# Párhuzamos és Elosztott Rendszerek II

## JAVA RMI

**Készítette: Dr. Mileff Péter**

**Miskolci Egyetem Általános Informatikai Tanszék**

### 1. Bevezetés

A számítógép-hálózatok rohamos terjedésével a hálózattal összekapcsolt számítógépekből álló rendszerek egyre népszerűbbek, hiszen lehetővé teszik a rendszer erőforrásainak szélesebb körű megosztását (*resource sharing*), a rendszer megbízhatóságának (*reliability*) illetve teljesítményének növelését, valamint a felhasználók egymás közti üzenetváltásait.

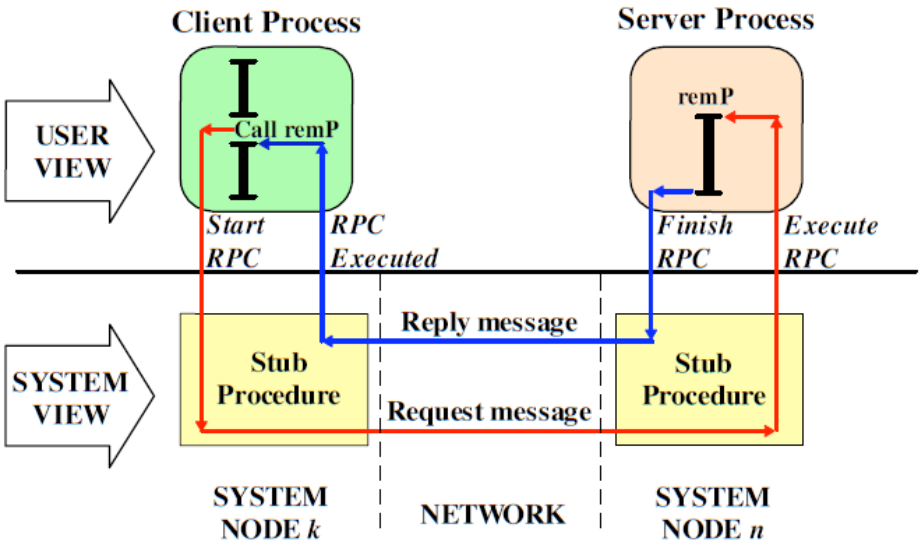
Programozók szempontjából elosztott egy rendszer akkor programozható kényelmesen, ha a lehető legkevésbé kell a hálózati protokollok, kommunikáció programozásával törődnie. A Jáva nyelvnek már a megjelenésénél is hangoztatott, kiemelt tulajdonsága az „elosztottság" volt. Sajnos az első nyilvános verzióban, a JDK 1.0-ban ez csak ígéretnek bizonyult. A korai könyvtárak támogatták ugyan a szállítási szintű hálózati kommunikáció egyszerű programozását, azonban az alkalmazások erre épülő protokollját már külön kellett beprogramozni.

Az 1997. januárjában megjelent új Jáva alap-környezet, a JDK (*Java Development Kit*) 1.1-es változata az RMI könyvtár megvalósításával jelentős újításokat hozott ezen a téren.

### 2. Távoli eljáráshívás

Korábban megfigyelhettük, hogy az elosztott rendszerek programozásában kiemelkedő szerepet játszik az úgynevezett távoli eljáráshívás (*RPC, Remote Procedure Call*) módszere.

Itt a programozó a fejlesztése során egy processzoros rendszerben gondolkozik, ahol egyszerű eljáráshívással valósítja meg a program részei között a kommunikációt. A fejlesztés egy későbbi lépésében szétválaszthatja a hívó és hívott eljárásokat, más-más számítógépre telepítve azokat. A felhasznált rendszerkönyvtárak biztosítják, hogy a hálózaton keresztül is az eljáráshívás, a paraméterek átadása és átvétele automatikusan, a programozó elől „rejtetten" történjen meg. Az alábbi ábra a távoli eljáráshívást általánosan mutatja be:




**1.ábra: Távoli eljáráshívás**

## 2.1 A Java RMI

Az RMI *(Remote Method Invocation*), azaz távoli metódushívás egy olyan eszköz a Java nyelvben, mely lehetővé teszi más VM-ben (Virtual Machine – virtuális gép) elhelyezkedő objektumok metódusainak meghívását. A rendszert úgy tervezték, hogy a hálózaton elosztott objektumok viselkedése hasonló legyen a helyi objektumok viselkedésével, de ne egyezzen meg, a teljes transzparencia biztosítása nem volt cél. A rendszer elrejti a kommunikációt végző részt, és a metódushívás hasonló a helyi, azaz lokális objektumok metódusainak meghívásával, bizonyos megszorításokkal és kötöttségekkel.

Tehát azt mondhatjuk, hogy az RMI az RPC-hez hasonló mechanizmust biztosít, csak objektum orientált módon teszi. A távoli eljáráshívások (RPC) legnagyobb problémája az összetett adatszerkezetek hálózaton keresztüli átvitele. Leggyakrabban az átvihető adatszerkezetek szigorú korlátozásával, a programtól független specifikációjával segítik a paraméterátadást.

A Jáva egyszerűbb, egységesebb szerkezete, a referenciák következetes, kizárólagos használata, valamint a Jáva virtuális gép tulajdonságai miatt az RMI-nél a paraméterátadás, legalábbis a programozó számára nem jelent semmiféle korlátozásokat. Távoli módszereink tetszőleges paraméterekkel rendelkezhetnek, a paraméterátadás alapvetően az objektum másolatának átadásával történik. A csonk az átadandó objektum állapotát (tagváltozóinak értékét) átalakítja a hálózaton átvihető, úgynevezett soros reprezentációvá, amelyet a másik oldal, a csontváz (a másik csonk) visszaalakít, létrehozva a szerver oldalon egy lokális objektumot, majd erre utaló hivatkozást ad tovább a szerver módszer törzsének.

Az RMI másik hatalmas előnye a Java-ban alkalmazott objektummodellhez való illeszkedésen kívül a kommunikáció egyszerűsége. Ugyanis a széles körben alkalmazott kliens-szerver modellben a rendszer részei közötti adatátvitel az I/O köré épül, sokszor saját protokollt kell kiépíteni az információcserére, illetve szinkronizációra. Ilyenkor a program írójának minden kapcsolódási pontnál implementálni kell a hálózati kommunikációt megvalósító eljárásokat, az adatátvitelt és szinkronizációt. (Ez annyit jelent, hogy létre kell hozni egy csatornát a kommunikáló felek között, ami általában egy TCP csatorna, szinkronizálni, kódolni, illetve dekódolni az átvitelre váró adatokat.) Ezen eljárások

hasonlítanak egymásra, de nehéz általános megoldást találni megvalósításukra a speciális esetek miatt, illetve könnyű hibát ejteni. Ezzel szemben az RMI egy magasabb szintű absztakciós eszköz, melynél az információáramlás a metódusok hívásakor átadott paraméterekkel oldható meg. Nagyon fontos, hogy a paraméterek csakis szerializálható adattípusok lehetnek.

## Szerializáció

Egy objektum *szerializáció* során egy bájtfolyammá alakul (általánosan kifejezve: adatfolyam), mely ezek után használható a Java nyelv szabványos kimenet és bemenet kezelési módszereivel. Ennek ellenkezője a *deszerializáció*, mely a visszaalakítást jelenti a bájtsorozatból objektummá. A szerializáció szó, mint technika takarhatja mind az előbbi értelemben vett szerializációt, mind a deszerializációt. A szerializáció alkalmas például egy objektum háttértárra való kiírására, azaz állományba mentéshez, illetve alkalmas a VM-ek közötti kommunikáció megvalósítására, ugyanis a szerializált objektum a hálózaton átküldhető, mint bájtfolyam, és a másik oldalon visszaalakítható objektummá.

A szerializáció egy nagyon előnyös tulajdonsága, hogy egy objektum szerializálásakor nem csak az objektum kerül szerializálásra, hanem minden olyan objektum is, amire az eredeti objektum valamely mezője referenciát tartalmaz. És ez ismétlődik ezen objektumokra rekurzívan. Amennyiben szeretnénk, hogy Java osztályunk szerializálható legyen, akkor az osztálynak implementálnia kell a *java.io.Serializable* interface-t. Pl.:

*class myClass implements java.io.Serializable.*

## 3. A Java RMI működése

A továbbiakban az RMI alapvető működését vizsgáljuk meg, nem említve az implementációs kérdéseket, amelyekre később térünk ki, közvetlenül a működése bemutatása után.

Az RMI tehát olyan eszközt ad a kezünkbe, mellyel más virtuális gépeken lévő objektumok metódusait tudjuk meghívni. Így a rendszer alapfogalma a távoli objektum. Viszont egy objektum nem minden metódusa hívható távolról (csupán azok, melyek egy úgynevezett távoli interfészben definiálva vannak), és fontos momentum, hogy nem biztosítja az osztályok távoli elérését (ami azt jelenti, hogy egy osztály konstruktor műveletei nem lesznek használhatók).

Ezen objektumok használata teljesen megegyezik a távolról nem hívható objektumok használatával, azaz a programozónak csak nagyon kevés helyen kell ezzel foglalkoznia, azaz a rendszer teljesen transzparens módon viselkedik. Az RMI rendszer nem más, mint Java osztályok gyűjteménye, azaz nem nyelvi elem. Mivel a hálózati kommunikációt végző rész teljesen láthatatlan, ezért mégis annak tűnik. Ebből következik, hogy egy saját RMI rendszer implementálása Java nyelven lehetséges, a felmerülő problémák és azok megoldásainak ismeretében. Fontos megjegyezni, hogy a Java fejlesztőkörnyezettel adott RMI rendszer saját szemétgyűjtő mechanizmussal is rendelkezik, és mivel működése hasonló a Java nyelvben, illetve VM-ben implementálttal, így erre nyugodtan hagyatkozhatunk, néhány megkötéssel.

A Java nyelvben nincsen olyan referencia, mely egy másik virtuális gépen elhelyezkedő objektumot címezne meg, ezért ehelyett egy úgynevezett **csonkot** kell használni.

A csonk egy olyan objektum, mely a lokális virtuális gépen helyezkedik el, és hasonlít a távoli objektumra azzal a különbséggel, hogy nincs benne implementálva a távoli objektum metódusainak törzse. A metódushívást tehát úgy kell elképzelni, hogy a csonk egy metódusát

hívjuk meg, mely a JRMP (Java Remote Method Protocol) protokollt használva továbbítja a hívást a távoli objektumnak, szerializálva a paramétereket, blokkolva a végrehajtást, majd ha visszaérkezik a visszatérési érték a hálózaton, deszerializálja azt, és visszaadja azt a hívó félnek. A csonk elkészítése automatikus, a Java fejlesztőkörnyezetben találunk hozzá eszközt. Tehát a csonk használata teljesen transzparens, egyenértékű a távoli referenciával. A hívás során a hálózaton a következő információk mennek át, melyek el vannak rejtve: a hívott objektum azonosításához szükséges információk, a hívott metódus azonosításához szükséges információk, illetve a metódushívás aktuális paraméterei, szerializálva. Fontos, hogy más-más virtuális gépen az ugyanarra az objektumhoz tartozó csonk igaz ugyan, hogy különböző objektum, de ugyanarra az objektumra vonatkozó referencia.

Míg a fogadó oldalon RMI háttérszálak futnak, melyek várják a beérkező távoli metódushívásokat. Amint kapnak egy ilyet, továbbadják azt a megcímzett objektumnak, mely ugyanazon a virtuális gépen helyezkedik el. Ebben már implementálva van a metódus törzse, így végrehajtódik az. A visszatérési értéket aztán visszaküldi a hívó oldalon szereplő csonknak, mely továbbadja a hívónak. Az előlünk **elrejtett információk**, melyek átmennek a hálózaton: a távoli metódus visszatérési értéke, és a kivétel, melyet vagy a távoli objektum dobott, vagy az RMI rendszer működése közben lépett fel.

A kivételkezelés elosztott környezetben bonyolult probléma, melyeket mindig a hívó oldalon kell lekezelni, amik szintén szerializálva mennek át a hálózaton. Az RMI rendszer működése közben fellépett hibák válthatják ki a távoli metódushívás kivételeit, melyet például a hálózat megszakadása vagy a hívott fél helytelen működése okozhat. Egyéb dobott kivételeknél a szerializáció során elveszhetnek egyéb járulékos információk, melyek a hibakeresést jelentősen megkönnyítenék. Ebből is látszik, hogy a paraméterként átadott objektumok nem mennek vissza a hívó félhez. Ez azt jelenti, hogy a paramétereken véghezvitt változás nem aktualizálódik a hívó oldalon, azaz érték szerinti paraméterátadásról beszélünk. Ez közvetlenül abból következik, hogy a deszerializáció során a hívott oldalon egy másolat keletkezik az eredeti objektumból.

## Exportálás

Abban az esetben, mikor példányosítunk egy objektumot, ami távolról elérhetővé akarunk tenni, közölni kell az RMI háttérszálakkal a készen állását, hogy most már fogadhat távoli metódushívásokat erre az objektumra vonatkozólag, illetve azért, hogy tudja, hova kell a hívást továbbadni. A közlést exportálásnak nevezzük.

Természetesen egy objektumot unexportálni is lehet, ezek után nem érhető el többet RMI-n keresztül. Exportáláskor létrejön egy csonk példány is a hívott oldalon, melyet használva a hívó oldalon érhetjük el az eredeti, távolról elérhető objektumot.

## RMI registry

Ahhoz, hogy referenciához jussunk a távolról hívható objektumra, szükségünk van a hozzá tartozó csonkra. De ahogy korábban említettük, a csonk példány a hívott oldalon keletkezik, tehát valahogy el kell jutnia a hívó félhez. Ennek lekérése az RMI registry-n keresztül történik, méghozzá RMI technikával, ugyanis a registry nem más, mint egy távolról elérhető objektum, melynek távolról hívható metódusaival rendelhetünk egy objektumhoz egy referenciát, kérhetjük le azt, illetve távolíthatjuk el a registry-ből. A registry csonk és név párokat tartalmaz. Egy távoli objektum azonosítását egy speciális cím teszi lehetővé, melyben szerepel a számítógép neve, opcionálisan egy port, illetve az objektum neve. A Java fejlesztő környezetben kapunk egy segédprogramot, mely létrehoz egy registry példányt, és exportálja azt. Erre azért van szükség, mivel a registry is egy távolról elérhető objektum. Viszont honnan

tud a hívó fél referenciát szerezni a registry-re? Mivel tudjuk ennek a címét és a port számát, valamint speciális azonosítóval rendelkezik, ezért csonkot generálhatunk anélkül, hogy kapcsolatba lépnénk az registry-t példányosító és azt exportáló virtuális géppel. Persze saját magunk is implementálhatunk RMI registry-t, sőt mi is példányosíthatjuk és exportálhatjuk a programból.

Fontos megjegyezni, hogy nem csak az registry-n keresztül kaphatunk meg egy referenciát ez távolról elérhető objektumra, hanem egy távoli metódushívás visszatérési értékeként is megkaphatunk egy hivatkozást, ami nem más mint egy csonk példány.

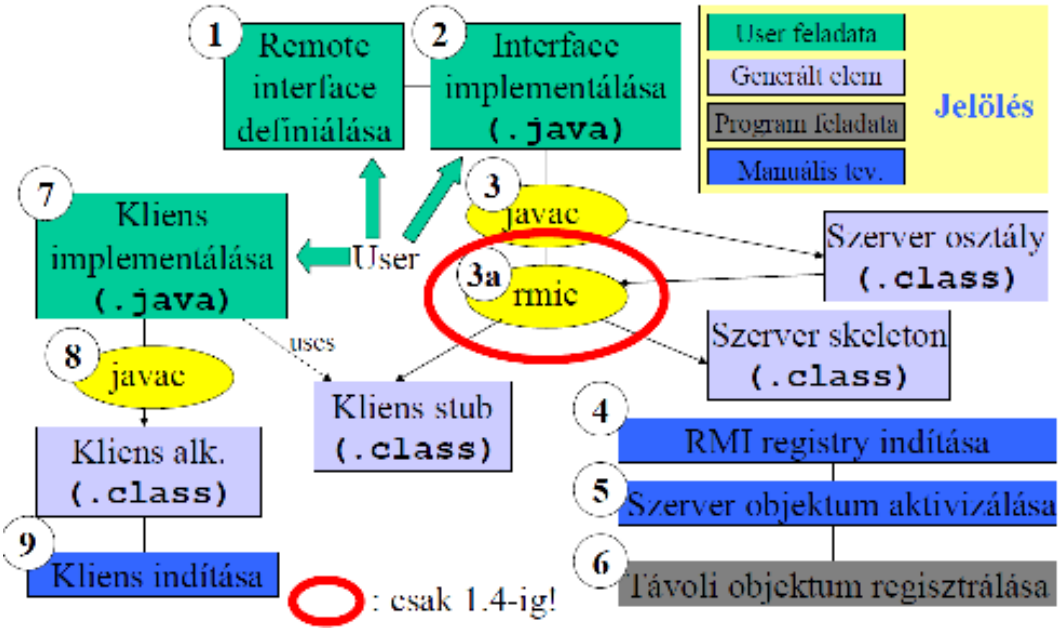
Fontos kérdések még az RMI-vel kapcsolatban a biztonsági feltételek, lévén szó hálózati kommunikációról, illetve az elosztott szemétgyűjtés megvalósítása.

## 4. Példaprogram

A következőkben egy egyszerű példaprogramon keresztül mutatjuk be az RMI megvalósítását.

### 4.1 A fejlesztés lépései

Az alábbi ábra a kliens és szerver fejlesztésének lépéseit mutatja be vázlatosan.




**2.ábra: RMI kliens és szerver fejlesztése (Forrás: Ficsor Lajos – Java RMI)**

**Lépések:**

1. **Távoli interface definiálása:**

A távol objektum interface-ének definiálása Java interface-ként. Ehhez a *java.rmi.Remote* interface kiterjesztése szükséges, valamint minden metódusnak dobnia kell egy *java.rmi.RemoteException* kivételt.

Pl.:
```
public interface TestServer extends java.rmi.Remote
{
}
```

2. **Távoli interface implementálása:**
   Az első lépésben definiált interface-t implementálni kell. A Java 1.5 verziók esetén a *java.rmi.UnicastRemoteObject* osztály leszármazottja kell legyen. Az 5. verziótól kezdve ez nem szükséges már. Ezután a szerver osztály példányosítani kell a *main()* metódusban, amely történhet ugyanabban az osztályban, vagy akár egy tetszőleges más osztályban is. Végül a szerver objektumot exportálni kell. A leszármazott változat esetén ez nem szükséges.

3. **A szerver osztály lefordítása**

3a. **A stub compiler lefuttatása (csak 5. előtti verziók esetén):**
   A csonk lefordítása az ***rmic*** parancs segítségével történik, amely paraméterei megegyeznek a *javac*-al. Ezen alkalmazás feladata a kliens csonk és a szerver oldali csonk (skeleton) generálása. A Java 1.5 verzióktól ez szintén nem szükséges.

4. **RMI registry indítása:**
   **rmiregistry** parancs indítása a szerver gépen, amely lehetővé teszi a távoli objektumok név szerinti elérését. Ez maga is távoli objektumként van implementálva. Minden szerver processz használhat saját registry-t vagy akár egy közöset is. A registry kezdetben üres, a távoli objektumokat a szervernek kell bejegyeznie.

5. **Szerver processz indítása:**
   A szerver processz létrehozza a szerver objektum példányát vagy példányait.

6. **Távoli objektumok regisztrációja:**
   A szerver processz bejegyzi a távoli objektumokat a registry-be. Ehhez felhasználja a *java.rmi.Naming* osztály metódusait. A szerver ezzel képes fogadni a kliensek kapcsolódását.

7. **Kliens kód megírása:**
   A távoli objektum eléréséhez használni kell *java.rmi.Naming* osztály metódusait. A távoli objektum használata ezután ugyanolyan, mint a helyi objektumé.

8. **Kliens kód lefordítása**
9. **Kliens indítása**

## 4.2 Hello World RMI

Távoli interface definiálása:

```
package hello;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Hello extends Remote {
        String sayHello() throws RemoteException;
}
```

A távoli interface implementálása és a szerver osztály: A kiszolgálót implementáló osztálynak egy speciális osztályból, a *java.rmi.server.UnicastRemoteObject*-ből kell leszármaznia és természetesen meg kell valósítania a távoli hívások interfészét.

```
package hello;

        import java.rmi.Naming;
        import java.rmi.RemoteException;
        import java.rmi.RMISecurityManager;
        import java.rmi.server.UnicastRemoteObject;

        public class HelloImpl extends UnicastRemoteObject implements Hello {

            public HelloImpl() throws RemoteException {
                    super();
            }

            public String sayHello() {
                    return "Hello World!";
            }

            public static void main(String args[]) {

                    // Create and install a security manager
                    if (System.getSecurityManager() == null) {
                        System.setSecurityManager(new RMISecurityManager());
                    }

                    try {
                        HelloImpl obj = new HelloImpl();

                        // Bind this object instance to the name "HelloServer"
                        Naming.rebind("//myhost/HelloServer", obj);

                        System.out.println("HelloServer bound in registry");
                    } catch (Exception e) {
                        System.out.println("HelloImpl err: " + e.getMessage());
                        e.printStackTrace();
                    }
            }
        }
```

A távoli objektumot használó kliens applet:

```
package examples.hello;

import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class HelloApplet extends Applet {

   String message = "blank";

   // "obj" is the identifier that we'll use to refer
   // to the remote object that implements the "Hello"
   // interface
   Hello obj = null;

   public void init() {
      try {
         obj = (Hello)Naming.lookup("//" +
            getCodeBase().getHost() + "/HelloServer");
         message = obj.sayHello();
      } catch (Exception e) {
         System.out.println("HelloApplet exception: " + e.getMessage());
         e.printStackTrace();
      }
   }

   public void paint(Graphics g) {
      g.drawString(message, 25, 50);
   }
}
```

## Forrásmunkák:


1. **Viczián István:** Alkalmazásfejlesztés Java-ban.
2. **Ficsor Lajos:** Java RMI áttekintés.
3. **java.sun.com:** Java Remote Method Invocation - Distributed Computing for Java: http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp.