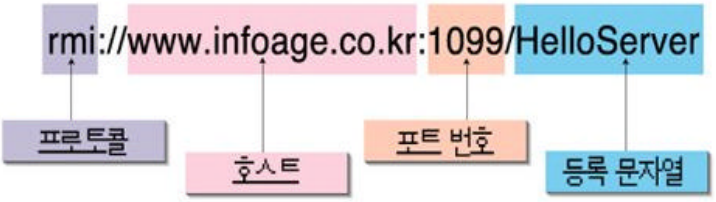
UnicastRemoteObject.exportObject(this);

UnicastRemoteObject 클래스의 정적(static) 메쏘드인 exportObject() 메쏘드를 이용해 원격 객체를 익스포트할 수 있습니다. 앞서 살펴보았던 원격 서버객체는 UnicastRemoteObject 클래스의 생성자에서 익스포트 과정이 이루어진다고 했었습니다. 서버가 아닌 경우 UnicastRemoteObject의 생성자를 이용할 수 없기 때문에 exportObject() 메쏘드를 이용해 수동으로 원격 객체를 익스포트해 주어야합니다. 이렇게 익스포트하는 과정까지 끝나고 나면 어떤 객체든 완벽한 원격 객체가 될 수 있습니다.

재미있게도 원격 객체에 대한 레퍼런스는 네트웍을 타고 여기저기 이동할 수도 있습니다. <리스트 2>의 [3번]에서 다음 부분을 볼까요?

```
m_server.registerClient(this);
```

원격의 서버객체에 역시 원격 객체가 된 클라이언트 객체 자기 자신의 레퍼런스를 넘겨주고 있습니다. 이제부터 서버 측에서는 넘겨받은 원격 클라이언트 객체의 메쏘드를 호출할 수 있습니다. (<리스트 1>의 [3번] 참조) 여기까지 RMI에 대해 초보 프로그래머 여러분들이 알아야 할 내용들을 거의 살펴보았습니다. 그럼 RMI가 실제의 응용 프로그램에는 어떻게 사용되는지 실 예를 보면서 익혀보겠습니다.



<그림 2> RMI의 URL 구성요소

<리스트 2> 헬로 RMI의 클라이언트 측 프로그램

```
// HelloClient.java : 클라이언트측 인터페이스
import java.rmi.*;

public interface HelloClient extends Remote {
    public void printMessage(String msg)
        throws RemoteException;
}

// HelloClientImpl.java : 클라이언트측 구현
import java.rmi.*;
import java.rmi.server.*;
```

```
public class HelloClientImpl implements HelloClient {
    private HelloServer m_server;
    public HelloClientImpl() {
        connectToServer();
        try {
            m_server.printMessage("Hello RMI");                    <--- [3번]
            m_server.registerClient(this);                  <--- [3번]
        } catch (RemoteException ex) {}
    }

    private void connectToServer() {
        try {
            UnicastRemoteObject.exportObject(this);                <--- [2번]
            m_server = (HelloServer)Naming.lookup(                 <--- [1번]
                    "//" + "localhost" + "/"
                    + HelloServer.SERVER_NAME);
            System.out.println("Connected to server: "
                    + HelloServer.SERVER_NAME);
        } catch (Exception ex) {}
    }

    public void printMessage(String msg) {
        System.out.println(msg);
    }

    public static void main(String args[]) {
        HelloClientImpl client = new HelloClientImpl();
    }
}
```

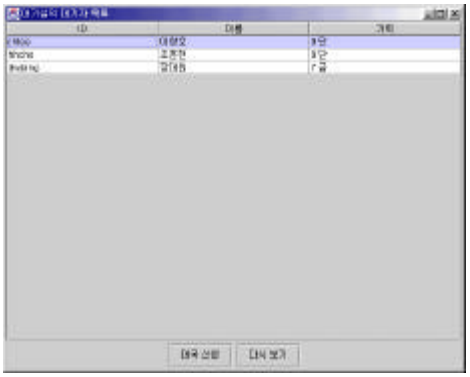
온라인 바둑 서비스 프로그램에 RMI 적용하기

매달 조그마한 프로젝트를 하나씩 소개해 드리고 있습니다만 이번엔 조금 규모가 큰 프로그램 하나를 소개해 드리겠습니다. 물론 이 달의 주제가 RMI인 만큼 RMI를 통신 프레임웍을 사용하는 프로그램입니다. <화면 3>과 <화면 4>에서 프로그램의 실행 화면을 볼 수 있는데 작년 이맘때 필자가 3~4개월에 걸친 긴 기간동안 만들었던 프로그램으로 RMI라는 것을 처음으로 사용해본 프로그램이기도 합니다.

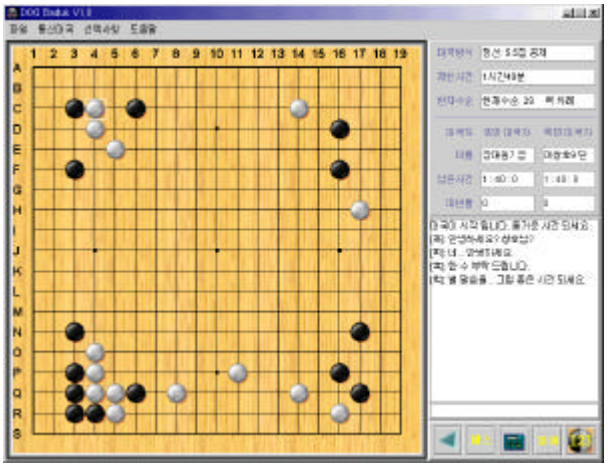
필자가 자바 공부를 시작한 것은 작년 초였습니다. 그래서 이 프로그램을 만들기 전까지 자바에 대해 많이 알고 있다고 할 수가 없었지요. 이 프로그램을 만들면서 많은 시행착

오도 겪고 또 필자가 모르는 부분이 있으면 책과 웹을 뒤지면서 정말 열심히 만들었던 기억이 납니다. 그런데 필자가 언젠가 꼭 만들어보고 싶었던 프로그램이라 그런지(기력은 낮지만 바둑을 좋아하는 관계로) 프로그램 개발 도중 짜증이 나거나 한 적은 없었던 것으로 기억됩니다. 아직도 애착이 많이 남아 있는 프로그램이라 언젠가 정식 버전을 완성해서 공개하고 싶은 마음을 간직하고 있습니다. 아무튼 프로그램을 다 만들었을 때 필자는 스스로 많이 성장해 있음을 느낄 수 있었습니다. 지난 기사에도 말씀 드렸습니다만 독자 여러분들도 꼭 자기가 좋아하는 종류를 택해서 프로그램을 하나 만들어보기를 권합니다. 개안의 기쁨을 맛볼 수 있을 것입니다.

이 프로그램은 인터넷이 연결되어 있는 온라인 상태에서 바둑 대국 서비스를 제공해 주는 프로그램입니다. 이 프로그램을 이해하기 위해 독자가 바둑을 알아야 하는 것은 아니니 긴장하지 말기 바랍니다. (그렇다 하더라도 독자가 오목이나 장기 정도를 알고 있으면 글을 이해하는 데 훨씬 도움이 될 것 같습니다.) 이 글에서는 게임 요소보다는 온라인 서비스라는 주제에 대해 초점을 맞춰 이야기해 보겠습니다.



<화면 3> 온라인 바둑 서비스 프로그램의 대기실 화면



<화면 4> 온라인 바둑 서비스 프로그램의 대국실 화면

<리스트 3> 온라인 바둑 서비스의 서버 측 인터페이스

```
package baduk.network.server;

////////////////////////////////////////////////////////////////////////
public interface ServerInterface extends Remote {

   // 사용자 로그 인
   public boolean login(ClientInterface client)
         throws RemoteException;

   // 사용자 로그 아웃
   public void logout(String playerID) throws RemoteException;

   // 대기실에 대기중인 플레이어들을 얻는다.
   public Vector getWaitPlayers() throws RemoteException;

   // 주어진 ID를 갖는 플레이어에게 대국 신청을 한다.
   public boolean askGame(String playerID, String opponentPlayerID)
         throws RemoteException;

   // 게임이 종료 되었을 때 호출됨
   public void gameFinished(String playerID) throws RemoteException;

   // 주어진 ID를 갖는 클라이언트를 얻는다.
   public ClientInterface getClient(String playerID)
         throws RemoteException;
}
```

<리스트 4> 온라인 바둑 서비스의 서버 측 구현

```
package baduk.network.server;

////////////////////////////////////////////////////////////////////////
public class BadukServer extends UnicastRemoteObject
      implements ServerInterface {

   // all clients
   private Hashtable m_players = new Hashtable();

   // all waiting clients
```

```
private Hashtable m_waitPlayers = new Hashtable();

// ----------------------------------------------------------------
public BadukServer() throws RemoteException {
    super();
}

// ----------------------------------------------------------------
private ClientInterface findClient(String playerID) {
    NetworkPlayer player = (NetworkPlayer)m_players.get(playerID);
    return (ClientInterface) player.m_client;
}

// ----------------------------------------------------------------
private ClientInterface findWaitClient(String playerID) {
    NetworkPlayer player
            = (NetworkPlayer)m_waitPlayers.get(playerID);
    return (ClientInterface) player.m_client;
}

// ----------------------------------------------------------------
private NetworkPlayer findNetworkPlayer(String playerID) {
    return (NetworkPlayer)m_players.get(playerID);
}

// ----------------------------------------------------------------
public boolean login(ClientInterface client)
        throws RemoteException {

    BadukPlayer player = new BadukPlayer();

    try {
    player = client.getPlayer();
    } catch (RemoteException ex) {}

    String playerID = player.getID();

    NetworkPlayer networkPlayer
            = new NetworkPlayer(client, player);

    // 모든 플레이어 리스트와 대기자 리스트에 각각 추가
```

```
        m_players.put(playerID, networkPlayer);
        m_waitPlayers.put(playerID, networkPlayer);

        return true;
    }

    // ------------------------------------------------------------------
    public void logout(String playerID) throws RemoteException {
        m_players.remove(playerID);
        m_waitPlayers.remove(playerID);
    }

    // ------------------------------------------------------------------
    public Vector getWaitPlayers() throws RemoteException {
        Vector waitPlayers = new Vector();

        BadukPlayer player;
        Enumeration enumerator = m_waitPlayers.elements();

        while (enumerator.hasMoreElements()) {
            player = ((NetworkPlayer)enumerator.nextElement()).m_player;
            waitPlayers.addElement(player);
        }

        return waitPlayers;
    }

    // ------------------------------------------------------------------
    public boolean askGame(String playerID, String opponentPlayerID)
            throws RemoteException {

        ClientInterface client1 = findWaitClient(playerID);   <--- [1번]
        ClientInterface client2 = findWaitClient(opponentPlayerID);

        try {
        if (!client2.networkCommandAskGame(client1.getPlayer()))
            return false;
        } catch (RemoteException ex) {}
        // remove two players from wait list
        m_waitPlayers.remove(playerID);
        m_waitPlayers.remove(opponentPlayerID);
```

```
        return true;
    }

    // ----------------------------------------------------------------
    public void gameFinished(String playerID) throws RemoteException {
        m_waitPlayers.put(playerID, findNetworkPlayer(playerID));
    }

    // ----------------------------------------------------------------
    public ClientInterface getClient(String playerID)
            throws RemoteException {
        return findClient(playerID);
    }

    // ----------------------------------------------------------------
    public static void main(String args[]) {

        // Create and install the security manager
        // System.setSecurityManager(new RMISecurityManager());

        try {
            BadukServer server = new BadukServer();
            Naming.rebind(Network.serverName, server);
            System.out.println(
                    "BadukServer created and bound in the registry " +
                    "to the name " + Network.serverName);
        } catch (Exception e) {
            System.out.println(
                    "BadukServer.main: an exception occurred:");
            e.printStackTrace();
        }
    }
}
```

사용자 관리에 대해

그럼 온라인 서비스를 위해 프로그램이 제공해야할 기능은 어떤 것이 있을까요? 대부분의 온라인 서비스 프로그램에서 가장 먼저 고려해야할 것은 사용자 관리일 것입니다. 사용자의 등록, 로그인, 로그아웃 등을 고려해야합니다. 우리 프로그램에서는 사용자 등록이 빠져있습니다만 온라인 서비스를 상업용으로 사용하려한다면 사용자 등록도 필수적인 고려사항일 것입니다. 이 글을 읽고 난 후에 독자들 스스로 사용자 등록에 관한 아이디어들을 정리해 보는 것도 좋겠습니다. 우선은 사용자들이 이미 등록되어 있다고 가정하겠습니다.

사용자 관리를 원활히 하기 위해서는 데이터베이스의 도움을 받는 것이 보편적인 해결방법입니다. 자바에서는 데이터베이스 지원을 위해 JDBC라는 DB 프레임웍을 제공하고 있는데 JDBC를 실제 프로그램에 적용하는 방법은 다음 달에 살펴보겠습니다.

<리스트 3>은 프로그램의 서버 측 인터페이스입니다. RMI 프로그램을 개발하는 가장 첫 단계가 원격 인터페이스를 정의하는 것이라는 것을 잊지 않으셨지요? 이 인터페이스를 보면 login(), logout()이라는 두 메쏘드가 사용자의 로그 인과 로그 아웃을 처리합니다. login() 메쏘드의 매개변수에서 ClientInterface라는 것은 클라이언트 측의 원격 인터페이스를 가르킵니다. 일반적으로 클라이언트는 원격 객체일 필요가 없지만 우리 프로그램에서는 서버의 부하를 줄이기 위해 클라이언트도 원격 객체일 필요가 있습니다. (원격 객체는 다른 가상머신에 존재하는 프로그램에서 자신의 메쏘드를 호출할 수 있는 객체라는 사실을 잊지 안았지요?)

어쨌든 클라이언트가 서버 측 원격 객체의 login() 메쏘드를 호출할 때 자신에 대한 레퍼런스를 넘겨주면 서버는 접속한 클라이언트들에 대해 여러 가지 작용을 할 수 있습니다. 대표적으로 클라이언트의 원격 메쏘드를 호출하거나 클라이언트 측의 원격 객체에 대한 레퍼런스를 다른 클라이언트에게 넘겨주어 서버를 거치지 않고 클라이언트들끼리 통신할 수 있도록 할 수도 있습니다. 서버를 거치지 않고 클라이언트들끼리 통신이 가능하다는 것은 매우 중요한 의미를 갖습니다. 서버에 접속한 클라이언트가 매우 많은 경우 클라이언트들끼리 통신이 가능하다면 서버에 대한 부하를 효과적으로 줄일 수 있을 것입니다. 이러한 메커니즘은 필자의 온라인 바둑 서비스 프로그램에도 적용되어 있습니다.

그럼 서버 측에서는 로그인 해오는 수많은 클라이언트들을 어떻게 관리하는 것이 좋을까요? <리스트 4>에 있는 서버 측 구현 클래스에서 한가지 해결책을 제시합니다. java.util.Hashtable을 이용하는 것이죠. 해시테이블은 전산 전공자라면 가장 기본적으로 알고 있어야할 자료구조 중의 하나입니다. 여기서 해시테이블에 대한 자세한 알고리즘 이야기를 할 수는 없으니 해시테이블에 대해 전혀 모르는 분은 마소 Jr. 5월호 12페이지에 있는 "이상환의 자료구조와 알고리즘" 편을 한번 참조해 보기 바랍니다.

해시테이블의 활용

해시테이블은 키(key)와 값(value) 이렇게 한 쌍이 하나의 레코드를 형성하는 여러 레코드들이 쌓여있는 자료구조입니다. 해시 알고리즘을 이용하면 키를 가지고 매우 빠른 속도로 그 키에 관련된 값을 알아낼 수 있습니다. 자바에서 제공하는 Hashtable 클래스는 키와 값에 모두 Object 형 객체를 사용합니다. 따라서 어떤 객체든 키 또는 값이 될 수 있는 것이죠.

우리 프로그램에서도 이러한 해시테이블의 속성을 이용하여 클라이언트를 관리합니다. 각 클라이언트들은 모두 각자의 ID를 가지고 있는데 클라이언트가 로그인 할 때 각 클라이언트마다 ID와 클라이언트 측 원격 객체를 하나씩 쌍으로 묶어서 해시 테이블에 넣어 둡니다. <리스트 4>의 login() 메쏘드에서 그러한 일이 어떻게 수행되는지 볼 수 있습니다. 일단 <표 1>과 같이 사용자의 ID와 원격 객체를 쌍으로 해시테이블에 저장해 놓으면 원하는 클라이언트에 대한 원격 객체를 쉽고 빠르게 찾아낼 수 있습니다.

좀 더 구체적인 사용 예를 생각해 볼까요? <화면 3>은 현재 접속해 있는 사용자 중에서 대국(바둑에서는 게임을 대국이라고 부릅니다.)을 하고 있지 않은 사용자들의 목록을 제시하는 화면입니다. 즉, 대국을 하기 위해 대기하고 있는 사용자들의 목록이지요. 사용자는 대기자들 중에 자신이 함께 대국하고 싶은 사람을 선택하여 그 사람에게 대국 신청을 하게 됩니다. 서버 측의 다음 메쏘드가 대국 신청을 가능하게 하는 메쏘드입니다.

```
public boolean askGame(String playerID, String opponentPlayerID)
```

첫 번째 매개변수인 palyerID는 대국을 신청하는 당사자의 ID이며 두 번째 매개변수인 opponentPlayerID는 같이 대국하고자하는 상대방의 ID를 나타냅니다. 그럼 이 메쏘드의 내부를 살펴볼까요? <리스트 4>의 [1번]을 보면 ID를 가지고 두 사용자의 원격 객체를 찾아내는 부분이 있습니다. (findWaitClient() 메쏘드는 매개변수로 주어지는 ID를 가지고 <표 1>과 같은 해시테이블을 뒤져서 원격 객체를 반환해주는 메쏘드입니다.) 이렇게 원격 객체에 대한 레퍼런스를 찾아내었으면 이제는 원격 객체의 메쏘드를 호출할 수 있습니다.

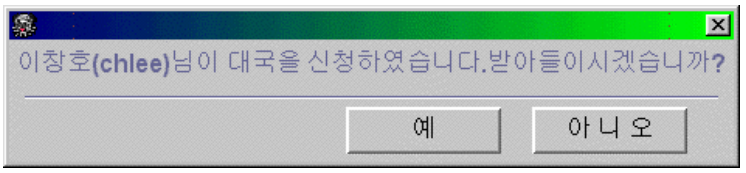
```
client2.networkCommandAskGame(client1.getPlayer())
```

client1,과 client2 모두 원격 객체입니다. client1은 대국을 신청한 쪽의 원격 객체이며 client2는 대국하고자 하는 상대방 쪽의 원격 객체입니다. 먼저 대국을 신청한 사람에 대한 정보를 알아내기 위해 client1에 대국자 정보를 얻어오는 메소드를 호출하여 그 정보를 가지고 client2에게 대국 신청을 하게 됩니다. 그러면 client2 쪽에는 <화면 5>와 같은 대화상자가 나타나게 되는 것이죠.
해시테이블은 그 활용 가치가 매우 높습니다. 보통의 경우 java.util.Vector 클래스를 이용하면 목록관리가 가능하지만 우리 프로그램에서처럼 검색이 필요한 경우엔 해시테이블

이 적격입니다. 벡터, 트리, 리스트, 해시테이블 그리고 독자들도 잘 알고있는 배열같은 것들을 콜렉션(Collection) 또는 컨테이너(Container)라고 부르는데 여러 개의 원소(Element)들이 모여 있다는 의미에서 그런 이름을 갖고 있습니다. 자바에서는 이런 자료구조들을 위해 콜렉션(Collection)이라고 하는 체계화된 프레이웍을 제공하고 있습니다. 그 중요도가 높은 만큼 이번 연재의 후반부에 다뤄볼 생각입니다.

<리스트 3>과 <리스트 4>와 앞에서 보았던 헬로 RMI 프로그램과 비교해 보세요. 그 구조가 매우 흡사함을 알 수 있습니다. RMI의 초기 설정과정이 루틴화되어 있기 때문입니다. 그래서 특별히 이 부분에 대해서는 필자가 더 이상 언급하지 않을 생각이니 아직 RMI 설정 과정에 익숙하지 않은 분들은 헬로 RMI 프로그램과 온라인 바둑 서비스 프로그램을 비교해 보면서 RMI의 설정 과정을 익혀보기 바랍니다.

<화면 5> 대국 신청 대화상자.



<표 1> 클라이언트들을 관리하기 위한 해시 테이블의 내용물

(Key) 클라이언트 ID	(Value) 원격 객체
hhcho	원격 객체 2
chlee	원격 객체 1
dwjang	원격 객체 3
…	…
…	…

클라이언트 측 살펴보기

클라이언트 측 프로그램은 무척 복잡합니다. 바둑 대국에 관련된 여러 가지 메쏘드들이 존재하기 때문이죠. 그래서 클라이언트 측에서는 필요한 몇 개의 메쏘드만 살펴보는 것으로 끝내겠습니다. 프로그램의 전체 소스코드는 이 달의 디스켓에 모두 담아 올릴 생각이니 전체 구조가 궁금한 분들은 그것을 참조하기 바랍니다.

<리스트 6>에서 [1번]을 보면 대국 시 상대편 클라이언트의 원격 객체에 대한 레퍼런스를 담아 두기 위해 ClientInterace 형의 레퍼런스를 담아두는 곳이 있습니다. 대국을 신청하여 상대편이 승낙하면 서버 측에서 상대편 클라이언트의 원격 레퍼런스를 얻어오게 됩니다. 이렇게 상대편 클라이언트의 원격 객체에 대한 레퍼런스를 가지고 있으면 이제부터는 서버를 통하지 않고도 클라이언트끼리 직접 RMI 통신을 할 수가 있습니다. 물론 클라이언트가 원격 객체이기 때문에 가능한 것입니다. 클라이언트를 원격 객체로 만드는 방법은 앞에서 헬로 RMI 프로그램을 만들 때 살펴보았습니다. 클라이언트 측도 앞에서 보았던 헬로 RMI와 매우 유사한 구조를 가지고 있으므로 자세한 설명은 이 정도로 마치겠습니다. 기타 내용은 소스코드를 보면서 음미해보기 바랍니다.

<리스트 5> 온라인 바둑 서비스의 클라이언트 측 인터페이스

```
package baduk.network.client;

////////////////////////////////////////////////////////////////////////
public interface ClientInterface extends Remote {
   public BadukPlayer getPlayer() throws RemoteException;
   public boolean networkCommandAskGame(BadukPlayer player)
         throws RemoteException;
}
```

<리스트 6> 온라인 바둑 서비스의 클라이언트 측 구현 클래스

```
package baduk.command;

/**
 * 프로그램의 모든 중요 명령들을 처리하는 명령 처리기다.
 * 사용자가 내리는 대부분의 명령들은 이곳에서 가공되어 실제 명령
 * 처리 객체들에게 전달 된다. 특히 통신 대국시에는 클라이언트에
 * 관련된 모든 서비스를 제공한다. 즉 서버나 다른 클라이언트들로부터의
 * 명령들을 프로그램 내의 다른 명령 처리객체나 또는 상대편
 * 클라이언트에게 전달하는 기능을 수행한다. 프로그램 내에서 가장
 * 중요한 일을 담당하고 있다 할 수 있겠다.
 */
////////////////////////////////////////////////////////////////////////
public class CommandController implements ClientInterface {

   // 바둑 서버
   private ServerInterface m_server;

   // 통신 대국시의 상대 클라이언트
   private ClientInterface m_opponentClient;                    <--- [1번]

   // 현재 로컬 대국자
   private BadukPlayer m_thisPlayer;

   // 현재 상대 대국자
   private BadukPlayer m_opponentPlayer;

   // ---------------------------------------------------------------
   // 로컬 대국자에 대한 정보를 얻는다.
```

```
public BadukPlayer getPlayer() throws RemoteException {
   return m_thisPlayer;
}

// ----------------------------------------------------------------
// 서버에 연결하여 로그인 한다. 로그인이 성공적으로 수행되면 즉시
// 대기자 목록 창이 열린다.
public void connectToServer() {

   // 현재 로컬 대국자에 대한 정보를 입력 받는다.
   m_thisPlayer = new BadukPlayer();
   EditPlayerDialog dialog = new EditPlayerDialog(m_thisPlayer);

   try {
   // 성공적으로 로그인 될 때까지 계속 서버에 로그인을 시도한다.
   while (true) {
      dialog.show();
      UnicastRemoteObject.exportObject(this);
      m_server = (ServerInterface)Naming.lookup(
            "//" + dialog.getServerAddress() + "/"
            + Network.serverName);
      System.out.println(Network.serverName
            + "에 연결되었습니다.");

      // 성공적으로 로그인 되었을 때
      if (m_server.login(this)) {
         showMessage("성공적으로 로그인 되었습니다.");
         break;
      }

      // 로그인에 실패했을 때
      else
         showMessage(
               "ID 또는 비밀 번호가 잘 못 되었습니다." +
               "다시 시도해 주세요.");
   }
   } catch (Exception ex) {
      System.out.println("Server not found!!");
   }

   m_menuToolbar.setToDisconnect();
```

```
    showWaitPlayerList();
}

// ----------------------------------------------------------------
// 어느 대국자가 대국을 신청해 왔다.
public boolean networkCommandAskGame(BadukPlayer player)
        throws RemoteException {
    String message =
            player.getName() + "(" + player.getID() + ")" +
            "님이 대국을 신청하였습니다." +
            "받아들이시겠습니까?";

    // 대국 신청을 받아들인 경우
    if (BadukDialog.showYesNoMessage(message)
            == BadukDialog.idYes) {
        hideWaitPlayerList();
        return true;
    }
    else
        return false;
}

// ----------------------------------------------------------------
// 로컬 대국자가 어느 대국자에게 대국 신청을 한다.
public void localCommandAskGame(String opponentPlayerID) {
 if (opponentPlayerID.equals(m_thisPlayer.getID())) {
     JOptionPane.showMessageDialog(null,
             "자신에게 대국 신청할 수 없습니다.");
     return;
 }

    startWaitMessage(
            "대국 신청을 하였습니다." +
            "응답이 올 때까지 기다리십시오.");
    try {
    boolean reply = m_server.askGame(m_thisPlayer.getID(),
            opponentPlayerID);
    stopWaitMessage();

    if (reply==true) {
        hideWaitPlayerList();
```

```
            // set opponent
            m_opponentClient = m_server.getClient(opponentPlayerID);
            m_opponentPlayer = m_opponentClient.getPlayer();

            // set me to opponent
            m_opponentClient.setOpponentClient(this);

            m_gameManner = new GameManner();
            startNetworkPlay();
        }
        else
            showMessage("대국 신청이 거부되었습니다.");
        } catch(RemoteException e) {}
    }
    // ----------------------------------------------------------------
    // 대기자 리스트를 얻는다. (WaitPlayerList 객체를 위해)
    public Vector getWaitPlayers() {
        Vector players = new Vector();
        try {
        players = m_server.getWaitPlayers();
        } catch (RemoteException ex) {}
        return players;
    }
}
```

RMI를 적극 활용하여 능률적인 개발자가 되자

지금까지 RMI에 대한 기초적인 사용법을 비롯해서 RMI가 어느 정도 규모 있는 실제 프로그램에 사용되는 예까지 살펴보았습니다. 온라인 바둑 서비스 프로그램에 대해서는 다른 내용들도 다루고 싶었지만 지면 관계상 RMI가 사용된 부분만 간략히 소개하게 되어 아쉬움이 많이 남습니다. 필자가 다루지 못한 부분에 대해서는 이 달의 디스켓을 통해 독자들 스스로 탐구해 보기 바라며 의문 나는 사항이 있으면 언제든 필자에게 질문 메일을 보내주기 바랍니다.

자바의 초기 버전에는 RMI가 없었기 때문에 RMI를 매우 생소한 것으로 생각하는 독자도 있었을 것입니다. 그렇지만 일정한 개발 단계만 잘 준수한다면 RMI의 사용법이 어렵지는 않다는 것을 알 수 있었습니다. 뭐든지 그렇지만 RMI도 깊게 파고들면 어렵고 복잡한 부분이 있게 마련입니다. 필자가 이 글에서 다룬 내용은 빙산의 일각에 지나지 않지요. 필자가 참조한 RMI 관련 문서는 선(Sun)사에서 직접 배포하는 RMI의 스펙이었습니

다. 그런데 필자가 그 문서를 보면서 놀란 것이 있다면 RMI 프레임웍이 매우 크고 복잡한 구조로 이루어진데 반해 실제로 일반적인 프로그래머들이 알아야할 부분은 극히 일부분이라는 것입니다. 물론 RMI를 이용해 저수준(Low Level)의 프로그래밍을 해야할 프로그래머들도 있지만 일반적인 경우 RMI의 그러한 내부 메커니즘은 몰라도 프로그램을 만드는데 거의 지장이 없다는 것이지요. 아주 다행스러운 일이 아닐 수 없습니다. 그 덕분에 독자 여러분들도 몇 번만 RMI 프로그래밍 경험을 쌓게되면 다시는 소켓 프로그래밍을 하고 싶지 않아질 것입니다.

RMI는 내부적으로는 소켓을 기반으로 하고있지만 사용자가 메쏘드 레벨에서 작업할 수 있는 환경을 제공해 줍니다. 그만큼 사용자는 프로그램의 로직에 집중할 수 있어서 프로그램 개발의 능률을 향상시킬 수 있습니다. 성능 향상이 생명인 프로젝트가 아니라면 RMI를 선택하는 것이 현명한 선택일 것입니다. 혹시 RMI를 사용하면서 성능 향상에도 관심 있거나 또는 RMI의 구체적인 내부 구조에 대해 관심 있는 분들은 아래 사이트에서 RMI 스펙을 받아서 읽어보기 바랍니다. 원하는 모든 것이 있을 것입니다.

http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html
ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf

JDK V1.3 미리 보기

필자가 이 글을 쓰기 시작할 때쯤에 JDK V1.3의 첫 번째 베타버전이 발표되었습니다. 1.2 버전 나온 것이 엊그제 같은데 벌써 1.3이라니 세월이 빠르다는 생각도 들었지만 갈수록 공부해야할 내용이 많아지는 것 같아서 부담스럽기도 했었습니다. 그래도 새로워진 자바가 궁금해서 재빨리 SDK와 문서를 받아서 필자의 컴퓨터에 깔아 보았습니다. 제일먼저 살펴본 문서가 "자바 1.3에 새로이 추가된 특징(New Features and Enhancements)"이었는데 이 문서를 보는 필자의 느낌은 실망 반 안도감 반이었습니다. 1.2에 비해 크게 달라진 것이 없었기 때문입니다. 물론 달라진 내용이 양적으로는 상당한 것이었습니다만 기존의 1.2버전을 보완하는 수준이었습니다. 아직 1.3의 정식 버전이 나오려면 멀었기 때문에(공식적으로 내년 1사분기에 출시 예정이지만 필자 생각으로는 훨씬 늦춰질 것 같네요) 나중에 어떤 모습으로 나타날지 모르겠지만 지금 당장은 크게 변화된 모습은 찾기 힘듭니다. 재미있는 것은 JDK V1.2가 나올 때 선에서는 "자바 2"라는 명칭을 붙였었는데 JDK V1.3도 "자바 2"라는 이름으로 부르고 있군요. 아무튼 궁금한 독자들은 아래 사이트를 통해 한 번 씩 받아서 테스트해 보세요. 새로워진 내용 중에 독자들의 관심을 끌만한 내용도 꽤 있으리라 믿습니다.

http://developer.java.sun.com/developer/earlyAccess/j2sdk13/

이왕 소개하는 김에 재미있는 것 하나를 프로그램으로 만들어 보여드리겠습니다. AWT에 새로이 추가된 클래스가 있는데요 이름이 로봇(Robot)입니다. 로봇이라는 이름에서 알 수 있듯이 프로그램에 자동화 기능을 추가할 수 있게 한 클래스인데요 자바 프로그램으로 로우 레벨의 디바이스 제어가 가능하게 합니다. 예를 들어 사용자가 키보드를 누른 것처럼 속인다든지 아니면 마우스 포인터를 프로그램밍적으로 이리저리 움직인다든지 하는 재미있는 기능을 가지고 있습니다. 마우스의 경우 프로그램에서는 사용자가 움직인 것인지 프로그래밍적으로 포인터가 움직인 것인지 전혀 구분할 수 없습니다. 필자가 예제로 만들어본 프로그램을 보면 어떤 클래스인지 금방 알 수 있습니다.

이 프로그램은 1초 단위로 마우스 포인터를 화면상의 (100, 100)과 (400, 400) 사이에서 왔다 갔다 하도록 하는 프로그램입니다. 실제로 이 프로그램을 실행해 놓으면 가만있어도 마우스가 혼자서 움직이는군요. 이 프로그램을 실행시켜 놓고 프로그램을 종료하기 위해 윈도우의 닫기 버튼을 누르려니 무척 힘들더군요. [1번]을 보면 마우스 포인터를 어떻게 옮길 수 있는지 알 수 있겠지요? 아래쪽에서는 역시 Robot 클래스의 메쏘드인 delay()를 이용해 지연 시간도 주어 보았습니다. 이 외에도 로봇 클래스에는 아래와 같은 몇 몇 재미있는 메쏘드들이 존재합니다.

public Image createScreenCapture(Rectangle screenRect)
화면상의 사각 영역을 캡쳐해준다.

public void keyPress(int keycode)
키보드가 눌린 것처럼 해준다.

로봇 클래스는 CBT(computer-based training) 프로그램이나 프로그램의 데모를 작성할 때도 유용하다는군요.

<리스트 1> JDK V1.3의 로봇(Robot) 클래스 예제

```
import java.awt.*;

//////////////////////////////////////////////////////////////////
public class RobotDemo {

    // ------------------------------------------------------------
    public RobotDemo() {

        Robot robot = new Robot();

        for (int i=0; i<1000; i++) {
            if (i%2 == 0)
                robot.mouseMove(100, 100);                    <--- [1번]
             else
                robot.mouseMove(400, 400);

            robot.delay(1000);
        }
    }

    // ------------------------------------------------------------
    public static void main(String[] args) {
        new TestClass();
    }
}
```

IE에서 RMI 실행하기

IE (Internet Explorer)는 RMI를 지원하지 않습니다. MS와 선의 자바 관련 법정 시비에서 선사가 MS를 상대로 라이센스 계약 위배 이유를 든 것 중 하나가 바로 IE에 RMI가 빠져 있다는 사실이었는데요 무슨 속셈인지 MS는 5.0에서도 RMI를 뺐습니다. 그렇다고 RMI를 아주 쓸 수 없는 것은 아닙니다.

IBM은 자바를 전폭적으로 밀어주는 기업 중 하나인데 바로 이 IBM에서 IE에서도 RMI를 쓸 수 있도록 해주는 패치 프로그램을 공개하였기 때문에 이 패치 프로그램을 다운 받아서 RMI를 실행할 수 있습니다. 여러분들은 아래 사이트를 방문해서 패치 프로그램을 받을 수 있습니다. 그 사용법도 무척 간편하니 다운받아(약 300kb) 사용하기 바랍니다. 물론 넷스케이프에서는 어려움 없이 RMI를 사용할 수 있습니다.

http://www.alphaworks.ibm.com/tech/RMI

RMI 핵심 정리

RMI 방식의 프로그래밍을 선택하고자 하는 사람들은 다음 내용을 필수적으로 염두에 두도록 하세요. RMI 방식이 기존 프로그래밍 방식과 거의 비슷하긴 하지만 몇몇 RMI만의 특징이 존재합니다. 가끔 RMI 초보 프로그래머들에게 버그의 모습으로 나타나기도 하는 특징들이니 주의 깊게 정리해 두기 바랍니다.

1. 클라이언트는 서버 측의 구현 클래스(Implementation Class)를 절대로 직접 사용하지 않습니다. 반드시 서버 측의 원격 인터페이스(Remote Interface)만을 사용하여 구현 객체에 접근할 수 있습니다.

2. 원격 메쏘드의 매개변수나 리턴 값으로 (원격 객체가 아닌) 일반 객체가 넘어갈 때에는 그 객체에 대한 레퍼런스가 넘어가는 것이 아니라 객체의 복사 본이 넘어갑니다. 예를 들어 살펴보겠습니다.

다음과 같은 두 개의 메쏘드가 정의되어 있다고 가정할까요? 하나는 보통의 메쏘드이고 하나는 원격 메쏘드입니다. (단순성을 기하기 위해 문법적으로 정확하지 않은 구문이 나올 수 있습니다.)

```
public void normalMethod(Point pt) {
   pt.x = 100;
   pt.y = 100;
}

public void remoteMethod(Point pt) throws RemoteException {
   pt.x = 100;
   pt.y = 100;
}
```

그리고 위의 두 메쏘드를 호출하는 부분이 다음과 같습니다.

```
Point a = new Point(50, 50);
Point b = new Point(50, 50);

normalMethod(a);
remoteMethod(b);
```

그럼 두개의 메쏘드 호출 후에 Point형 객체 a와 b는 어떤 값이 되어 있을까요? 예상대로 a는 (100, 100)으 값으로 바뀌어 있지만 b는 (50, 50) 그대로 남아있게 됩니다. 일반객체인 b가 RMI 메쏘드 호출의 매개변수로 사용되었기 때문에 그 복사 본이 넘어가게 됩니다. 따라서 remoteMethod() 메쏘드가 복사 본에 수정을 가하더라도 원본은 변하지 않는 것이죠. 객체에 대한 레퍼런스가 넘어가는 것은 서로 다른 가상머신 사이에서는 의미 없는 것이 되어버리기 때문에 RMI 메쏘드 호출 시에는 레퍼런스가 아닌 복사 본이 넘어가도록 정의해 놓은 것입니다.

3. 원격 메쏘드의 매개변수나 리턴 값으로 원격 객체가 넘어갈 때에는 그 객체에 대한 레퍼런스가 넘어갑니다. 앞의 상황과는 반대의 상황입니다.

4. 원격 메쏘드의 매개변수나 리턴 값으로 자바의 원시 타입이 아닌 일반 객체가 넘어가기 위해서는 그 객체를 정의하는 클래스가 java.io.Serializable 인터페이스를 구현(implements)해야 합니다. 더 자세한 내용은 박스기사 "초보 프로그래머의 디버깅 노트"를 참조하세요.

초보 프로그래머의 디버깅 노트

RMI의 마샬링 예외

RMI 프로그래밍을 하다보면 다음과 같은 실행시간(Runtime) 예외를 한번씩은 만나게됩니다. 아래에 보면 MarshalException이라는 예외가 나오는데 마샬(Marshal)이라는 것은 일종의 인코딩(encoding) 또는 변조라는 의미로 생각하면 됩니다. 따라서 인코딩 과정 중에 뜻하지 않는 일이 발생했다는 의미의 예외입니다.

```
java.rmi.MarshalException: error marshalling arguments; nested exception is:
        java.io.NotSerializableException: UserInfo
         java.io.NotSerializableException: UserInfo
        at java.io.ObjectOutputStream.outputObject(ObjectOutputStream.java:845)
        at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:342)
        at sun.rmi.server.UnicastRef.marshalValue(UnicastRef.java:272)
        at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:110)
        at HelloServerImpl_Stub.receiveUserInfo(Unknown Source)
        at HelloClientImpl.<init>(HelloClientImpl.java:13)
        at HelloClientImpl.main(HelloClientImpl.java:33)
```

RMI도 사실은 내부적으로는 소켓을 사용하여 메시지를 주고받습니다. 예를 들어 사용자가 정수 값 하나를 보내려한다면 4바이트인 정수 값을 바이트열로 소켓을 통해 보냅니다. 이때 상호 통신 프로토콜을 위해 약간의 헤더도 포함되는데 이런 식으로 데이터를 가공해서 보내는 것을 마샬링이라고 부릅니다. 일종의 데이터 인코딩입니다.

메시지를 받는 쪽에서는 반대로 헤더를 풀어서 어떤 데이터인지 해석한다든지 하는 반대 작업이 필요합니다. 이러한 반대 작업을 언마샬(Unmarshal)이라고 합니다. 아무튼 RMI에서 메시지를 주고받기 위해서는 마샬링과 언마샬링 과정이 반드시 필요한데 이 과정에서 소켓이 사용된다는 사실이 중요합니다.

그런데 소켓을 이용해 자바의 원시 타입이 아닌 객체를 저장하기 위해서는 그 객체가 직렬화 가능(Serializable)해야 합니다. RMI는 내부적으로 ObjectOutputStream과 ObjectInputStream을 확장한 클래스를 사용하는데 이 클래스를 이용해 객체를 읽거나 쓰려면 반드시 그 객체가 직렬화 가능해야합니다. 객체를 직렬화 가능하게 하는 가장 일반적인 방법은 그 객체를 만드는 클래스가 java.io.Serializable 인터페이스를 구현(implements)하게 하면 됩니다. 그렇다고 사용자가 특별히 어떤 메쏘드를 추가로 구현할 필요는 없습니다. java.io.Serializable 인터페이스가 어떠한 메쏘드도 정의하지 않는 빈 마커 인터페이스(Marker interface)이기 때문이죠. 단순히 클래스를 정의할 때 implements Serializable이라는 구문을 삽입하면 됩니다. 그럼 여기서 위의 에러를 만들어 낸 코드를 살펴볼까요?

```
// 사용자 정보를 저장하는 클래스
public class UserInfo {
    public String m_name;
    public UserInfo(String name) {
        m_name = name;
    }
}
```

위의 클래스에 의해 만들어지는 객체를 매개변수로 넘겨주는 다음과 같은 원격 메쏘드가 있습니다.

```
public void receiveUserInfo(UserInfo userInfo) throws RemoteException;
```

매개변수 userInfo는 마샬링 과정을 거쳐야 네트웍을 타고 넘어갈 수 있다고 했습니다. 그런데 userInfo는 Serializable 인터페이스를 구현하지 않기 때문에 마샬링 과정에서 에러메시지를 출력하게 됩니다. 해결법은 간단합니다. 사용자 정보 클래스의 public class UserInfo {를 public class UserInfo implements java.io.Serializable {로 고치면 끝입니다. RMI가 내부적으로 소켓을 사용하고 있다는 사실을 잊지 말기 바랍니다.