RMI

Az alapoktól a részletekig

RMI, activation, class loader-ek, serialization

Tartalomjegyzék

BEVEZETÉS

Mi az az RMI?

Az RMI (Remote Method Invocation, Távoli Metódus Hívás) egy olyan Java technika, mely lehetővé teszi más VM-ekben — és így más számítógépeken — lévő objektumok metódusainak meghívását, ugyan úgy, mint ha azok csak szokványos helyi objektumok metódusai lennének. A két VM hálózaton keresztül kommunikálhat.

Az RMI kihasználja a Jáva architektúra és platform függetlenségét, és szükség esetén kódletöltést is végez. Pl., ha egy távoli metódushívás paramétereként olyan objektumot adunk meg, aminek osztálya nem elérhető a hívott VM-ben, akkor az automatikusan letöltheti az osztályt a hívó fél által megadott URL-ról.

Mi kell a dokumentum használatához?

A dokumentum feltételezi a Java nyelv alapos ismeretét, általános programozói ismereteket, alapvető Internetes ismeretket (IP, TCP, port), de nem feltételez semmilyen jártasságot az elosztott rendszerek készítésének területén.

Ez a dokumentum nem referencia jellegű, hanem inkább magyarázó. Ezért az olvasásához szükséges az API specifikáció is (amit pl. a java.sun.com-ról lehet letölteni).

A dokumentum Sun JDK 1.2 Standard Edition használatát feltételezi.

A dokumentum felépítése, használata

A dokumentum folyamatos, elejétől a végéig való olvasásra van kitalálva. A fejezetek feltételezik, hogy az előző fejezetek tartalma már világos.

A dokumentum az RMI-t annak Sun JDK 1.2-ben lévő megvalósításán keresztül mutatja be, és elég sok implementáció specifikus részletről ír, melyeknek megjegyzése hosszú távon teljesen felesleges. Csak azért választottam az RMI megismerésének ezt a módját, mert — számomra legalábbis — könnyebb egy rendszer viselkedését egy konkrét, működő implementáción keresztül megismerni.

A leírásban sok apró hazugság (egyszerűsítés) található, de ezekre a későbbi részekben általában fény derül.

A példákkal kapcsolatban annyit, hogy általában először a körülményesebb, de a működést jobban megmutató módszereket alkalmaztam bennük, de a fejezet végén mindig megmutatom az egyszerűbb megoldást.

Dokumentum konvenciók

A dőltbetűs részek kevésbé fontos szövegrészeket, megjegyzéseket jelölnek.

Ha egy angol szakkifejezést, aminek nincs széles körben elfogadott Magyar megfelelője, magyarosítottam, annak legelső használata után zárójelben szerepel az eredeti angol szakkifejezés.

A dokumentumban szereplő magyarosítások:

- osztály: class

- teljes név (osztályé): full qualified name
- alosztály: subclass
- konstruktor: constructor
- metódus: method
- mező: field
- tag: member:
- szerializálás: serialization
- deszerializálás: deserialization
- törpe kenguru: wallaby
- osztály betöltő: class loader
- definiáló osztálybetöltő: defining class loader
- kiindulási osztálybetöltő: initiating class loader
- csontváz: skeleton
- bináris alak (osztályé vagy interfészé): binary representation
- kivétel: exception
- dob (kivételt): throw, raise
- interfész (Java nyelvben): interface

A dokumentumban a következő szavak azonos jelentéssel szerepelnek:

- objektum, példány
- X osztály példánya, X példány
- távoli objektum, távolról elérhető objektum, távoli elérésre felkészített objektum
- távoli referencia, távoli objektumra mutató referencia, stub
- VM, JVM, Java Virtual Machine, Jáva virtuális gép, virtuális gép
- csonk, stub, kliens oldali csonk
- csontváz, skeleton, szerver oldali csonk

1. NÉHÁNY DOLOG, AMIT AZ RMI-HEZ ISMERNI KELL

Ez a fejezet olyan témákat tárgyal, melyek nem tartoznak szorosan az RMI-hez, de alapvető ismeretünk nélkül az RMI működését nem lehet megérteni.

1.1 Röviden az osztály betöltésről

1.1.1 Dinamikus betöltés és linkelés

A tipikus C/C++ programoknál[1] a kész, lefordított modulokat[2] (.obj) statikus linkeléssel[3] állítjuk össze kész futtatható állománnyá (pl. exe). Így, ha bármelyik modult megváltoztatjuk, a programot teljesen újra kell linkelni. Ráadásul, ha modul1 használta modul2-t és modul2 megváltozott, gyakran nem elég modul2 újrafordítása és az újbóli linkelés, hanem újra kellet fordítani modul1-et is, különben a program hibásan működhet. (Pontosabban: ha modul2 és modul1 forráskódja átfedett egy modul2-höz készült header-nél, és az megváltozott, akkor ezzel modul1 forráskódja is megváltozott.) Azaz a modulok gyakorlatilag nem teljesen elszeparált, önálló egységek. De szerencsére a fejlesztő eszközök mindezekre figyelnek helyettünk, és szükség esetén elvégezték a szükséges újrafordításokat és linkelést. Ez a módszer mindaddig jól működik, míg olyan alkalmazásokat készítünk, amik önálló zárt egységet képeznek, így a modulokat jól kézben tudjuk tartani. Ha ez nem így van (pl. az operációs rendszer és az alkalmazások kapcsolatánál), akkor jönnek be a képbe a dinamikusan linkelhető könyvtárak (Windows-ban DLL) meg még sok egyéb más technika.

A Jáva megközelítése a fentiektől alapvetően különbözik. Hagyományos értelemben vett linkelésről nem lehet beszélni. Egyszerűen van egy csomó különálló osztály (tipikusan külön-külön class fájlokban), melyek használhatják egymást. Ha az egyiket megváltoztatjuk, akkor csak azt az egyet kell újrafordítani és kész (feltéve, hogy az osztály interfésze nem változott meg inkompatibilis módon). És linkelni sem kell. Ez óriási előny, ha az osztályt, amit frissítünk, sok Jáva alkalmazás használja, pláne ha ezek közül sok hálózaton keresztül tölti le a class fájlt, és így kívül esik hatókörünkön. **Az osztályok betöltése futás időben történik**, méghozzá akkor, **mikor először szükség van rájuk**[4]. Azaz, X class akkor kerül betöltésre és lesz beillesztve a futtató környezetbe, mikor az alkalmazás futása során először, olyan részre ér, ahol szükség van X osztályra. És a feloldatlan **szimbolikus utalások is futás-**

---

[1] Hogy ne érje szó a ház elejét: Nem szeretném összehasonlítani a C++-t és Java-t, mivel nem is lehet. Ez a fejezet nem erről szól. Mindkettőnek megvan a létjogosultsága. Reáadásul, ha precízek akarunk lenni: a C++ csak egy nyelv, míg a Java egy platform (Java platform: JVM+API) plusz egy nyelv (Java language) plusz az ezekhez kapcsolódó néhány egyéb technika összességét takaró védjegy.

[2] Modul, itt: egy vagy több forrásfájl, ami lefordítható egyetlen obj-á, de általában feloldatlan utalásokat tartalmazhat más modulokban lévő eljárásokra, változókra. A kész futtatható állomány több lefordított modul összekapcsolásával (linkelés) jön létre.

[3] Linkelés, itt: pl. van két modul: M1 és M2. M1 használja a P eljárást, ami M2-ben található. Mikor M1-et lefordítjuk, a lefordított állomány a P hívásoknál P konkrét címe helyett csak egy szimbolikus információt tartalmaz: „ide kell majd behelyettesíteni valami P nevű izé címét”. Aztán egy linker programmal M1-et és M2-t összeragasszuk mondjuk egy futtatható állománnyá. A linker megleli M2-ben a P nevű eljárást, és M1-ben a szimbolikus utalások helyére beírja P konkrét címét az összelinkelt állományban. Ez csak ebben a végső fázisban lehetséges, mert eddig nem lehetett tudni, hogy P hova fog esni a kész programban, hiszen az M1 fordításakor a compiler azt se tudhatta, hogy egyáltalán honnan fog majd előkerülni a P.

[4] Legkésőbb akkor. Lehet, hogy a VM bizonyos esetekben előre betölt osztályokat, és pont az RMI esetén gyakran ez történik. De ez most részletkérdés.

időben kerülnek feloldásra, azaz mikor a VM futtatás közben belebotlik egybe, feloldja. Ennek persze van egy hátulütője: lehet, hogy csak futásidőben fog kiderülni, hogy egy szükséges osztály nem áll rendelkezésre, vagy nincs is olyan metódusa, amit hívni szerettünk volna, vagy egyéb probléma van vele. Szerencsére ezt a hibalehetőséget a Java fordítók általában ellenőrzik, csak hát nem biztos, hogy a futtatás pontosan olyan környezetben történik majd, mint a fordítás.

1.1.2 Osztályok vizsgálata futásidőben

Jávában **futás időben lekérdezhetők a betöltött osztályok adatai, java.lang.Class** osztályú objektumokon keresztül. Egy java.lang.Class példány – kicsit paradox módon – egy osztályt jelképez. Metódusai segítségével megtudhatjuk, hogy az osztálynak mi a neve, melyik osztályból van leszármaztatva, milyen mezők és metódusok szerepelnek benne, milyen interfészeket implementál, stb.:

```
// Kérünk egy a java.util.ArrayList osztályát jelképező
// Class példányt
Class cl = Class.forName("java.util.ArrayList");

// Kiirjuk az osztály publikus metódusait: */
java.lang.reflect.Method[] m = cl.getMethods();
for (int i = 0; i < m.length; i++) {

    // Vissztérési típus és név kírása
    System.out.print(m[i].getReturnType() + ' '
                     + m[i].getName() + '(');

    // Paraméter típusok kírása
    Class[] p = m[i].getParameterTypes();
    for (int x = 0; x < p.length; x++) {
        if (x != 0) System.out.print(", ");
        System.out.print(p[x].getName());
    }
    System.out.println(')');
}

// Csinálunk belőle egy példányt, a paraméter nélküli
// konstruktorát használva (Tudom, new-el egyszerubb lett volna,
// de tegyuk fel nem tudjuk előre, milyen osztalyt tartalmaz cl.)
Object newobj = cl.newInstance();
```

Az osztályok futás idejű vizsgálatát és az egyéb ezzel kapcsolatos műveleteket **reflection**-nek nevezik.

Továbbá, futás időben lekérdezhető egy objektum osztálya, azaz megkaphatjuk az osztályt jelképező java.lang.Class példányt:

```
// Lekéredzzük, hogy obj épp milyen osztály példányára taralmaz
// referenciát (tehát nem az számít, hogy hogy deklaráltuk obj-t)
Class cl = obj.getClass();

// Kiírjuk az osztály teljes nevét
System.out.println(cl.getName());
```

1.1.3 Osztály-betöltők

Az osztályok betöltését úgynevezett osztály-betöltők (class loader) végzik. Egy osztály-betöltő elsődleges feladata, hogy egy magadott **teljes név**[5] **alapján megszerezze az osztály vagy interfész bináris**

[5] Teljes név, vagy full qualified név: ami a package nevét is tartalmazza, pl. java.io.OutputStream.

alakját[6] (binary representation). Magyarul, hogy megtalálja, hogy hol van a szükséges class-fájl, és úgy ahogy van betöltse azt a memóriába. Illetve, nem biztos, hogy fájlt fog keresni, lehet hogy hálózatról tölti majd le a bináris alakot, vagy esetleg ott helyben generálja le azt.

A kezdetek kezdetétől működik egy úgynevezett bootstrap class loader, ez tölti be az alapvető osztályokat. Ezen kívül létezhetnek **custom class loader-ek**, ezek **java.lang.ClassLoader leszármazottak**. Ilyeneket akár mi is írhatunk, egyáltalán nem bonyolult. Szinte csak annyi az egész, hogy be kell tölteni egy byte tömbbe a bináris alakot. Az hogy a bináris alakot milyen forrásból szerezte, az osztály-betöltő dolga, és ez a lényeg.

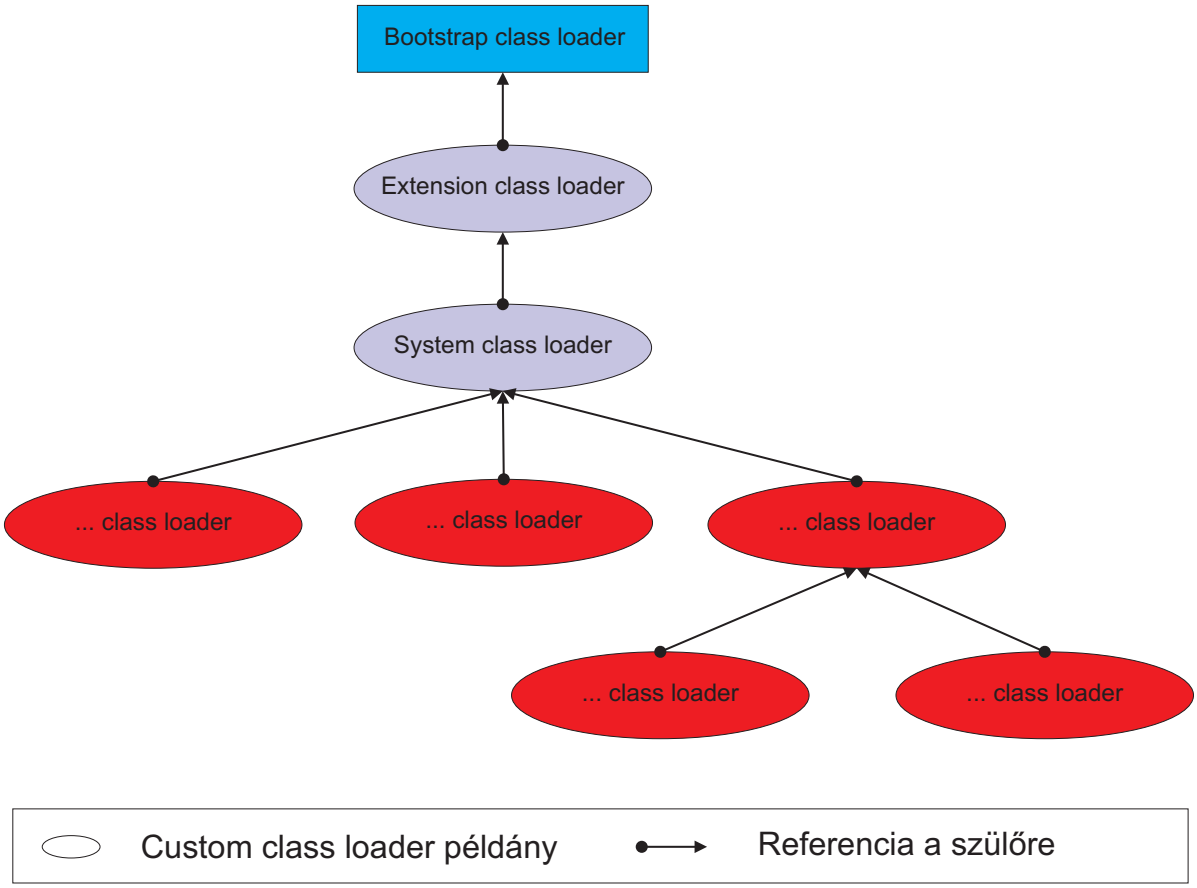
Mikor a JVM be akar töltetni egy osztályt egy custom class loader-el (osztályt vagy interfész, az osztály-betöltők szempontjából egyre megy, ezért ezentúl csak osztályt írok), meghívja annak loadClass metódusát és paraméterként a betöltendő osztály teljes nevét (ez egy String) adja meg. A loadClass-nak visszatérési értékként egy a betöltött osztályt reprezentáló java.lang.Class példányt kell visszaadnia, amit a ClassLoader.defineClass metódussal tud előállíttatni az osztály bináris alakjából (egy byte tömbből), vagy java.lang.ClassNotFoundException-t kell dobnia. Azt az osztály-betöltőt, amelyik X osztály betöltetésekor a **defineClass-t meghívta, X osztály definiáló osztály-betöltőjé**nek hívjuk. (Gyakorlatban a defineClass hívója és a bináris alak beolvasója ugyan az az osztály-betöltő.)

A ClassLoader példányok általában[7] fa topológiába vannak rendeződve: minden ClassLoader példány egy másik ClassLoader példányt (és nem osztályt) vall szülőjének (ClassLoader.getParent). Tehát most nem öröklési hierarchiáról van szó! **A tipikus ClassLoader implementáció loadClass metódusa először a szülője loadClass metódusát hívja** meg. Ha a szülő nem járt sikerrel, csak akkor próbálkozik ő maga a bináris alak beolvasásával. Így nem biztos, hogy az lesz X osztály definiáló osztálybetöltője, akinek loadClass metódusát a VM X osztály betöltésért meghívta. Azt az osztály-betöltőt, akit **X betöltésére a VM** közvetlenül **hívott, kiindulási (initiating) osztály-betöltő**nek nevezzük.

---

[6] Bináris alak: Egy az egyben, nyers byte sorozatként, ami egy class fájlban van. A bináris alak formátumát a JVM specifikáció tárgyalja teljes részletességgel.

[7] Nem csak fa topológia lehetséges, lehet pl. erdő (több független fa) is.



1.1-1. ábra: Osztály-betöltők egy lehetséges fa topológiája.

A fenti ábra egy lehetséges topológiát mutat. A bootstrap osztály-betöltő tölti be az alapvető Jáva osztályokat, melyeket a JDK-hoz alapban megkapunk. Az extension osztály-betöltő tölti be a telepített extension-okban[8] lévő osztályokat. A System osztály-betöltő tölti be a CLASSPATH alapján megtalálható osztályokat. A többi osztály-betöltő általában közvetve vagy közvetlenül a system osztály-betöltő gyereke szokott lenni. Mivel a bootstrap kivételével mindenki először a szülőjével próbálja betöltetni az osztályt, **a szülő osztálybetöltőknek prioritása van a gyerekeikkel szemben**. Ha pl., valamelyik pirossal jelölt osztálybetöltő lesz megbízva egy X nevű osztály betöltésével, de X-et a system osztály-betöltő megtalálja, akkor a system osztálybetöltő lesz a definiáló osztálybetöltő, és a piros csak a kiindulási osztályletöltő

Minden osztályhoz megjegyzésre kerül, hogy ki volt a kiindulási és ki volt a definiáló osztálybetöltője. Mikor a VM a program végrehajtása során, X osztályon belül olyan helyre ér, ahol szükség lenne egy Y nevű osztályra, akkor X definiáló osztály-betöltőjét hívja meg az Y nevű osztályt betöltésére. Tehát **egy Y nevű osztály betöltésével annak az X osztálynak a definiáló osztály-betöltője lesz megbízva (mint kiindulási osztálybetöltő), akiben az Y nevű osztály használva van**.

Két osztályt akkor tekint a VM **egyező**nek**, ha teljes nevük és definiáló osztály-betöltőjük is megegyezik**. Ha pl. Y nevű osztályt L_A és L_B osztály-betöltő is betölt, akkor a két Y nevű osztályt teljesen különbözőnek tekinti a VM. Vegyük úgy, mintha a két név nem is egyezne, mintha az egyik neve L_A:Y a másiké meg L_B:Y lenne. Pl., ha egy L_A:Y példányt L_B:Y-á akarunk cast-olni[9], az a VM szemében ugyan olyan képtelenség, mintha mondjuk egy java.util.Date-t java.io.Socket-é akarnánk

[8] Extension: Utólag telepíthető bővítés. Osztályok gyűjteménye, JAR formátumban. A JAR fájlt a Sun JDK 1.2 jre/lib/ext könyvtárába kell másolni, az extension class loader az ott lévő JAR fájlokban keresi majd az osztályokat.

[9] cast-olás: Pl. str = (String) obj. Itt String-é cast-olunk.

cast-olni. Ez nem értelmetlen dolog, mert lehet, hogy L_A és L_B máshonnan töltik le a bináris alakokat, és így a két letöltött Y-ban csak a nevük egyforma.

Példa

Nézzünk egy konkrét példát: Legyen org.xyz.ClassA osztály őse a java.lang.Object. A org.xyz.ClassA-ban használva van org.xyz.ClassB és com.foo.ClassC. Az org.xyz.* osztályokat nem érjük el helyben, ezeket a http://www.xyz.org/class/-ról tölthetők csak el. A com.foo.* class-ok viszont helyben elérhetők, benne vannak a CLASSPATH-ban.:

```
/* Oszály-betöltő példány készítése, jelen esetben egy
 * URLClassLoader. Ez a megadott URL-okról tud letölteni
 * class fájlokat.
 * A konsturktor a system class loader-t állítja majd be szülőnek,
 * mert nem adtunk meg szülőt:
 */
ClassLoader ucl = new java.net.URLClassLoader(
        new URL[]{new URL("http://www.xyz.org/class/")});

/* „Kézzel” betöltetjük az osztályt. A bináris alakot az
 * URLClassLoader a www.xyz.org/class/org/xyz/ClassA.class URL-ról
 * fogja letőlteni:
 */
Class c = ucl.loadClass("org.xyz.ClassA");

// Csinálunk egy ClassA példányt
Object obj = c.newInstance();
```

Minek mi lesz a definiáló osztály-betöltője?

- ClassA-nak ucl (az URLClassLoader példányunk). Az URLClassLoader loadClass metódusa először meghívja a szülője (a system class loader) loadClass metódusát a "org.xyz.ClassA" paraméterrel. Hasonlóképpen a system class loader is meghívja a szülőjét és így tovább, le egészen a bootstrap loader-ig. Miután ezek nem találják az osztályt, ucl fogja azt letölteni a www.xyz.org/class/org/xyz/ClassA.class URL-ról. Így aztán, a bináris alakot tartalmazó byte tömböt ő alakította át Class-á egy defineClass hívással, ő lesz a ClassA definiáló osztály-betöltője, és egyben a további ClassA-ban szereplő osztályok kiindulási osztály-betöltője.

- Object-nek letöltésével is az ucl-t fogja megbízni a VM, mert ClassA őse. Az, a fent leírtakkal megegyezően továbbítja a hívást a system class loader-nek, és még az is továbbadja a hívást… stb., és végül kiderül, hogy az Object betöltője a bootstrap class loader, és már különben is be volt töltve. A java.lang.Object definiáló betöltője a bootstrap class loader.

- Mikor a VM találkozik ClassB-vel ClassA-ban, ClassA definiáló osztálybetöltőét hívja, és innentől ugyan az a történet, mint ClassA betöltésénél. Tehát ClassB definiáló betöltője is ucl lesz.

- Mikor a VM találkozik ClassA-ban a ClassC-vel, ucl-t hívja, ami szokás szerint továbbítja a hívást a system classloader-nek. Az viszont megtalálja a ClassC-t, így ő lesz annak definiáló osztály-betöltője. Ezért, a ClassC-n belül használt további osztályok betöltése miatt már a system class loader-t fogja hívni a VM, mint kiinduló osztály-betöltőt.

Tehát a fentiekben vázolt mechanizmusok azért jók, mert így:

- Megtalálja az osztályokat: ClassB letöltésével a VM automatikusan a megfelelő osztályletöltőt bízta meg. (Ha www.xyz.org-ék szolgáltatják ClassA-t, akkor bizonyára szolgáltatják a hozzá szükséges ClassB-t is, ami nekünk nincs meg helyben)

- Nincs felesleges letöltés: Csak az a két osztály lett az ucl-el letöltve, amit muszáj volt letölteni. Ahol lehetett, helyben elérhető osztály használtunk.

- Lehetőleg nincs keveredés az egyforma nevek miatt: Az a ClassC ami a ClassA-ban szerepelt így ugyan az a ClassC lett, mint amit mi eddig is helyben használtunk, mivel a definiáló osztály-letöltője a system class loader lett. Nem kell azzal szenvedni, hogy a ClassA-ban található ClassC az egy másik ClassC, csak a teljes nevük ugyan az. Ugyan ez vonatkozik az Object-re is: csak egy java.lang.Object nevű osztály van, amit a bootstrap class loader töltött be.

A fenti példában, nem írhatjuk azt, hogy: „org.xyz.ClassA obj = (org.xyz.ClassA)ucl.newInstance();”, mivel a javac nem lenne hajlandó lefordítani, mert nem találná org.xyz.ClassA-t. És ha még le is fordíttatnák, ha a programsort tartalmazó osztály betöltője a system class loader, arra késztetnénk a VM-et hogy a system class-loderrel töltesse be az osztályt, ami nem sikerülne. De akkor, hogy hívjuk meg obj azon metódusait, amiket nem a java.lang.Object-től örökölt?:

- A(z) „1.1.2 Osztályok vizsgálata futásid” fejezet végén felvillantott reflection-nal futás időben megkereshetnénk a metódusokat, és meg is hívhatnánk azokat. De ez elég körülményes lenne.

- ClassA implementáljon egy X interface-t, és X bináris alakja legyen helyben[10] elérhető. Így X-é már átcastolhatjuk az új példányt: „X obj = (X)ucl.newInstance();”. Így a ClassA felhasználóinak csak X kell, hogy meglegyen, ami sokkal ritkábban változik, mint maga az implementáció, azaz ClassA. És ha X változik is, akkor bizonyára csak bővül, így a rég elavult X még ugyanúgy használható az új ClassA-hoz, legfeljebb nem érjük el vele az új funkciókat.

URLClassLoader

Az RMI nagyon gyakran használ java.net.URLClassLoader-eket. Az URLClassLoader a konstruktorában megadott URL-okat használja a class fájl letöltésére, ha a szülője azt nem találta.

A megadott URL-okat az elsőtől kezdve sorban végig próbálja, míg meg nem találja a betöltendő osztályt, vagy végig nem próbálta az összes URL-t.

Ha egy URL **‘/’ karakterrel végződik**, akkor azt **bázis könyvtár**nak értelmezi. Pl.: ha az URL „http://x.org/class/” és a keresett osztály neve „com.foo.A”, akkor „http://x.org/class/com/foo/A.class”-t próbál letölteni. Ha az URL **nem ‘/’ karakterrel végződik**, akkor úgy veszi, hogy egy **JAR fájl**t adtunk meg, ami az osztályt tartalmazza.

Az API specifikáció nem tárgyalja, hogy milyen protokollokat használhatunk, de a HTTP, FTP és a FILE protokoll a Sun implementációban támogatott.

Ha az URLClassLoader konstruktorának nem adunk meg szülőt, akkor szülője a system class loader lesz.

1.1.4 Összefoglalás

Szerencsére az itt leírt folyamatokba általában nem kell belegondolnunk, mikor alkalmazásokat fejlesztünk. Ami a lényeg:

- A Jáva futásidőben linkel, és ez nagy rugalmasságot ad neki. A feloldatlan hivatkozások tisztán szimbolikusak.

- Lehet alternatív forrásokból (tipikusan hálózat) class-t letölteni futásidőben

[10] Pontosabban a javac által elérhető. Ez gyakorlatilag azt jelenti, hogy vagy a bootstrap, vagy az extension vagy a system class loader megtalálná.

- Ha egy osztály valamilyen „alternatív” forrásból lett letöltve, akkor általában:
  - Az osztályban szereplő további osztályok szükség esetén automatikusan letöltődnek ugyan azt az alternatív forrást használva
  - Ha egy a letöltött osztályban szereplő további osztály helyben is elérhető, akkor a helyben elérhető verzió lesz felhasználva
- A távolról letöltött osztályok példányait általában egy helyben elérhető interface-on keresztül érjük el, amit a letöltött osztály implementál.

1.2 Röviden a serialization-ról

1.2.1 Mi az a serialization?

A serialization (továbbiakban szerializálás) **objektumok byte sorozattá alakítása**. Ennek ellenkezője a deserialization (továbbiakban deszerializálás) **byte sorozat objektummá visszaalakítása**. A szerializálás szó tágabb értelemben a technikát jelenti, ami magában foglalja a szerializálás és a deszerializálás műveletét is.

A szerializálás lehetővé teszi:

- Objektumok mozgatását független VM-ek közt. Pl. egy objektumot szerializálással byte folyammá alakítunk és átküldjük az Interneten egy másik VM-nek, aki a beolvasott byte folyamot deszerializálással visszaalakítja objektummá. Az objektumot átküldtük egy másik VM-be. Elsősorban ez az, ami miatt a szerializálásnak fontos szerep jut az RMI-ben.
- Objektumok életének meghosszabbítását a VM életén túl. Pl. létrehozunk egy objektumot, és byte folyammá alakítva kiírjuk egy fájlba. A fájlt 30 év múlva elővesszük, és deszerializálással visszaalakítjuk egy objektummá. Íme ott az objektum, abban az állapotban, ahogy 30 éve „hibernáltuk”.

1.2.2 Hogy lehet (de)szerializálni?

A **java.io.ObjectOutputStream** és a **ObjectInputStream** osztályok teszik lehetővé a (de)szerializálást. Ezek OutputStream és InputStream leszármazottak és implementálják a DataOutput ill. DataInput interface-okat (lásd java.sun.com-on az API specifikációban és Java Tutorial/Essential Java Classes/Reading and Writing-ben). Az ObjectOutputStream pluszban tartalmaz egy **writeObject**, míg az ObjectInputStream egy **readObject** metódust.

A lenti példa kiírja az obj objektumot a lista.ser fájlba (szerializálás):

```
// Mondjuk legyen egy java.util.ArrayList példány
ArrayList obj = new ArrayList();
obj.add("Ezt most csak azért,");
obj.add("hogy legyen miért kimenteni.");

// Létrehozzuk ObjectOutputStream-et
FileOutputStream fout = new FileOutputStream("lista.ser");
ObjectOutputStream oout = new ObjectOutputStream(fout);

// Kiírjuk a fájlba
oout.writeObject(obj);
oout.close();
```

És a visszaolvasás (deszerializálás):

```
// Előállítjuk az ObjectInputStream-t
FileInputStream fin = new FileInputStream("lista.ser");
```

```
ObjectInputStream oin = new ObjectInputStream(fin);

// Visszaolvassuk a fájlból
ArrayList obj = (ArrayList) oin.readObject();
oin.close();

// Ezt csak úgy ellenörzésképp
System.out.println(obj.get(0));
System.out.println(obj.get(1));
```

1.2.3 Mi kerül a byte folyamba?

A szerializáláskor kiírt byte folyamba (pl. a fenti példa lista.ser fájlába) a következő információk kerülnek bele:

- A kiírt objektum osztályáról információk, ezek:
  - Az osztály teljes neve, deszerializáláskor majd ilyen nevű osztály példányát kell létrehozni
  - A kiír objektum osztályából generált 64 bites kód (un. serialVersionUID), deszerializáláskori kompatibilitás ellenőrzésre (lásd később). Ez egyfajta hash-kód, ami az osztály kompatibilitás szempontjából fontos jellemzői alapján van előállítva.
  - A kimentett mezők (lásd a következő pontban) típusa és neve, ez majd kell deszerializáláskor, hogy a kimentett értékeket az új objektum megfelelő mezőibe írjuk
- Az objektumban szereplő mezők értéke, de csak a nem-statikus és nem-transient mezőké
- A fenti információk minden olyan példányról, amire az objektum valamely mezője referenciát tartalmazott, és így tovább rekurzívan (pl. fenti példában 3 objektum került kiírásra: az ArrayList példány és a két String példány, amit tárolt).

Tehát **nem került kiírásra maga az osztály definíció** (pl. a metódusok törzse), csak az osztály azonosításához szükséges információ, és a mezők pillanatnyi értéke (ami tulajdonképp az objektum állapota). Ezért a deszerializálásnál majd rendelkezésre kell állnia az osztálynak, különben az objektum nem hozható létre és java.lang.ClassNotFoundException lép fel.

Pontosabban, **nem muszáj pont annak az osztálynak rendelkezésre állnia, ami a szerializáláskor használatban volt, hanem megfelel egy azonos nevű kompatíbilis osztály is**. Ez gyakorlatilag a class egy másik verziója (régebbi vagy újabb) szokott lenni. A Sun JDK 1.2-ben lévő szerializáció nem tekint két osztályt kompatibilisnek, pl. ha:

- Nem pont ugyan azok a nem-statikus mezők vannak bennük ($\Rightarrow$ az állapot nem visszaállítható)
- Nem pont ugyan azok a nem-private metódusok vannak bennük ($\Rightarrow$ különböző az interfészük). Itt csak a metódusok nevének, a paraméterek típusának és visszatérési érték típusának kellene egyeznie.

De két osztályt még kompatíbilisnek tekinthet attól, hogy:

- Különböző a bennük metódusok törzse ($\Rightarrow$ csak implementációs különbség)
- Nem ugyan azok a private metódusok szerepelnek bennük ($\Rightarrow$ csak implementációs különbség)
- Egy általuk implementált interface-k neve ugyan az, de taralmuk különböző (Mivel az interface-k tartalma nem szerepel a szerializált alakban, ez ki se derül)
- Nem-private metódusok más kivételeket dobnak (Azért ez számomra meglepő, hogy nem számít)

Mint e fejezet elején leírtam, **egy példány szerializálása több másik példány szerializálását is maga után vonhatja** (amikre a mezőkben referencia volt). Továbbá, egymás utáni writeObject hívásokkal

több objektumot kiírhatunk egymás után a stream-re, mielőtt lezárnánk azt (close). Ha egy ObjectOutputStream-re, többször ki lesz írva ugyan az az objektum, akkor csak az első kiíráskor lesz teljes egészében kiírva, a további estekben csak egy utalás kerül a stream-be helyette, az első kiírt példányra. Deszerializáláskor pedig, csak az első példány olvasásakor jön létre új objektum, a további előfordulásainál az első példányra mutató referenciát ad vissza a readObject. Ez igen fontos tulajdonság: **azok a referenciák, melyek szerializáláskor ugyan arra az egy objektumra mutattak, a deszerializálás után is ugyan arra az egy objektumra fognak mutatni**.

1.2.4 Mi szükséges a szerializálhatósághoz?

Egy objektum szerializálása és deszerializálása nem mindig olyan egyszerű kérdés, mint első ránézésre gondolnánk, pl. mert:

- Az objektum állapotához olyan jellemzők is hozzátartoznak, melyek nem álnak vissza a mezők értékének egyszerű visszaállításával. Gondoljunk csak egy objektumra, ami elkezdett írni egy fájlba. Mi történjen, mikor deszerializáljuk? Próbálja megnyitni a megkezdett fájl, vagy inkább ne is engedjük a szerializálását? (A FileOutputStream nem szerializálható.)

- Az objektum olyan private mezőket tartalmaz, amiknek értéke titkos, de mivel az objektum szerializáláskor ezek is kiíródnak stream-be, hozzáférhető lenne értékük.

Egy **objektum csak akkor szerializálható** (csak akkor hajlandó az ObjectOutputStream szerializálni)**, ha annak osztálya** közvetve vagy közvetlenül[11] **implementálja a java.io.Serializable interface-t**. Ez nem tartalmaz metódusokat, csak jelzésre szolgál.

Azon mezők értéke, melyeket egy szerializálható osztály olyan osztályoktól örökölt, amik még nem implementálták a Serializable interfészt, nem kerülnek szerializáláskor kiírásra. Ezért deszerializáláskor meg lesz hívva a legközelebbi nem szerializálható ős paraméter nélküli konstruktora, hogy a mezőiket inicializálhassa. A többi osztálynak, amik már implementálták a Serializable-t, nem lesz meghívva a konstruktora, ehelyett mezőik értéke a szerializáláskor kiírtakra lesz visszaállítva.

Lehetőség van arra is, hogy az **alapértelmezett szerializálási mechanizmus helyett sajátot használjunk**. Ez alapvetően kétféle módon történhet:

- Saját ObjectOutputStream és ObjectInputStream implementációt alkalmazunk. Általában, csak egy alosztályt származtatunk, és felülbíráljuk (override) az eredeti implementáció néhány metódusát (lásd: 1.2.5 fejezetben)

- A szerializálható osztályon belül olyan speciális metódusokat és mezőket helyezhetünk el, melyeket az ObjectOutputStream és ObjectInputStream figyelembe vesz. Ezzel lehetővé válik, hogy az osztály befolyásolja, sőt, akár maga implementálja példányai szerializálásának és deszerializálásának mikéntjét. Erről nem lesz szó ebben a dokumentumban, a RMI megértéséhez elég, ha tudunk erről a lehetőségről.

1.2.5 ObjectInput- és OutputStream leszármazottak

Az ObjectInputStream és az ObjectOutputStream számos olyan metódust tartalmaz, melyek alapban üresek vagy csak valami nagyon triviális műveletet végeznek, de egy leszármazott osztályban való felülbírálásukkal (override) könnyen megváltoztatható az ObjectInputStream ill. ObjectOutputStream viselkedése. Itt csak az RMI szempontjából legfontosabbak kerülnek tárgyalásra:

---

[11] Közvetve vagy közvetlenül implementálja: Ő maga vagy egy őse közvetlenül implementálja (class … implements …) a Serializable interface-t, vagy egy interface-t ami Serializable leszármazott.

ObjectOutputStream.annotateClass és ObjectInputStream.resolveClass

Az annotateClass metódus szerializáláskor, a class információk stream-re való kiírása után, de még a mezők értékének kiírása előtt hívódik meg. Itt plusz információkat írhatunk ki a stream-re, mint például egy URL-t, ahonnan a class fájl letöltöttük. Pl.:

```
package com.foo.example.*;

import java.io.*;
import java.net.*;

class MyObjectOutputStream extends ObjectOutputStream {

    public MyObjectOutputStream(OutputStream out)
           throws IOException {
        super(out);
    }

    /*
     * Kiírja a stream-ra azokat az URL-okat, miket az osztály
     * letőltésénél használtunk. (A törzset nem fontos megérteni.)
     */
    protected void annotateClass(Class oclass) throws IOException {

        // Definiáló class loader lekérdezése
        ClassLoader cl = oclass.getClassLoader();

        // Eldönti, hogy ki kell-e irní az URL-t
        boolean writeURL;
        if (!oclass.getName().startsWith("java.")
                && cl != null // bootstrap
                && cl instanceof URLClassLoader) {
            // Ha cl a system class loader vagy annak egy
            // szülője, akkor nem írunk URL-t, mert akkor
            // helyben megvan az osztály
            ClassLoader sys = ClassLoader.getSystemClassLoader();
            while (sys != null && cl != sys) {
                sys = sys.getParent();
            }
            writeURL = sys == null;
        } else {
            writeURL = false;
        }

        // Kiírja az URL-okat, ha writeURL true
        if (writeURL) {
            writeObject(((URLClassLoader) cl).getURLs());
        } else {
            writeObject(null);
        }
    }
}
```

A resolveClass deszerializáláskor a szokványos class információk (lejjebb látható ObjectStreamClass paraméterben megkapjuk) beolvasása után, de még a mezők értékének beolvasása előtt hívódik meg. Itt kell beolvasnunk az annotateClass által kiírt információkat. Ez lehet pl. egy URL, ahonnan a class fájl letölthető. A visszatérési értékének a példány létrehozásához használandó Class-nak kell lennie.

Előző példát folytatva, a resolveClass implementációnk megpróbálhatja elérni a Class-t helyben, és ha nem találja, akkor megkísérli letöltheti azt a szerializáláskor az annotateClass által kiírt URL-ok valamelyikéről. Pl.:

```
package com.foo.example.*;

import java.io.*;
import java.net.*;
import java.util.*;
import java.lang.ref.*;

public class MyObjectInputStream extends ObjectInputStream {

    private static Map classLoaderCache = new WeakHashMap();

    public MyObjectInputStream(InputStream in)
           throws StreamCorruptedException, IOException {
        super(in);
    }

    /*
     * Beolvassa a stream-ról azokat az URL-okat, miket az osztály
     * letőltésénél használtunk. (A törzset nem fontos megérteni.)
     */
    protected Class resolveClass(ObjectStreamClass osc)
              throws IOException, ClassNotFoundException {
        URL[] codebases = (URL[]) readObject();
        ClassLoader cl;

        if (codebases != null) {
            cl = getURLClassLoader(codebases);
        } else {
            cl = ClassLoader.getSystemClassLoader();
        }

        return cl.loadClass(osc.getName());
    }

    /*
     * En nem override! Ezt csak a MyInputSrteam resolveClass
     * metódusa hívja.
     */
    private static synchronized URLClassLoader
             getURLClassLoader(URL[] codebases) {
        URLClassLoader cl;

        // Megnézi, hogy van-e a classLoaderCache-ben megfelelő
        // URLClassLoader, ha nincs csinál egyet és beteszi a
        // classLoaderCache-be.
        ... Ezt most kihagyom mert hosszú és semmi köze a témához

        return cl;
    }
}
```

Az, hogy milyen plusz információt írunk ki és milyen formátumban, a mi dolgunk. A lényeg, hogy a deszerializáláskor a saját ObjectOutputStream osztályunkhoz készül ObjectInputStream osztályunkat használjuk, és így az értelmezni tudja a plusz információt.

ObjectOutputStream.replaceObject

Ez minden kiírandó objektum előtt meghívódik, még mielőtt bármit is kiírnánk róla. Paraméterként megkapjuk a kiírandó objektumot, visszatérési értékként visszaadjuk, hogy mi kerüljön helyette szerializálásra.

A lenti példa minden olyan java.net.URL-t, amiben a host „127.0.0.”-val kezdődik, lecserél egy olyan java.net.URL-ra, amiben a host www.foo.com. (Ez most csak ilyen kínomban kitalált példa…)

```
import java.net.URL;

public class MyObjectOutputStream extends ObjectOutputStream {

    public MyObjectOutputStream(OutputStream out)
           throws IOException {
        super(out);
        enableReplaceObject(true); // Különben nem fogja meghívni
    }

    protected Object replaceObject(Object obj) throws IOException {
        if (obj instanceof URL
                && ((URL)obj).getHost().startsWith("127.0.0.")) {
            // Készít egy új URL példányt
            URL url = (URL)obj;
            return new URL(url.getProtocol(),
                           "www.foo.com", url.getPort(),
                           url.getFile());
        } else {
            return obj; // Nem cseréljük le
        }
    }
}
```

2. RMI

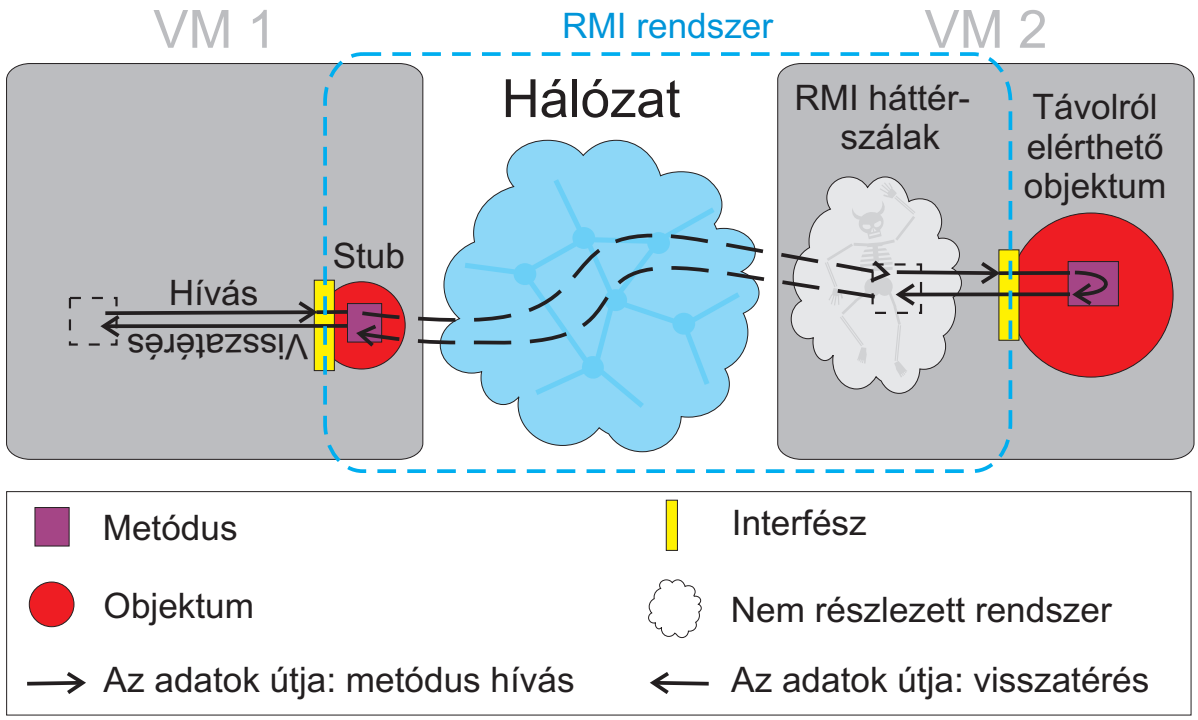
2.1 Működési elv

2.1.1 Mi az az RMI?

Az RMI lehetővé teszi olyan objektumok metódusainak meghívását, melyek másik VM-ben (és így esetleg más számítógépben) találhatók, mint a hívó. Teszi mindezt a lehetőségekhez mérten **áttetsző**-en, azaz a programozónak csak ritkán kell figyelembe vennie, hogy nem helyi, hanem távoli objektumot hív, vagy hogy a metódus hívója távoli is lehet. Úgy gondolkodhatunk, mintha—a valójában más-más VM-ben elhelyezkedő objektumok—egyetlen nagy közös VM-ben lennének.

Az RMI tulajdonképp csak egy csomó „sima”[12] java osztály gyűjteménye, melyek eltakarják a programozó elől a hálózati kommunikáció részleteit, és azt az érzetet keltik, mintha tényleg nyelvi értelembe vett metódushívás történne.

2.1.2 Alapvető működési elv



2.1-1. ábra: RMI egyszerűsített működése mechanizmusa. VM 1-ből hívjuk a VM 2-ben lévő objektum egy metódusát. A két VM hálózaton keresztül kommunikál. Ha kitakarjuk a kékkel keretezett részt („RMI rendszer”), akkor egy normál metódushívást kapunk. (Aki hiányolja őket: A skeleton-ok (szerver oldali csonkok) az „RMI háttér szálak” felhőben vannak, és szándékosan (rmic -v1.2) nincsenek kirajzolva.)

Java-ban nincs távoli referencia (értsd: távolra mutató referencia), azaz olyan referencia mely egy másik VM-ben lévő objektumot címezne meg. Ezért, a metódust hívó VM-ben egy úgynevezett stub-ot hozunk létre. Ez egy normál helyi objektum, melynek interfésze olyan, mint a távoli objektu-

[12] Én legalábbis semmi olyat nem vettem észre benne, amit ne lehetne tisztán Jávában megoldani.

mé, de nem tartalmazza a távoli objektum metódusainak törzsét. A metódushívásokat hálózaton az erre a célra kidolgozott protokollal (JRMP, Java Remote Method Protocol) továbbítja a távoli objektumhoz, majd megvárja, míg a visszatérési érték vissza nem érkezik a hálózaton, és azt visszaadja a hívónak. A hívó szemszögéből tehát ugyan úgy viselkedik, mint a távoli objektum. A stub „eljátssza" a távoli objektumot, de az interfészén és a helyén (ahol a hálózaton megtalálható) kívül semmit sem tud róla.

A stub-ra mutató referencia távoli referenciaként fogható fel. Több stub példány, más-más VM-ekben ugyan ahhoz a távoli objektumhoz továbbíthatja a hívásokat, mintha ugyan arra az objektumokra lennének referenciák.

A hívott fél VM-jében RMI-vel foglalkozó szálak rendszere fut a háttérben (az ábrán „RMI háttér szálak"), mely várja a hálózatról beérkező metódushívásokat. Mikor a hívás egy stub-tól megérkezik, meghívja a távolról elérhető objektum megfelelő metódusát (normál metódushívással, hiszen egy VM-ben van vele), majd a meghívott metódus visszatérési értékét visszaküldi a stub-nak a hálózaton keresztül. Így, a távolról meghívott objektum szemszögéből egy távoli metódushívás semmiben sem különbözik egy helyi hívástól.

Megvalósítási okokból a hívó VM-jében is futnak a háttérben RMI-vel foglalkozó háttér szálak, de ezek szerepe ezen a szinten még lényegtelen (ezért nincsenek az ábrán). A háttérszálakat mindkét oldalon valamilyen RMI-vel kapcsolatos metódushívásunk indítja el „titokban", anélkül hogy erre figyelnünk kellene.

Azt a felet, **amelyik a távoli metódust hívja kliens**nek (client), azt a felet pedig, amelyik **hívást fogadja** pedig **szerver**nek (server) szokták nevezni. Ezek az elnevezések azonban gyakran félrevezetők: RMI-ben kliensről és szerverről csak egy konkrét távoli metódushívás kapcsán beszélhetünk[13], hiszen lehet, hogy egy VM távoli metódusakt hív, de ő maga is szolgáltat távolról hívható objektumokat. Ezért sokszor, a „kliens" helyett a „hívó", a „szerver" helyett pedig „hívott" kifejezést fogom használni.

2.1.3 Mi megy át a hálózaton?

Mit kell átküldenie a stub-nak a hálón:

- A cél objektum (aminek a metódusát hívjuk) azonosításához szükséges információt
- A hívandó metódus azonosításához szükséges információt
- A paramétereket, melyekkel a metódust hívtuk

Mit kell visszaküldeni a stub-hoz a hálón:

- A metódus hívás eredményét, ami lehet:
  - Visszatérési érték (amit return-al adott vissza a metódus)
  - Kivételt, ami:
    - A távoli objektum metódusában lett dobva ill. lépett fel
    - Az RMI mechanizmusban lépett fel (egy kis átlátszatlanság)

Ezeknek pontos mikéntje részletkérdés, ami leginkább csak az RMI fejlesztőinek érdekes, vagy később lesz tárgyalva.

Említést érdemel a cél objektum azonosítása: mivel **a stub** objektum kifejezetten egy bizonyos távoli objektumhoz van létrehozva (mint ahogy egy referencia is egy bizonyos objektumra mutat), **tudja, hogy hova kell továbbítani** a hívásokat (hálózati cím, stb.).

---

[13] Persze ha egy VM kifejezetten csak távoli objektumokat szolgáltat, a másik meg szinte csak hív távoli objektumokat, akkor lehet mondani, hogy az ott a szerver ez meg itt a kliens…

Ami viszont külön figyelmet érdemel, az a **paraméterek elküldése és a visszatérési érték visszaküldése**. Mindkét esetben primitív érétkeket vagy objektumokat kell átküldeni a hálózaton. Ez a már leírt **szerializálással történik**. Az RMI a java.io.ObjectOutputStream és ObjectInputStream egy alosztályát használja, ami lehetővé teszi, hogy a deszerializáció során a fogadó VM számára nem elérhető osztályokat az letölthesse a küldő által megadott URL-ról (erről részletek majd külön fejezetben).

A szerializálással csak egy másolatot küldünk az objektumról, ezért ha a paraméterként átadott objektum a hívott fél VM-jében módosul, a hívó VM-ben a módosulás nem fog megtörténni. Ez pedig, nem felel meg a helyi metódushívás szemantikájának. Ezen a ponton a távoli eljáráshívás nem úgy viselkedik, mint a helyi, erre figyelni kell. További következmény, hogy nem serializable objektumokat nem használhatunk paraméternek.

Az átlátszatlanság oka tehát az, hogy **nem referencia szerint** („cím szerint") **küldtük át az objektum paramétert, hanem érték szerint**. A távolról elérhető objektumokra azonban létezhet látszólagos távoli referencia (mint előzőleg le lett írva), amit egy stub példány valósít meg. Ha olyan objektum példány kerül elküldésre (akár paraméterként akár visszatérési értékként), ami **fel lett készítve távoli elérésre, akkor** az RMI-nél alkalmazott ObjectOutputStream alosztály az **objektum helyett** egy ahhoz készült **stub példányt küld** el (replaceObject). Így **ilyenkor, a helyi eljáráshívás szemantikája megmarad**, a változások mindkét VM-ből látszódni fognak, hiszen minden metódushívás ugyan ahhoz az egy távolról elérhető objektumhoz lesz továbbítva (és mint később látni fogjuk, a távoli objektumot csak metódushívással érhetjük el, így mezőket közvetlenül nem érjük el). Ebből következik az is, hogy maga **a távolról elérhető objektum sohasem lesz átküldve a hálón csak a hozzá készült stub-ok**. A távolról elérhető objektum tehát nem kell, hogy szerializálható legyen, csak a hozzá készült stub.

2.1.4 Kivételek

Távoli eljáráshívásnál, mind a távoli objektum metódusában mind az RMI rendszer működése során fellépett kivételeket a **hívó VM-jében** kell lekezelnünk, nem a hívott félnél. A hívott fél programozója soha nem fog ezekkel a kivételekkel találkozni.

Ez, ha belegondolunk, logikus:

- A hívott metódusban fellépett kivételt, a helyi eljáráshívás szemantikájának megfelelően, a hívónak kell megkapnia és kezelnie. A hívó jelen esetben egy másik VM-ben van.

- Az RMI során fellépett hiba következménye, a hívás vagy a már kész eredmény visszaküldésének meghiúsulása. Ezért a hívott metódus vagy még, vagy már nem kompetens az ügyben. A hibát a hívó tudomására kell hozni. Ezek a hibák a hívó VM-ben, a hívott távoli metódus által dobott java.rmi.**RemoteException** leszármazott kivételként jelennek meg.

Az RMI-vel hívott metódust, a hívott VM-ben a háttérben futó RMI szálak egyike hívja közvetlenül, így ő kapja meg a hívott metódusban fellépett kivételt is, amit szerializálva visszaküld a stub-nak, aki a deszerializálással visszaállított kivétel példányt dobja (throw).

Az RMI hibák az RMI-vel foglalkozó kódrészeken belül jönnek létre. Ezekre a hívónak számítania kell, egy távoli metódushívás mindig dobhat java.rmi.RemoteException-t. Ennek oka, hogy nincs mód annak garantálására, hogy a távoli metódushívás közben nem lép fel hiba, pl. mert szétmegy a hálózat vagy lekapcsolják a hálózat túloldalán a számítógépet.

2.2 Gyakorlati megvalósítás

2.2.1 A távolról elérhető osztály elkészítése

A távoli interfész

A kliensek a távoli objektumot csak annak távoli interfészén keresztül érhetik el. A távoli objektum minden más implementációs részlete számukra láthatatlan marad. Azokat a metódusokat, melyek nincsenek benne ebben az interfészben, csak helyben lehet meghívni, normál metódushívással.

A „távoli interfész" kifejezés alatt ne egy konkrét Jáva nyelvi értelemben vett interface-re gondoljunk, ez egy elvont fogalom. **A távoli interfészt a távolról elérhető objektum osztálya által (közvetlenül vagy közvetve) implementált azon interface-ek alkotják, melyek (közvetlenül vagy közvetve) java.rmi.Remote leszármazottak**. Maga a java.rmi.Remote interfész üres, csak jelzésként szolgál.

A távoli interfészben szereplő minden metódusnak **kötelező a java.rmi.RemoteException** lehetőségének jelzése (throws), így a hívó programozója nem felejtkezhet meg azok lekezeléséről (checked exception).

Az implementáció osztály

Az implementáció osztály a távoli interfész egy konkrét implementációja, a távolról elérhető objektum osztálya. Ezt az osztályt ugyanúgy kell megírni, mint bármilyen más Jáva osztályt. De a következőket figyelembe kell venni:

- Csak azok a metódusok lesznek távolról elérhetők, amik a távoli interfészben is szerepelnek. A többit csak helyben hívhatjuk majd.
- A távoli interfészben szereplő metódusok paramétereinek és visszatérési értékének típusa primitív érték vagy szerializálható osztály kell, hogy legyen, vagy majd távolról elérhető objektumokat kell bennük átadni. (az okot lásd: 2.1.3-ben)
- A java.lang.object equals és hashCode metódusai nem dobnak java.rmi.RemoteException, így eleve nem lehetnek a távoli interfész tagjai, nem lehet őket távolról hívni. Mármost, a stub-ra mutató referencia elvileg egyenértékű a távoli objektumra mutató referenciával, így pl., két stub-nak ami ugyan arra az objektumra mutat, ugyan azt a hash kódot kéne visszaadnia, és „equals"-nak kellene lenniük. Ugyan ez a viszony kell, hogy fennálljon egy stub és az ahhoz tartozó távoli objektum között is (pl. obj.equals(obj_stubja)). Ennek biztosítására az implementáció osztályt mindig **java.rmi.server.RemoteObject leszármazottnak ajánlatos készíteni**, ami tartalmazza a fentieket kielégítő equals és hashCode metódusokat. (A „==" operátor még ezek után is „rosszul" fog működni: két külön stub példányra mutató referenciára false-t ad akkor is, ha azok ugyan ahhoz a távoli objektumot készültek.) Az object wait, notify és notifyAll metódusai sem működnek átlátszóan, és csak a stub-ot fogják érinteni.

A nevekkel kapcsolatban az a konvenció, hogy ha implementáció osztály távoli interfészét csak egy interface alkotja, és annak neve pl. com.foo.Valaszolo, akkor az implementációs osztályé com.foo.ValaszoloImpl.

A stub osztály

Az implementáció osztályhoz készülni kell egy stub osztálynak is, aminek példányain keresztül más VM-ből majd elérhetők lesznek az implementációs osztály példányai.

A stub osztályt nem nekünk kell megírni, hanem az **rmic** nevű programmal generáltathatjuk, a lefordított implementáció class fájlból. A generált stub osztály neve (a fenti példát alapul véve) com.foo.ValaszoloImpl_Stub lesz és a com.foo.ValaszoloImpl azon interface-eit fogja implementálni, melyek közvetve vagy közvetlenül a java.rmi.Remote interface leszármazottjai. Ezek fogják alkotni a

távoli interfészt. A távoli interfész fogalma, a stub osztállyal realizálódik: **a stub osztály az implementációs osztálynak csak a távoli interfészt alkotó interface-eit implementálja**, így a kliensek csak az ebben szereplő metódusokat tudják majd meghívni.

A távoli interfészt implementáló metódusok a stub-ban semmi mást nem tesznek, mint hálózaton továbbítják a metódushívást egy implementáció osztály példányhoz, megvárják amíg visszaérkezik a hívás eredménye, és azt visszaadják. Ezért a stub osztályok elkészítésénél csak az implementáció osztály távoli interfésze érdekes. Ha azt nem változtatjuk meg, felesleges újrageneráltatni a stub osztályt. Továbbá, ha egy implementációs osztály közvetlenül nem implementál semmilyen java.rmi.Remote leszármazott interfészt, hanem csak örökölte azokat, nem kell (és az rmic nem is hajlandó) hozzá stub-ot készíteni, csak a legközelebbi olyan őséhez, aki közvetlenül implementált. Konkrét példával: Ha AImpl osztály közvetlenül implementálja az A nevű Remote leszármazott interface-t, és Aimpl2 osztály AImpl leszármazottja és nem implementál közvetlenül egy Remote leszármazott interface-t sem, akkor az AImpl2 példányokhoz használt stub-ok is AImpl_Stub példányok lesznek. AImpl2_Stub osztályra nincs szükség.

A JDK 1.1-ben, az rmic egy un. skeleton osztályt is generált (com.foo.ValaszoloImpl_Skel). Ez a(z) 2.1-1. ábra "RMI háttér szálak" felhőbe volt használatban, és a stub hívott VM-beli párja volt. A JDK 1.2 RMI-ében már nincs erre az osztályra szükség és a stub-ok által használt protokoll is megváltozott. Ha a Sun JDK 1.2 rmic-e alapértelmezésként JDK 1.1-el és JDK 1.2-vel is egyszerre kompatíbilis stub-ot, és skeleton-t generál. Ha a távoli objektumot nem fogják elérni JDK 1.1-es kliensek, akkor az rmic -v1.2 kapcsolójával tisztán JDK 1.2-es stub-ot generáltathatunk, és nem készül skeleton class.

Összefoglalás

A távolról elérhető osztály készítéséhez a következők szükségesek:

- Egy vagy több interface távolról elérésre, mind java.rmi.Remote leszármazott
- Egy azokat implementáló osztály (ez a „távolról elérhető osztály" a címben)
- Egy stub osztály (rmic generálta) és ha JDK 1.1 kompatibilitás szükséges egy skeleton osztály is

Példa

Távoli interfész (most csak egy interface-ből áll):

```
package com.foo;

public interface Valaszolo extends java.rmi.Remote {
    String kerdez(String kerdes) throws java.rmi.RemoteException;
    static final int CSAK = 666;
}
```

Egy interface a Jáva nyelv szabályai szerint csak nem-statikus publikus absztrakt metódusokat, és statikus konstans mezőket tartalmazhat. A konstans értékének kiolvasásához természetesen nem kell a stub-nak a távoli objektumhoz fordulnia.

Az implementáció osztály:

```
package com.foo;

public class ValaszoloImpl extends java.rmi.server.RemoteObject
                           implements Valaszolo {
    public String kerdez(String kerdes) {
        return "Sajnos nem tudom a választ arra hogy: " + kerdes;
    }
}
```

Itt érdemes megfigyelni, hogy a kerdez metódus itt már nem dob RemoteException-t. (Mert: ha helyben hívjuk, akkor ilyen nem léphet fel. Ha távolról hívjuk, akkor meg az interface-t használjuk, ott meg már dob.)

És aztán jöhet a fordítás (Windows DOS-promptja alól):

```
<C:\user\src\class\> javac com\foo\Valaszolo.java
<C:\user\src\class\> javac com\foo\ValaszoloImpl.java
<C:\user\src\class\> rmic –v1.2 com.foo.ValaszoloImpl
```

Melyik class-nak hol kell majd elérhetőnek lennie:

OSZTÁLY (CLASS FILE)	ELÉRHETŐNEK KELL LENNIE?	
	Szerver VM-ben	Kliensek VM-ekben
Implementáció osztály	Igen	Nem
Távoli interfészt alkotó interface-ek	Igen	Igen*
Generált stub	Igen	Igen*
Generált skeleton	Igen	Nem

*: Nem kell helyben elérhetőnek lennie, ha hálóról le lehet tölteni (lásd: 2.2.4 Osztály letöltés)

2.2.2 Az objektum távolról elérhetővé tétele: exportálás

Az objektum elérhetővé tételére a JDK 1.2-ban szereplő osztályok közül a **java.rmi.server.UnicastRemoteObject** és a java.rmi.activation.Activatable áll rendelkezésünkre. Az UnicastRemoteObject az egyszerűbbik, ez az alap, ezért végig ezt fogom használni. Activatable-ről külön fejezet van. A Sun implementációban ezek az osztályok (alapértelmezésben) TCP alapú hálózati kommunikációt használnak.

Mikor a szokásos módon (new) létrehoztuk az implementáció osztály egy példányát, a(z) 2.1-1. ábra „RMI háttérszálak” felhőjével közölni kell, hogy a példány létezik és készen áll, különben az nem fogja a példánynak továbbítani a hálózatról érkező hívásokat. Ezt exportálásnak nevezzük. Mindezeket, pl. így tehetjük meg:

```
ValaszoloImpl tudakozo = new ValaszoloImpl();
java.rmi.server.UnicastRemoteObject.exportObject(tudakozo);
```

Ha pedig meg akarjuk óvni a példányt a külvilág további zaklatásaitól:

```
java.rmi.server.UnicastRemoteObject.unexportObject(tudakozo,
                                                   false);
```

ami a false paraméterrel csak akkor sikeres, ha egy távoli metódushívás végrehajtása sincs folyamatban az unexportálandó példányban.

Az exportáláskor készül egy stub példány, ami tartalmazza az exportált objektum címét, ami alapján a stub a hálózaton megtalálja azt. Ezt a stub példányt fogjál használni a kliensek az exportált objektum elérésre. Ha elfelejtettünk stub osztályt generáltatni, exportáláskor java.lang.classNotFoundException-t fog keletkezni[14].

Az exportált objektumhoz készült stub példányt a RemoteObject.toStub(exportált_obj) statikus metódussal kérdezhetjük le.

[14] A Sun implementáció az exportáláskor tölti be a stub osztályt. Az osztály nevét az exportálandó példány osztályából következteti ki (pl. ha az exportálandó osztály közvetlenül implementál egy Remote interfészt, akkor ExportáltObjektumOszályánakNeve + "_Stub").

Pontosan miből áll egy távoli objektum címe, amit a stub példány tartalmaz? A Sun implementációban UnicastRemoteObject-el exportált objektumok eseten: host, port, és egy objektumazonosító (java.rmi.server.ObjID). A host:port-on várja (accept) a szerver VM-jében működő RMI rendszer az objektumnak érkező hívásokat. Az objektumazonosító (ami 22 különösebb jelentés nélküli byte) az exportáláskor lesz generálva, és egy VM-en belül egyedülálló, ezért egyértelműen kijelöl ott egy objektumot. Így egy porton több objektum számára is érkezhetnek bejövő hívások.

Mint a(z) 2.1.3 fejezetben erről szó volt, a távoli metódushívások paraméterként átadott vagy visszatérési értékeként visszaadott távoltói elérhető objektumokat azok stub-jára cseréli az RMI-ben használt ObjectOutputStream leszármazott. Ez azonban csak akkor lehetséges, ha az objektumot már exportáltuk, mivel ekkor lesz hozzá létrehozva a stub példány, amire lecserélhető. Ha exportálatlan távolról elérhető objektumot adunk meg egy távoli metódushívás paraméterkén vagy visszatérési értékeként, JDK 1.2.2 előtt kivétel lépett fel, JDK 1.2.2-töl maga az exportálatlan objektum lesz szerializálva elküldve ha szerializálható, egyébként kivétel lép fel.

Az implementáció osztályt a RemoteObject leszármazott **UnicastRemoteObject**-ből is származtathatjuk. Ennek konstruktora rögtön exportálja az objektumot, így nem kell nekünk azt „kézileg" megtenni. Ekkor tehát a ValaszoloImpl így nézne ki:

```
public class ValaszoloImpl
             extends java.rmi.server.UnicastRemoteObject {

    // Az UnicastRemoteObject konstruktora RemoteException-t dob,
    // ha az exportálás sikertelen.
    public ValaszoloImpl() throws java.rmi.RemoteException {}

    ... innentől ugyan az
}
```

Az exportObject hívás nem szükséges, elég new-el létrehozni az objektumot, mint ha az csak egy szokványos helyi objektum lenne. **Ha lehet, implementáció osztályunkat mindig UnicastRemoteObject-ból származtassuk, ne közvetlenül RemoteObject-ből.**

2.2.3 A stub elérhetővé tétele más VM-ek számára: RMI registry

Az exportálás után a szerver oldalon az RMI rendszer várja az objektumnak érkező távoli metódushívásokat, és készült egy stub példány is az exportált objektumhoz, aminek segítségével távolról hívni lehetne annak metódusait. A baj csak az, hogy a stub példány a szerver VM-jében van, el kellene valahogy juttatni a kliensek VM-jébe.

Ez paradox módon RMI-vel fog megtörténni. **Az RMI registry egy közönséges távolról elérhető objektum** (amúgy egy sun.rmi.registry.RegistryImpl példány), **ami** más távoli objektumra mutató **stub-név párosokat raktároz**, és egyfajta telefonkönyv szolgáltatást valósít meg. A „telefonkönyvbe" egy távolról elérhető objektumokat lehet bejegyeztetni, valamilyen tetszőleges néven (String). A bejegyzett stub-hoz a név alapján juthat hozzá a többi VM. (Az RMI registry-vel való kommunikáció természetesen távoli metódusainak hívásával történik.)

De honnan fog a kliens és szerver stub-ot szerezni az RMI registry-hez? Mint későbbiekben le lesz írva, az RMI registry címe (host:port, ObjID) előre ismert, és így le tudjuk generáltatni a stub-ot anélkül, hogy az RMI registry-t futtató VM-el bármilyen kapcsolatunk lenne.

Először, létre kell hozzunk egy RMI registry példányt és exportálni azt. Ezt a JDK-hoz adott rmiregistry program megteszi helyettünk. Tegyük fel, hogy a www.foo.com szervere előtt ülünk és a (hasra ütés) 2030-as porton szeretnénk hogy várakozzon a bejövő távoli metódushívásokra:

```
<C:\> rmiregistry 2030
```

Eddig a host:port ismert, de mi lesz az ObjID? Az RMI registry ObjID-je előre rögzített (egyébként 0), így ez is ismert. Az előre ismert ObjID az RMI registry különlegessége, ez az, ami miatt le tudjuk generáltatni hozzá a stub-ot. (A port-ot az UnicastRemoteObject export metódusának is meg lehet

előre adni, az ObjID-t viszont nem.) Ettől eltekintve az RMI registry olyan, mint egy sima UnicastRemoteObject-el exportált objektum.

Egy RMI registry-re a stub a statikus java.rmi.registry.LocateRegistry.getRegistry(host, port) metódussal (vagy ennek overload-olt változataival) készíttethető el. A stub class (sun.rmi.registry.RegistryImpl_Stub) és a távoli interfész (**java.rmi.registry.Registry**) eleve rendelkezésre áll, mert a JDK tartalmazza őket. A távoli objektum címe is rendelkezésre áll.

Az előző fejezet exportálós példáját folytatva, regisztráljuk a ValaszoloImpl példányt „Tudakozó szolgálat" néven, a távoli metódushívásokra a www.foo.com 2030-as portján várakozó RMI registry-ben:

```
// RMI registry-hez stub szerzése
java.rmi.registry.Registry registry =
java.rmi.registry.LocateRegistry.getRegistry("www.foo.com", 2030);

// Regisztrálás (meghívjuk a távoli registry rebind metódusát)
registry.rebind(tudakozo, "Tudakozó szolgálat");
```

A fenti példa csak akkor működik, ha a www.foo.com-on fut, mert biztonsági okokból a registry csak a saját host-járól érkező távoli metódushívásoknál enged meg regisztrációt. A távoli objektumot az **unbind**(név) metódussal szedethetjük ki az RMI registry-ből, ez is csak az RMI registry hosztjáról hívható.

Ezen túl, mikor egy másik VM-ben szükség lenne a távoli objektumra mutató referenciára, akkor:

```
// RMI registry-hez stub szerzése
java.rmi.registry.Registry registry =
java.rmi.registry.LocateRegistry.getRegistry("www.foo.com", 2030);

// stub szerzése az "tudakozo"-hoz:
Valaszolo x = (Valaszolo) registry.lookup("Tudakozó szolgálat");

// Mostmár hívhatjuk a távoli objektum metódusait
System.out.println(x.kerdez("RMI rulez?"));
```

A lookup természetesen bármelyik host-ról hívható.

A fenti példákkal kapcsolatban észre kell venni, hogy míg a „tudakozo" lokális változó típusa ValaszoloImpl, a hívó oldalon az x típusa Valaszolo. A stub-okra mutató referenciát tároló változó vagy mező típusa, a távoli interfészt alkotó interface-ek egyike szokott lenni.

Az RMI registry alapértelmezett portja 1099. Ha nincs rá különösebb okunk, hogy mást használunk, használjuk ezt (ekkor az rmiregistry programnak nem kell megadni a port-ot). A LocateRegistry overload-olt változatainál, ahol nem kell megadni portot, ott az 1099 lesz, ahol nem kell megadni hosztot ott az a localhost lesz.

A fenti példákban lévő RMI registry műveleteknek van jóval kevesebb gépeléssel járó változata, ha lehet, használjuk inkább ezeket:

```
java.rmi.Naming.rebind("//www.foo.com:2030/Call me!", callme);
```

és

```
x =
(Valaszolo) java.rmi.Naming.lookup("//www.foo.com:2030/Call me!")
```

Mikor egy RMI registry példány megszűnik, elfelejti a regisztrációkat.

Az RMI registry-t nem csak a JDK-hoz adott rmiregistry programmal indíthatjuk, hanem saját java programunkban is létrehozhatjuk, a java.rmi.registry.LocateRegistry.createRegistry(...)-vel. Ez rögtön exportálja is az objektumot, az előre rögzített ObjID-vel.

Ki kell emelni, hogy az RMI registry csak egy lehetséges, de nem az egyetlen módja a stub-ok megszerzésének. Mindazonáltal, **az első távoli objektumra mutató stub-ok megszerzésének ez a tipikus**

módja. A további stub-okat már megkaphatjuk olyan távoli objektumok metódusainak visszatérési értékeként, melyekre az RMI registry-ből szereztünk stub-ot.

2.2.4 Osztály letöltés

Osztály letöltésre akkor van szükség, mikor a deszerializálás folyamán kiderül, hogy a létrehozandó példány osztálya (gyakorlatilag egy class fájl) nem elérhető helyben a VM számára. Az RMI működéskor deszerializálás folyamán létrehozandó objektum a következő lehet:

- A távoli metódushívás egy paramétere (a hívó küldte és a hívott VM-ben kell létrehozni)
- A távoli metódushívás visszatérési értéke (a hívott küldte vissza, a hívó VM-ben kell létrehozni)
- A távoli metódushívás által dobott kivétel (a hívott küldte vissza, a hívó VM-ben kell létrehozni)
- Egy deszerializált objektumban előforduló további objektumok

Szerializálás

Az RMI rendszer által használ java.io.ObjectOutputStream alosztály az annotateClass metódust felülbírálja (override) és **URL-okat ír a stream-be**:

- Ha az osztályt definiáló class loader-e egy java.net.URLClassLoader (vagy annak leszármazottja), de nem a system class loader vagy annak közvetlen vagy közvetett szülője, akkor annak URL-jai (getURLs) kerülnek kiírásra a stream-be
- Egyébként, a **java.rmi.server.codebase** system property-ben szóközzel elválasztva megadott URL-ok íródnak ki a stream-re, vagy null ha a propery nincs beállítva.

További szabály: A „java.”-val kezdődő nevű osztályokhoz soha nem lesz kiírva URL, mert feltételezzük hogy ezek a többi VM-ben is elérhetőek.

Tehát a dolog logikája az, hogy ha az osztály nem volt helyben elérhető, akkor tovább adjuk azt az URL-t, ahonnan mi magunk is letöltöttük azt. Ha viszont helyben elérhető volt, akkor a java.rmi.server.codebase-t adjuk tovább, amiben azoknak az URL-oknak kellene lennie, ahonnan az általunk helyben elérhető osztályok letölthetők.

Az RMI által használt java.io.ObjectOutputStream alosztály felülbírálja a replaceObject metódust, és a java.rmi.Remote-t közvetve vagy közvetlenül implementáló osztályok példányait az azokhoz tartozó stub példányra cseréli (csak ha az objektumot már exportáltuk, mint erről már szó volt). Ezért, a távolról elérhető objektumot implementáló osztályt nem kell letölteni, csak a hozzá készült stub osztályát.

Deszerializálás

Az RMI rendszer által használt java.io.ObjectInputStream alosztály a resolveClass metódust felülbírálja, és URLClassLoader-rel próbálja betölteni az osztályokat. Az URLClassLoader által próbált URL-ok, ebben a sorrendben, a következők lesznek:

- Ha a **java.rmi.server.useCodebaseOnly** system property érétke *nem* "true" és az annotateClass írt ki URL-okat, akkor az annotateClass által kiír URL -ok
- Az ebben a VM-ben (a deszerializáló VM-ben) lévő java.rmi.server.codebase system propery-ben megadott URL-ok

Ne feledkezzünk meg arról, hogy az URLClassLoader (mint a legtöbb custom class loader) először a szüleivel próbálja letöltetni az osztályt. A szülő tipikusan a system class loader, tehát ha az osztály helyben is elérhető, akkor nem lesz letöltve. (Sun JDK 1.2-ben a context class loader (aki nem tudja mi az, ne törődjön vele) lesz beállítva szülőnek, de ha nem állítottuk el, akkor az bizonyára a System class loader.)

Biztonsági kérdések

Az osztály letöltés nem veszélytelen dolog, hiszen nem tudhatjuk, hogy az idegen helyről letöltött osztály metódusai mivel fognak próbálkozni. Ezért, **amíg nincs beállítva SecurityManagger** a VM-ben, az RMI **nem próbál letölteni class-okat**. A java.rmi eleve tartalmaz egy security managert, a java.rmi.RMISecurityManager-t, ez alapban olyan szigorú, hogy nem engedi az RMI-hez szükséges hálózati kommunikációt sem. A szigort, pl. egy policy fájllal lehet enyhíteni, ami pl. így nézhet ki (ezzel csak a saját gépünkön futó alkalmazások kommunikálhatnak, ez csak kísérletezésre jó):

```
grant {
    permission java.net.SocketPermission
               "127.0.0.1", "connect,accept";
};
```

A policy fájl elérési útvonalát a java.security.policy system protperty-ben adhatjuk meg. További részleteket lásd a Java security-vel foglalkozó dokumentumokban.

A system property-ket a java paraméterében is meg lehet adni, pl.:

```
<C:\> java -Djava.security.policy="c:\user\test.policy"
           -Djava.rmi.server.codebase="http://www.foo.com/class/"
           com.foo.MyRMIServer
```

És a JDK-hoz adott programoknak (pl. rmiregistry-nek) is, pl:

```
<C:\> rmiregistry -J-Djava.security.policy="c:\user\test.policy"
           -J-Djava.rmi.server.codebase=http://www.foo.com/class/
```

Amit könnyű elrontani

Ha a class letöltés nem működik, annak néhány lehetséges oka:

- java.rmi.server.codebase végéről lemaradt a '/', vagy '/' helyett '\'-t használtunk, vagy az elejéről lemaradt a protokoll (http://, ftp://, file:/).
- Nem adtunk meg a class-t helyben elérő VM-ben a java.rmi.server.codebase-t, így az nem annotálta az elküldött objektumokat URL-al
- Nincs a kliensen security manager és ezért a class letöltés tiltott

Vigyázat! Ha az RMI registry helyben megtalálja a stub osztályokat, a leírtak értelmében nem a stub küldője által megadott URL-okat fogja továbbadni a stub-bal a lookup hívójának, hanem a saját java.rmi.server.codebase property-ét (már ha egyáltalán az be volt állítva). Ha ez nem egyezik a regisztráló által használt java.rmi.server.codebase-el, akkor a registry által megadott URL esetleg nem lesz jó, és a lookup hívója nem tudja majd letölteni a stub class-t, és java.rmi.RemoteException lép fel a lookup-ban. Egy lehetséges megoldás: a CLASSPATH környezeti változót úgy kell beállítani, hogy az rmiregsitry ne találja meg a stub osztályokat és ezért kénytelen legyen letölteni azokat.

És még egy megfigyelésem a class letöltéssel kapcsolatban (JDK 1.3-al): A már egyszer letöltött osztályok egy ideig megjegyzésre kerülnek, így ha a deszerializáláskor ismét előkerül egy már egyszer letöltött osztály, akkor nem biztos, hogy újra le lesz töltve. Ezért hiába fordítjuk újra a letölthető osztályt, a túloldalon a régi verzió lesz használatban.

2.2.5 Szemétgyűjtés

Java-ban a feleslegessé vált objektumok törlése automatikusan történik. Ehhez tudnia kell a VM-nek, hogy elérhető-e még az objektum valahogy egy aktív szálból, vagy sem. A garbage collector (GC) azonban csak azt nézi, hogy mire van helyi referencia. A távoli referenciákról (mivel ilyen Jáva nyelvben nincs) amit a stub-ok testesítnek meg, még nem hallott, így pl. lazán kidobna egy olyan objektumot, amire nincs helyi, csak távoli referencia. (Szóhasználatomban: stub példány = távoli referencia)

Az exportált objektumokra az RMI háttérszálak őriznek helyi referenciát, így a GC nem dobja ki őket. De ha már nincs távoli és helyi referencia sem az objektumra, az RMI háttérszálakból való elérhetőségnek nem szabadna akadályoznia az objektum kidobhatóságát (gondoljunk átlátszóságra: ilyenkor elvileg már egy referencia sincs az objektumra). Az RMI rendszer ezért nyilván tartja, hogy egy távolról elérhető objektumra van e stub más VM-ekben. **Amíg van rá távoli referencia** (azaz stub), addig az objektumra az azt tartalmazó VM RMI rendszere tárol egy helyi referenciát, így az **nem lesz kidobva** akkor sem, ha már elvileg csak távoli referencia van az objektumra. Ha viszont már nincs rá távoli referencia, akkor az RMI háttérszálak csak gyenge referenciát[15] tárolnak az objektumra, és így nem akadályozzák a GC-t munkájában. (Ne felejtsük el, hogy amennyiben egy rmi registry-ben regisztráltuk az objektumot, akkor ott már van rá egy távoli referencia.) Ezt az elosztott rendszerekre[16] kibővített szemétgyűjtést, distributed garbage collecting-nek (DGC) hívjuk.

Ha a távoli objektum implementációnk implementálja a java.rmi.server.Unreferenced interface-t, akkor mikor már nincs rá több távoli referencia, az objektum unrefrenced metódus meg lesz hívva. Tévedések elkerülése végett, ez a DGC működéséhez egyáltalán nem szükséges, de néha hasznos lehet, jó tudni róla.

Mikor egy VM-be megérkezik egy stub ami X távoli objektumhoz készült, az RMI rendszer visszaszól a távoli objektum VM-jén futó RMI rendszernek, hogy „ebben a VM-ben van referencia az X objektumra”. Mikor az X objektumhoz tartozó utolsó stub példány kidobásra kerül a VM-ben, szól a távoli objektum VM-e felé, hogy „ebben a VM-ben már nincs referencia az X objektumra”. A távoli objektum VM-jében futó RMI rendszer ezek alapján nyilvántartja, hogy a hálózat mely távoli VM-jeiben van referencia az általa szolgáltatott távoli objektumokra.

A háttérben futó RMI rendszer időközönként visszajelzést küld azoknak a VM-eknek, akikben lévő távoli objektumokra referenciát tárol: „ebben a VM-ben (még mindig) van referencia az X objektumra”. (Ezt szemléletesen a referencia bérlésének meghosszabbításának szokták nevezni.) Ha egy bizonyos VM-ben lévő távoli referenciáról ilyen jelzés 10 percen[17] belül nem érkezik, akkor a távoli objektumot szolgáltató VM-en futó DGC úgy veszi, hogy abban a VM-ben már nincs referencia az objektumra. Így, ha egy VM meghal, és ezért soha nem tud szólni, hogy a távoli referenciát már nem használja, a DGC nem fogja öröké tartogatni az objektumot. Persze lehet, hogy a távoli referencia birtoklója más okokból nem tudott válaszolni, és ilyenkor előfordulhat, hogy egy objektum ki lesz dobva, holott még volt rá távoli referencia. A **távoli referenciáknál a referential integrity nincs biztosítva**. Azaz nem biztosított, hogy egy távoli referencia (egy stub) egy még létező távoli objektumra mutat. Ez egyébként nem csak a DGC miatt van így: egy objektumot bármikor unexportálhatunk, és akkor nem lesz elérhető. Ha ilyen stub-ot használunk, RemoteException fog fellépni.

A DGC-től ne várjunk villámgyors reakciókat. Sokáig eltarthat, amíg kiderül, hogy valamire már nincs távoli referencia. Ha valami kavarodás volt (ide értve azt is, mikor a kliens VM teljesen szabályosan leáll, ugyanis ilyenkor sincs az RMI rendszernek módja az ekkor GC által még fel nem szabadított stub-ok megszűnéséről üzenetet küldeni), akkor ez esetleg 10 percecet (ez a „bérleti idő”) is igénybe vehet. A stub-ot használó VM akkor küld üzenetet arról, hogy a stub-ot már nem használja, mikor a stub-ot a helyi GC kidobja[18]. Mivel a GC nem szeret kapkodni, ez néha csak percekkel az után történik

---

[15] Gyenge referencia (weak reference): Olyan referencia, ami nem tartja vissza a GC-t attól, hogy a referált objektumot kidobja, de amíg az nincs kidobva, a gyenge referenciával készíttetni lehet rá erős referenciát (erős referencia: ez az, amit mindenki ismer, a „normális” referencia, ami nem engedi, hogy a referáltat a GC kidobja).

[16] Elosztott rendszer: olyan rendszer, amely több viszonylag független komponensből, tipikusan több számítógépből áll össze működő egésszé. A rendszer alkotói elosztva vannak jelen, nem koncentráltan. Az RMI-vel kialakított rendszer tipikusan ilyen.

[17] Ez a „bérleti idő” alapértelmezése, de meg lehet mást is adni java.rmi.dgc.leaseValue system protperty-vel, milliszekundumban. A „bérlet” megújításával a „bérlők” először a bérleti idő felénél próbálkoznak.

[18] Azoknak, akik tudják, hogy mi az a weak és phantom reference: Pontosabban, gondolom akkor mikor a ReferenceQueue-ban felleli őket az RMI rendszer, de ez gyakorlatilag kb. ugyan az időpont.

meg, hogy a stub kidobható állapotba kerül, azaz már lehet tudni, hogy senki sem fogja többet használni.

Ha a VM-ben már egy általunk indított szál sem fut, akkor megszoktuk, hogy a programunk azonnal kilép, azaz a VM leáll, visszakapjuk a promptot. **Ha** azonban **van exportált objektumunk a VM-ben, akkor a VM nem fog leállni**, mert a háttérben futnak az RMI rendszert kiszolgáló szálak. Ezek a szálak majd csak akkor álnak le, mikor már nincs távoli referencia az exportált objektumokra (a DGC szerint), és az utolsó exportált objektumot is kidobta a helyi GC. Így tehát a kilépés néhány perces késleltetéssel történhet meg, mert meg kell várni amíg kiderül, hogy már nincs távoli referencia az objektumokra, és aztán meg kell várni, míg a helyi GC kidobja őket. Ha unexportObject-el unexportáltunk minden exportált objektumot, akkor természeten ez a késleltetetés nem következik be.

2.2.6 Néhány szó a hálózati kommunikációról

Ha absztrakt szinten vizsgáljuk a kliens és szerver kapcsolatát: **A kliens** hívja a szerver metódusát, tehát ő **kezdeményez**i a kapcsolatot, ő kapcsolódik a szerverre. **A szerver passzív** és valamilyen porton várja a metódushívásokat, és a kliens kezdeményezésre kiépült kapcsolatot használja az eredményt visszaküldésére is. Tehát a szerver elvileg sohasem próbál kapcsolódni a klienshez.

Socket factory-k

Ez az alfejezet feltételezi a java.io.Socket és java.io.SocketServer osztályok ismeretét, és némi jártasságot a hálózati protokollok terén, de át is lehet ugrani.

A RMI rendszer lehetőséget ad arra, hogy a szerver és a kliens közti kommunikáció socket szintű megvalósítását mi írjuk meg. Így pl. megoldható, hogy az adatok, tömörített vagy titkosított formában közlekedjenek a hálón. Ehhez egy saját java.net.Socket és egy saját java.net.ServerSocket leszármazott osztályt kell készítenünk. A kliens a kommunikációhoz a Socket lesz használatban, míg a szerver oldalon a ServerSocket.

Az általunk készített Socket és ServerSocket használatára kétféle módon vehetjük rá az RMI rendszert:

- A java.rmi.server.RMISocketFactory.**setSocketFactory**(**RMISocketFactory** factory) metódussal.
- **UnicastRemoteObject** esetén az exportObject-nak vagy, ha a távoli objektum implementációnk UnicastRemoteObject leszármazott a konstruktorának átadott java.rmi.server.**RMIClientSocketFactory** és **RMIServerSocketFactory** példánnyal.

A socket factroy-ik olyan objektumok, melyek Socket vagy ServerSocket példányokat (illetve ezek leszármazottjait) gyártanak. Ezeket a socket factroy-nak egy metódusa adja vissza: a ServerSocket-eket a **createServerSocket**(int **port**), a Socket-eket a **createSocket**(String **host**, int **port**). Az RMIClientSocketFactory és a RMIServerSocketFactory csak interfészek, az egyik a createSocket-et tartalmazza, a másik a createServerSocket-et. Az RMISocketFactory egy absztrakt osztály, ami mindkettő interfészt implementálja (illetve nem mert azért absztrakt…). (A setSocketFactory meg statikus metódus, ezért hívhatjuk.)

Ha a setSocketFactory-nak megadunk egy RMISocketFactroy leszármazott példányt, akkor ezen túl, exportáláskor az RMISocketFactroy példány createServerSocket(int port) metódusa lesz meghívva, és az ez által legyártott ServerSocket-en fog hallgatózni (accept metódus) a bejövő hívásokra az RMI rendszer. Ha egy porton több exportált objektum hallgatódzik, természetesen csak az első exportálásánál lesz legyártatva ServerSocket példány. Ha egy stub, egy távoli objektum metódusát szeretné meghívni, akkor a setSocketFactory-val megadott RMISocketFactroy példány createSocket(String host, int port) metódusa által visszaadott Socket lesz felhasználva a kommunikációhoz. Mivel a visszaadott Socket már kiépített a kapcsolattal rendelkezik, a stub-nak már csak az a dolga, hogy elküldje, illetve olvassa rajta keresztül a byte-okat.

Az UnicastRemoteObject konstruktor esetén egy RMIClientSocketFactory-t implementáció és egy RMIServerSocketFactory-t implementáló objektumot lehet megadni. Ez látszólag ugyan az, mint a

setSocketFactory esete, csak két részletben adjuk át ugyan azt. A két módszer közt a hatalmas különbség a következő:

- A **setSocketFactory**-val azt határozzuk meg, hogy ezen a VM-en mit használjanak az exportált objektumok a hívások fogadásához szükséges ServerSocket-ek legyártatására, és hogy ezen a VM-en mit használjanak a stub-ok a távoli szerverek eléréséhez szükséges Socket-ek legyártatására. Tehát **minden objektumra és stub-ra vonatkozik, de csak ezen a VM-en belül.**

- Az **UnicodeRemoteObject**-nál azt határozzuk meg, hogy az exportObject-el (vagy a konstruktor által automatikusan) exportált objektum mit használjon a hívások fogadásához szükséges ServerSocket legyártatására, és hogy az ehhez az objektumhoz készült stub-ok mit használjanak az ezzel az objektummal való kapcsolatteremtéshez szükséges Socket-ek legyártatására. Tehát, a **socket factory-k csak erre az egy objektumra vonatkoznak**, de viszont **minden VM-ben**. Az általun megadott **RMIClientSocketFactroy példány a stub-bal átutazik** a távoli VM-ekbe. (Az RMIServerSocketFactory példány nem, ezért kell külön paraméterként átadni őket.) Ezért, ilyenkor a kliensen elérhető kell, hogy legyen a RMIClientSocketFactroy implementációnk osztálya (és az általa visszaadott Socket osztálya), akár helyben, akár a már bemutatott osztályletöltéssel, különben a stub deszerializációja sikertelen lesz.

Az UnicodeRemoteObject beállítása az adott objektumra felülbírálja a setSocketFactroy-val megadott socket factory-kat.

A socket facory-kben rejlik egy értékes lehetőség: Mivel a createSocket már a kiépített kapcsolattal rendelkező Socket-et adja vissza, a metódus kísérletezhet többféle kapcsolat-felvételi módszerrel, és csak azt a Socket-et adja vissza, amelyikkel a kapcsolat kiépítése végül is sikerült. Ezt az alapértelmezett RMIClientSocketFactory (ami akkor van használatban, ha mi a fent felsorolt módszerekkel nem adtunk meg még mást) a **tűzfalak kijátszására** használja ki:

1. A kliens oldali stub először, közvetlenül JRMP (Java Remote Method Protocol) protokollt próbál használni a szerverrel való kommunikációra. Sok tűzfal ezt nem fogja hagyni, mert csak a néhány gyakrabban használt protokollok használatát engedélyezi.
2. Ha az előző nem sikerült, akkor, mivel a HTTP használatát a legtöbb tűzfal engedélyezi, a szervernek egy HTTP POST kérés body-jába próbálja elküldeni az üzenetet. A szerver oldali alapértelmezett factory-val gyártott ServerSocket ezt észleli, és olyan Socket-et ad vissza ami leszűri a HTTP miatti sallangokat. A visszatérési érték HTTP válasz formájában lesz visszaküldve. Sok tűzfal még ezt sem fogja engedni, mert a távoli objektumot szolgáltató szerver nem a HTTP well-known portján (80) várja az üzenetet.
3. Ha az előző sem sikerült, akkor megpróbálja az előzőt, de most a 80-as portra küldi a POST kérést, a szerver oldalán direkt e célra telepített CGI-nek, ami továbbítja a megadott portra azt: http://host/cgi-bin/java-rmi?forward=<port> és a többi adatot a CGI a szokásos módon, a standard bemeneten kapja meg. A CGI felépíti a kapcsolatot a forwad paraméterben megadott port-al, továbbítja a kapott üzenetet, majd megvárja a választ és azt HTTP válaszként visszaküldi.

A felsorolt módszerek, sorrendben egyre rosszabb teljesítményt nyújtanak, de legalább létrejön a kommunikáció.

És így a végén még visszautalnák arra a bekezdésre, ami úgy kezdődött, hogy „Ha absztrakt szinten vizsgáljuk a kliens és szerver kapcsolatát”. Az absztrakt szó azért van ott, mert:

- A Socket és a ServerSocket „interfésze” mögé sokféle implementációt lehet bujtatni, és így lehet, hogy ami kívülről Socket-ként viselkedik, az az alacsonyabb szintű folyamatok tekintetében már nem.

- Az RMI rendszer próbálja a kapcsolat felépítések számát minél alacsonyabbra szorítani, mivel ez elég időigényes művelet. Ezért, a kliens és a szerver közt kiépült kapcsolatot csak bizonyos késleltetéssel bontja le, így ha sűrűn egymás után több távoli metódushívás van ugyan arra a host:port-ra és egymással „equals” RMIClientSocketFactory-val kellene legyártani a Socket-et, akkor újra és

újra felhasználja ugyan azt a Socket-et, nem készít újat. Tehát nem igaz, hogy minden egyes metódus hívás egy Socket legyártatásával és új kapcsolat kiépítéssel jár. Ne felejtsük el a sajátgyártmányú socket factory-k equals metódusára figyelmet fordítani!

- A kliensek néha „csak úgy maguktól" is csatlakoznak a szerverre a distributed garbage collection miatt. A GDC miatti hívásoknál (lásd: 2.2.5 Szemétgyűjtés), az X távoli objektumra vonatkozó olyan üzenetek, mint: „(még) van rá referencia", „már nincs rá referencia" továbbításánál is az X RMIClientSocketFactory-ját használják. *(Az rmiregistry.exe-vel indított registry-nek átadott RMIClientSocketFactory által gyártott Socket-ekkel, az nem tud kapcsolódni a szerverre, mert a security managere ezt nem engedi. Így nem tud visszajelezni a szervernek, hogy tárolja a távoli referenciát, és így a szerver egyel kevesebb távoli referenciáról fog tudni, mint amennyi van. Az rmiregistry nem jelzi ki, hogy problémája van, úgyhogy csak azt fogjuk észlelni, hogy a szerver valamiért állandóan kidobálja az épp kliens nélküli regisztrált objektumot. Megoldás: az rmiregisty -J kapcsolója segítségével adjunk meg olyan policy fájlt (-J-Djava.security.policy=<policy fájl>), ami megengedi a kapcsolódást.)*

De szerencsére nekünk (a socket factroy, a Socket és a ServerSocket írójának) általában teljesen lényegtelen, hogy épp miért történik a kommunikáció, nekünk csak csatlakozni és hallgatódzni kell, és byte folyamokat küldeni és olvasni.

Mi fog történni hálózati szinten?

Jó lenne tudni, hogy most akkor pontosan melyik portokon fog hallgatódzni a szerver? Meg konkrétan, hálózati szinten milyen kommunikáció fog zajlani? Erre általában annyi a válasz, hogy: Implementáció függő. De UnicastRemoteObject-eket, alapértelmezett socket factory-kat és Sun JDK 1.2 és 1.3-at feltételezve, az alábbit tudom mondani:

- Az egyetlen használt szállítás szintű protokoll a TCP. (Az alapértelmezett socket factory-val.)
- Egy UnicastRemoteObject-el exportált objektumnak érkező hívásokat, egy exportáláskor választott 1023 fölötti véletlenszerű porton várja az RMI rendszer, ha nem adunk meg explicit módon portot exportáláskor. Egy JVM-en belül több objektumot is exportálhatunk ugyan arra a portra. A „véletlenszerűen" vállaszott port is általában mindig ugyan az összes exportált objektumhoz (ennek előnyei nyilvánvalóak, gondoljunk csak a kiépült TCP kapcsolatok újrafelhasználhatóságára).
- A szerver soha sem próbál csatlakozni a kliensekre, csak a kliensek csatlakoznak a szerverre. Nem túl valószínű, hogy ez a jövőben megváltozna, de hát ki tudja.
- Tudtommal a kliens a következő okokból próbálhat csatlakozni a szerverre:
  - Távoli metódushívás
  - DGC: bérlet meghosszabbítása, azaz visszajelzés, hogy a VM tárolja a stub-ot az objektumhoz
  - DGC: bérlet felmondása, azaz egy adott objektumra többé nem tárol stub-ot a küldő VM
- Azokban az esetekben, amikor ezt figyeltem, a DGC üzenetekhez (bérlet meghosszabbítás, és felbontás) a kliens a szervernek ugyan arra a portjára csatlakozott, mint amire távoli metódushívásnál csatlakozna. De ha ez még igaz is az említett JDK verziókra, akkor se számítsunk erre, ez bármikor megváltozhat.
- Tegyük fel, hogy egy JVM-ben egyetlen objektumot exportálunk, az X portra. Ekkor, én még nem tapasztaltam olyat, hogy bárki is csatlakozni kívánt volna más portra a szerver JVM-en (RMI ügyben)

2.2.7 Példa

Vegyük a következő szokásos példát (nem a Hello World-öt…): Van egy nagyteljesítményű szerverünk, aminek feladatokat lehet küldeni. Az végrehajtja azokat és visszaküldi a kliensnek az eredményt.

Van egy Task (feladat) nevű interfészünk. Ilyet implementáló osztályba csomagolva küldi a kliens a feladatot a szerverre:

```
package com.foo.example.rmi1;

public interface Task {
    public Object execute();
}
```

És van egy Executer (végrehajtó) nevű távoli interfészünk. Ez lesz a szerver távoli interfésze:

```
package com.foo.example.rmi1;

import java.rmi.*;

public interface Executer extends Remote {
    public Object executeTask(Task task) throws RemoteException;
}
```

És a távoli interfészt implementáló osztály:

```
package com.foo.example.rmi1;

import java.rmi.*;
import java.rmi.server.*;

public class ExecuterImpl extends UnicastRemoteObject
                          implements Executer {

    public ExecuterImpl() throws RemoteException {}

    public Object executeTask(Task task) {
        return task.execute();
    }
}
```

Mint látható, a távoli objektum executeTask metódusának kell elküldeni a végrehajtandó feladatot, ami annak meghívja az execute metódusát, és visszaadja az eredményt. Ez tipikusan olyan alkalmazás, ahol kódletöltés szükséges, mert a szervernek le kell töltenie a különféle Task-ot implementáló osztályokat, melyekkel a kliensek az elvégzendő faladatot küldik.

És egy Task-ot implementáló osztály. Ez szerializálható, így lehet távoli metódushívás paramétere:

```
package com.foo.example.rmi1;

public class TestTask implements Task, java.io.Serializable {

    private double x;

    public TestTask(double x) {
        this.x = x;
    }

    public Object execute() {
        System.out.println("TestTask végrehajtása...");

        // Most ez jelképezi a számításigényes feladatot...
        double result = Math.asin(x) / Math.log(x);

        return new Double(result);
```

```
    }
}
```

Elindítjuk az rmiregistry-t az alapértelmezett porton, majd elindítjuk az alábbi osztályt (miután rmic-el elkészítettük a stub-ot), ami létrehoz és exportál egy ExecuterImpl példányt, és regisztrálja azt a rmiregsitry-ben:

```
package com.foo.example.rmi1;

import java.rmi.*;

public class Server {

    public static void main(String[] args) throws Exception {
        // Hogy lehessen kódletöltés
        System.setSecurityManager(new RMISecurityManager());

        Naming.rebind("Executer", new ExecuterImpl());
        System.out.println("Várom a beérkező hívásokat...");
    }
}
```

Ha elindítjuk a fenti osztályt, akkor az kiírja, hogy „Várom a beérkező hívásokat…” majd vár, soha sem lép ki. Ennek oka, hogy a létrehozott és exportált ExecuterImpl példányra van egy távoli referencia az rmiregistry-ben. (Részleteket lásd a(z) „2.2.5 Szemétgyűjtés” fejezetben.)

És most elindítjuk az alábbi osztályt, ami meghívja az imént exportált objektum executeTask metódusát:

```
package com.foo.example.rmi1;

import java.rmi.*;

public class Client {

    public static void main(String[] args) throws Exception {
        Executer ex = (Executer) Naming.lookup("Executer");
        Object result = ex.executeTask(new TestTask(0.123));
        System.out.println("Az eredmény: " + result);
    }
}
```

Kövessük a Task paraméter útját:

1. Az ExecuterImpl_Stub (ex) executeTask metódusa a szerverre küldi a TestTask példányt, szerializált formában.
2. A szerver VM-ben működő RMI rendszer fogadja a távoli metódushívást. A szerializált TestTask példányt deszerializálja, és az így nyert TestTask példányt paraméterként használva meghívja a helyi ExecuterImpl példány executeTask metódusát.
3. Az executeTask metódus meghívja a kapott TestTask példány execute metódusát, ami szerver képernyőjére kiírja, hogy „TestTask végrehajtása…”, kiszámolja az eredményt, majd visszaadja azt visszatérési értékként.
4. Az executeTask visszatér az eredménnyel az őt hívó RMI rendszerbe, ami azt szerializálva visszaküldi a kliensre
5. A kliens oldalon az eredményre várakozó ExecuterImpl_Stub megkapja azt, deszerializálja, és visszatér vele az executeTask metódusa.

Kinek melyik osztályt kell elérnie

	Client VM-je	Server VM-je	RMI Registry VM-je
Task	H	H	L*
TestTask	H	L	-
Executer	H	H	L*
ExecuterImpl	-	H	-
ExecuterImpl_Stub	L	H	L

Jelmagyarázat:

H Helyben fellelhetőnek kell lennie

L Helyben fellelhetőnek vagy letölthetőnek kell lennie

- Nincs rá szükség

* Ezeknek a letöltése azért szükséges, mert szerepelnek a ExecuterImp_Stub-ban. Illetve, a Task csak metódus paraméterként szerepel, és mivel a registry nem hívja ezt a metódust, szerintem teljesen felesleges letöltenie. Úgy vettem észre, hogy az RMI előre letölti az összes osztályt, ami egy letöltött osztályon belül szerepel, és nem várja meg az első használatukat, szóval ennek is talán ez az oka.

Ha kivétel lépett volna fel

Ha a kivétel a TestTask példány execute metódusában lépne fel, akkor a szerver ezt hasonlóképp visszaküldené a kliensre, mint ahogy a normális eredménnyel is tette. A kliens oldalon a stub ezt deszerializálná és dobná (throw).

Ha az RMI-ben belső működésében lépne fel kivétel (pl. a szerver nem tudja deszerializálni a TestTask-ot, mert nem tudja letölteni annak osztályát), akkor is a kliens oldalon lépne fel kivétel, méghozzá egy java.rmi.RemoteException.

Ha a Task-ok távolról elérhető objektumok lennének

Ehhez csak a Task interface-t és a TestTask osztály kell kicsit megváltoztatni:

```
package com.foo.example.rmi1;

import java.rmi.*;

public interface Task extends Remote {
    public Object execute() throws RemoteException;
}
```

és

```
package com.foo.example.rmi1;

import java.rmi.*;
import java.rmi.server.*;

public class TestTask extends UnicastRemoteObject implements Task {

    ...

    public TestTask(double x) throws RemoteException {
       ...
    }
```

```
    ...
}
```

A konstruktornak ezért kell RemoteExcepion-t dobina, mert az UnicastRemoteObject konstruktora dobhat ilyet.

És mivel az ExecuterImpl.executeTask-ban lévő Task.execute mostmár dobhat RemoteException-t:

```
...

public class ExecuterImpl extends UnicastRemoteObject
                          implements Executer {

    public Object executeTask(Task task) throws RemoteException {
       ...
    }
}
```

Ha ezzel a task verzióval újra elindítjuk a servert és a klienst, akkor hol fog végrehajtódni a Task? Kövessük a Task paraméter útját:

1. A kliens meghívja az ex.executeTask-ot-ot. A paraméterben megadott TestTask példány exportált távolról elérhető objektum, ezért az ExecuterImpl stub-ja (a példában ex) helyette a TestTask példány stub-ját küldi szerializálva a szerverre. (Mikor a TestTask példány még nem volt távolról elérhető, maga a TestTask példány került elküldésre.)
2. A szerver oldalon az RMI rendszer deszerializálja a stub-ot, és meghívja az executeTask-ot, az imént deszerializált stub-ot használva paraméterként.
3. Az executeTask meghívja a kapott Task execute metódusát. Mivel az egy TestTask stub, az a hívást továbbítja a hozzá tartozó TestTask példányhoz, azaz vissza a kliensre (ami így most tulajdonképp szerver szerepet tölt be)
4. A kliens oldali RMI rendszer megkapja a hívást, és meghívja a TestTask példány execute metódusát. Így végül a kliensen fut le a metódus, a kliens képernyőjén jelenik meg a „TestTask végrehatása…” üzenet.
5. Az execute metódus visszatér, az eredményt a kliens RMI rendszere a szerverre küldi. A szerveren lévő TestTask_Stub példány execute metódusa visszatér az eredménnyel, folytatódik az executeTask metódus végrehajtása.
6. Az executeTask metódus visszatér, az őt hívó szerver oldalon működő RMI rendszerbe, ami visszaküldi az eredményt a kliensre. Így az eredmény ment feleslegesen egy oda-visszát.
7. A kliensen lévő ExecuterImpl stub executeTask metódusa visszatér a kapott eredménnyel.

Miután a kliens kiírta az eredményt, a TestTask példányra már nincs referencia, ezért a kliens futása véget ér. Persze ez DGC-ről szóló fejezetben leírtak miatt ez valószínűleg pár perces késleltetéssel következik majd csak be.

Ez utóbbi példával kapcsolatban még azt szeretném kiemelni, hogy:

- A kliensnek kinevezett VM-ben, az általa szolgáltatott objektumot a szerver VM anélkül érte el, hogy az regisztrálva lett volna az rmiregistry-ben. Az rmiregistry csak a távoli objektumra mutató referenciák megszerzésének egy lehetséges eszköze, semmi más.
- Amit kliensnek neveztünk, az tulajdonképp szerverként is funkcionált a későbbiekben. Ebben az esetben a kliens és a szerver szerencsétlen elnevezés.

2.2.8 Összefoglalás

Mely pontokon nem átlátszó az RMI:

- Viselkedésben eltérések:
  - A távolról nem elérhető objektumok érék szerint lesznek átadva
  - A referential integrity nem biztosított
  - Számos java.lang.Object-től örökölt metódus másképp viselkedik (wait, notify)
  - A távoli metódushívás jelentősen lassabb lehet, mint a helyi
- Megkötések:
  - Távolról való elérésre csak interface-ekben előforduló tagok használhatók: public nem-statikus metódus, vagy statikus final mező.
  - A nem Serializable objektumok nem adhatók át paraméterként, és nem lehetnek visszatérési értékek, eltekintve az exportált objektumoktól, ahol a szerializálható stub osztály példánya lesz elküldve
  - Az implementációs osztálynak közvetve vagy közvetlenül java.rmi.server.RemoteObject leszármazottnak illik (de nem kell) lennie.
- RMI miatt plusz teendőkkel kell foglalkozni:
  - A távolról hívható metódusok java.rmi.RemoteException-t dobhatnak, ezekre fel kell készteni a programot
  - Távoli interface-t kell készíteni, Stub class-okat kell generáltatni
  - Esetleg gondoskodni kell a távolról elérhető objektum exportálásáról, rmiregistry-ben való regisztrálásáról.
  - A távoli referenciák megszerzésére gyakran rmiregistry-t kell használni
  - Az átküldött objektumok osztályának elérhetőségéről (letölthető vagy helyben elérhető legyen a másik oldalon) gondoskodni kell
  - Egyes gépeken speciális folyamatokat kell indítani, pl.: rmiregistry, rmid (ez utóbbiról majd később lesz szó)
  - Többet kell gondolnunk a biztonságra

A távolról elérhető objektum elkészítése és üzemeltetése:

1. Elkészítjük a távoli interfészt alkotó interface-okat (Remote leszármazottak)
2. Elkészítjük az implementáció osztályt (általában UnicastRemoteObject leszármazott)
3. rmic-el legeneráltatjuk a stub osztály az implementációs osztályhoz
4. Készítünk egy alkalmazást, ami készít egy implementációs osztály példányt és exportálja azt, majd szükség esetén regisztrálja egy rmiregistryben. Ha lesz ide kódletöltés, akkor beállít egy SecurityManagert-t is.
5. Elindítjuk az rmiregistry-t (ha ezt nem végzi el az előbbi alkalmazás)
6. A fenti alkalmazást elindítjuk a megfelelő system property-kkel (pl. java.rmi.server.codebase).

Egy távolról elérhető objektumot használó alkalmazás elkészítése és üzemeltetése:

1. Elkészítjük az alkalmazást, ami bizonyára rmiregistry-t fog elsősorban használni a stub-ok beszerzésére. Ha lesz ide kódletöltés, akkor SecurityManager is kell.

2. Meggyőződünk róla, hogy a távoli interfészek és egyéb nem letölthető class-ok elérhetők a VM számára
3. Elindítjuk az alkalmazást. Ha a hívott félnél nem ismert osztályú objektumokat adunk ált, esetleg szükséges, a java.rmi.server.codebase beállítása.

2.3 Aktivizálható objektumok

2.3.1 Mi az az aktivizálható objektum?

Egy aktivizálható objektum a kliens szemszögéből teljesen úgy viselkedik, mint egy szokványos (nem aktivizálható) **távolról elérhető objektum**, mint pl. egy UnicastRemoteObject[19]. A különbség a szerver oldalon látszik: Az aktivizálható objektum, ellenben egy nem aktivizálhatóval akkor is elérhető a kliensnek, mikor valójában nem is létezik, nincs belőle példány. Ha ilyenkor egy kliens meghívja egy metódusát, a távolról elérhető objektum a kliens számára láthatatlanul, példányosítva és exportálva lesz (aktivizálódik), és fogadja a metódushívást.

Az „aktivizálható objektum" kifejezés tulajdonképp egy absztrakció, mert nem mindig egy valós, létező objektumra utal, hanem csak egy a kliens szemszögéből látszólag létező távoli objektumra. Az aktivizálható objektum csak akkor valós objektum, mikor úgymond aktív. **Az aktivizálható objektumot akkor mondjuk aktívnak, ha** mögötte a szerver oldalon egy igazi, **exportált objektum** áll, és akkor mondjuk inaktívnak, ha nem áll mögötte exportált objektum.

Az aktivizálható objektumoknak két fő előnye van a nem aktivizálhatókkal szemben:

- A távolról elérhető objektumoknak **csak akkor kell létezniük** (és fogyasztaniuk a rendszer erőforrásokat), **mikor tényleg szükség van rájuk**.
- Az aktivizálható objektumokkal elérhetjük, hogy **a kliens szemszögéből hosszú életű objektumoknak tűnjenek**. Ha pl. kikapcsoljuk a szervert, majd másnap bekapcsoljuk, a kliensek a tegnap beszerzett stub-al továbbra is elérhetik az aktivizálható objektumot. Hogy miért, az majd később kiderül.

2.3.2 Kitérő: A stub-ok közelebbről

Mint írtam, a kliens számára átlátszó, hogy aktivizálható objektum metódusát hívja vagy nem aktivizálhatóét. Az „átverés" természetesen a stub-ban keresendő.

Ha megnéznék közelebbről egy rmic által generált stub osztályt, azt látnánk, hogy az, a távoli metódushíváshoz, egy java.rmi.server.RemoteRef interfészű objektum invoke metódusát használja. Maga a stub osztály az RMI-hez szükséges kommunikáció teendőivel (szerializálás, deszerializálás, hálózati kommunikáció, visszaküldött eredményre várakozás) egyáltalán nem foglakozik, mindent a RemoteRef-re bíz. A RemoteRef-et a stub, konstruktorának paramétereként kapja meg, és egy mezőjében eltárolja.

A RemoteRef-et implementáló objektumot tehát az adja meg, aki a stub-ot konstruktorával példányosítja. A stub példányosítása tipikusan exportáláskor történik meg:

[19] Értsd (és innentől kezdve mindig): Olyan objektum, ami az UnicastRemoteObject export metódusával lett exportálva. Természeten ide tartoznak az UnicastRemoteObject leszármazottak is.

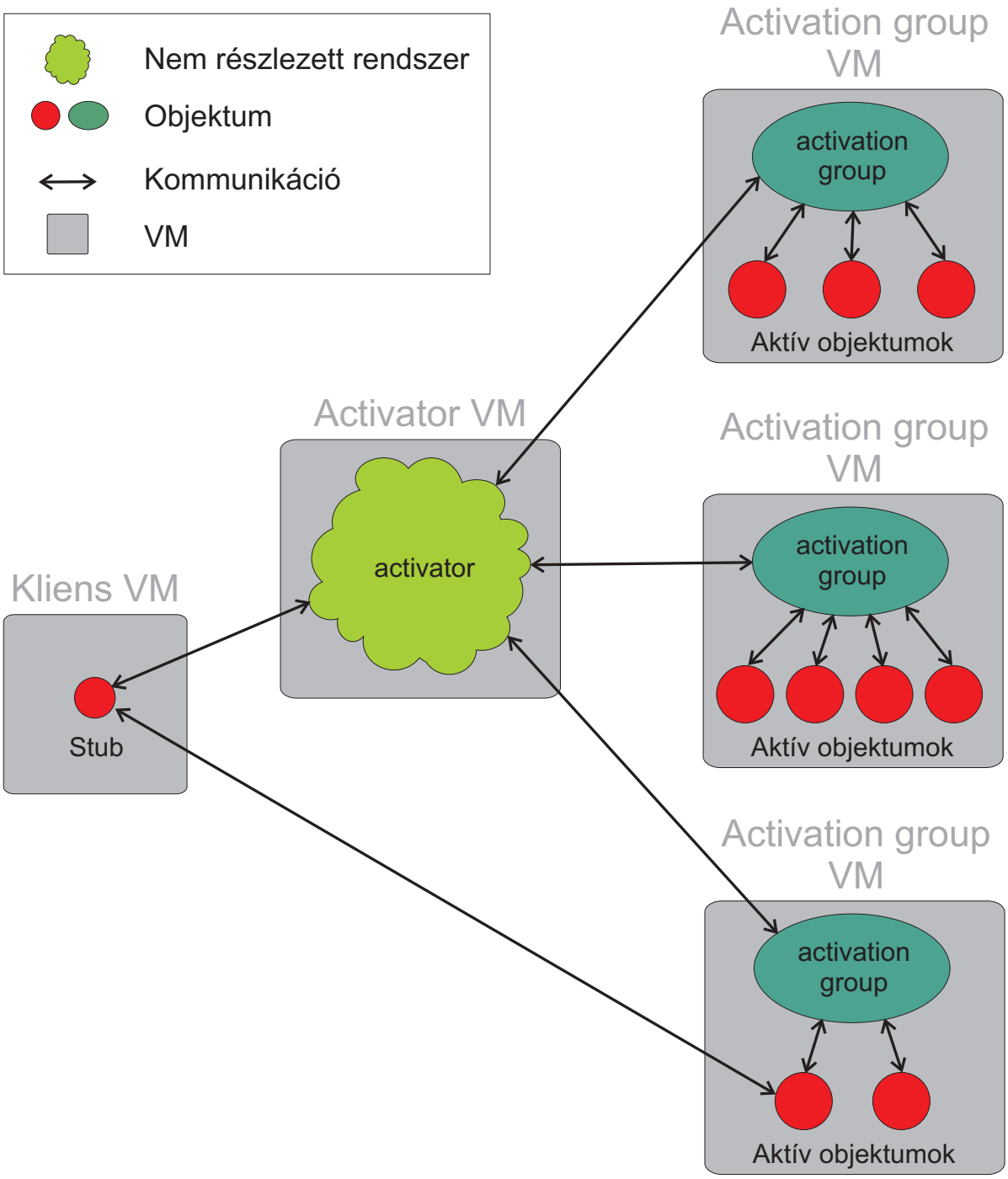
MIVEL EXPORTÁLTUNK	MILYEN OSZTÁLYÚ LESZ A REMOTEREF IMPL.	ADATOK A REMOTEREF PÉLDÁNYBAN
UnicerRemoteObject, socket factroy nincs megadva	sun.rmi.server.UnicastRef	host, port, ObjID
UnicerRemoteObject, socket factroy-val	sun.rmi.server.UnicastRef2	host, port, ObjID, RMIClientSocketFactory
Activatable, socket factory-val vagy a nélkül	sun.rmi.server.ActivatableRef	UnicastRef2 az aktív objektumra, ActivationID

Az java.rmi.activation.Activatable szerepe megegyezik az UnicastRemoteObject-évél, csak ez aktivizálhatóként exportálja az objektumokat. **Az aktivizálható osztályhoz ugyan az a stub osztály használható, mint** amit **UnicastRemoteObject-ekhez** készítettünk**. Csak az exportáláskor készült stub példány által tartalmazott RemoteRef példány fog különbözni**. Ettől fog a két stub másként viselkedni.

2.3.3 Az aktivizációs rendszer építőelemi

Az UnicastRemoteObject esetén, a kommunikációnak két szereplője volt: a kliens oldalon a stub, és szerver oldalon az exportált objektum. De az aktivizálható objektum mögött nem áll exportált objektum mikor inaktív, ezért ilyenkor valaki mással kell kommunikálnia a stub-nak (pontosabban az abban lévő ActivatableRef-nek), hogy aktivizáltassa azt. Ez a valaki az úgynevezett **activator**. Az activator nyilván tartja, hogy a keresett objektum már aktív-e, és ha nem, létrehozatja és exportáltatja azt egy harmadik szereplővel, egy **activaion group**-al.

Egy activator-hoz nulla, egy vagy több activation group tartozik, és minden activation group-hoz nulla, egy vagy több aktivizálható objektum (2.3-1. ábra). **Minden egyes activation group külön VM-ben működik, ebben a VM-ben hozza létre az aktív objektumokat**, mikor a hozzá tartozó **aktivizálható** objektumokat aktivizálni kell**.** Az activator is egy külön VM-ben van. Az activation group-ok és az activator UnicastRemoteObject-ekként vannak implementálva, és távoli metódushívással kommunikálnak egymással és a stub-al. Technikailag az egész aktivizációs rendszer közönséges UnicastRemoteObject-ekből áll, akár mi is megírhatnánk az egészet az eddigi fejezetek alapján.



2.3-1. ábra: Egy aktivizációs rendszer felépítésének sematikus ábrája. Az activator alá több activation group, és minden group alá több aktivizálható objektum tartozhat (az ábrán csak az épp aktívak látszanak, mivel az inaktív aktivizálható objektum mögött nem áll exportált objektum). Az activator egy konkrét implementációja több távolról elérhető objektumból is állhat, ezért van felhőként ábrázolva ellipszis helyett.

Az aktivizálható objektum stub-ja

Mint a(z) 2.3-1. ábra mutatja, a stub két távoli objektummal kommunikál: az activator-ral, és az aktív objektummal. A stub nem mindig tudja az aktív (exportált) objektum „címét”. Ilyenkor, ha távoli metódushívást akarunk végrehajtatni vele, az activator-hoz fordul, hogy megtudja azt. Ha az objektum épp inaktív (tehát nincs is aktív objektum), az activator aktivizáltatja (létrehozatja és exportáltatja). A

megkapott aktuális[20] „címet” a stub eltárolja, és továbbiakban közvetlenül azzal éri el az aktív objektumot.

Az olyan stub-ot, ami tudja a hozzá tartozó aktiv objektum aktuális „címét”, és így közvetlenül tud kommunikálni azzal, **live-stub-nak (élő-stub) nevezzük**. Azokat a stub-okat amik nem tudják a címet, dead-stub-nak vagy halott-stub-nak fogom továbbiakban nevezni.

Hogy a dolog kevésbé legyen misztikus, néhány implementációs részlet: Az aktuális „cím” nem más, mint egy UnicastRef2. Ez az egyik RemoteRef implementáció, amivel az UnicastRemoteObject-eket elérik a kliensek. A halott-stub-ban vagy nincs tárolva ez a RemoteRef (null), vagy tartalma már nem aktuális (ez utóbbi eset azért lehetséges, mert a referential integrity nem garantált).

Egy aktivizálható objektumokra egy java.rmi.activation.ActivationID-vel lehet hivatkozni (attól függetlenül, hogy épp aktív e). Ez egy globálisan egyedülálló és permanens[21] azonosító, ami az activator „címét” (nem az aktív objektumét) és egy az activator-ön belül egyedülálló azonosítót tartalmaz. A stub tartalmazza a hozzá tartozó aktivizálható objektum ActivationID-jét. Ez alapján, szükség esetén fel tudja keresni az activator-t, ami az azonosító alapján tudni fogja, hogy melyik aktivizálható objektum aktuális „címére” kíváncsi a stub.

A ActivationID-ben szereplő „cím” is egy UnicastRef. Ez az egyik RemoteRef implementáció, amivel az UnicastRemoteObject-eket elérik a kliensek. Ez a Sun implementációban egy RMI registry-re mutat, amit az activator indított, alapértelmezésben a 1098-as porton. Ebben a registry-ben regisztrál "java.rmi.activation.ActivationSystem" néven egy távolról elérhető objektumot, a stub ennek az egyik metódusát hívja, mikor az aktuális címre kíváncsi. Ez az áttételesség azért szükséges, mert az RMI registry ObjID-je mindig ugyan az (más objektumoké minden exportáláskor más és más lehet), így az ActivationID-ben szereplő UnicastRef később (pl. egy újraindítás után) is érvényes lesz, meg lehet vele találni az ActivationSystem-et.

Az activator

Az activator felelős a kliensek által használt aktivizálható objektumok aktivizálásáért. Továbbá neki kell megadnia a már aktív objektum közvetlen címét a halott-stub-oknak.

Egy aktivizálható objektumokra ActivationID-jével hivatkozhatunk. Az ActivationID-ket az activator generálja, egy regisztráció során. A regisztrációnál minden olyan információt meg kell adnunk, amire az actovator-nak az aktív objektum előállíttatásához szüksége lesz:

- Az implementáció osztály[22] neve
- Az URL-ok, ahonnan le lehet tölteni az implementáció osztályt
- Egy tetszőleges objektum, ami az implementációs osztály konstruktorának lesz átadva
- Melyik activation group-ban legyen létrehozatva az objektum (ez egy ActivationGroupID, de erről majd később)

Visszatérési értékként kapunk egy ActivationID-t. **Az activator permanensen megjegyzi a regisztrációs adatokat** (pl. lementi egy fájlba), így a kliensek akár a szerver összeomlása és újraindítása után is használhatják a régi stub-okat, mert a bennük tárolt ActivationID továbbra is érvényes lesz.

---

[20] Aktuális cím: Az aktív objektum címe csak ideiglenes. Mikor az aktivizálható objektum inaktivizálódott nem lesz érvényes a cím, mivel nincs is exportált objektum. Másfelől, ha később újra aktivizálódik is, újabb exportálás történik, így az aktív objektum új ObjID-t kap, a régi cím biztosan nem lesz érvényes.

[21] Permanens olyan értelemben, hogy az után is érvényes marad, ha az egész rendszert újraindítottuk.

[22] Az osztály, amelyik implementálja a távoli interfészt. A távolról elérhető objektum osztálya.

Mikor egy halott-stub-nak szüksége van egy aktivizálható objektumhoz tartozó aktív objektum címére, az activator-nak megadja az ActivationID-t. Ez alapján az már tudja, hogy melyik aktivizálható objektumról van szó, és a következőt teszi:

1. Ha, a 4. pontban leírt cache-ben megtalálja a live-stub-ot, akkor visszaküldi azt a kliensnek, és vége.
2. Ha az, az activation group, amelyikben az aktivizálható objektumnak lennie kéne, még nem létezik, akkor létrehoz egy új VM-et, és abban egy új activation group-ot
3. Az activation group-nak elküldi az implementációs osztály regisztrációs adatait, ami alapján az példányosítja, és exportálja azt ha az objektum még nem volt aktív, majd visszaadja az aktív objektum címét is tartalmazó live-stub-ot.
4. Az activation group a későbbi kliens kérések kiszolgálásának gyorsítására és a hálózati kommunikáció minimalizálására cache-eli a live-stub-ot, majd visszaküldi azt a stub-nak. (A halott-stub a megkapott live-stub-ból kiszedi majd a hiányzó címet (UnicastRef2 példányt).)

Az activation group cache-se olyan formában tárolja a live-stub-okat, hogy azok nincsenek hatással a DGC-re.

A Sun JDK 1.2 tartalmaz egy activator implementációt, az rmid-et (RMI Activation System Daemon). Ezt az rmid (rmid.exe) futtatható állománnyal indíthatjuk el. Ez a regisztrációs adatokat a log könyvtárában tárolja.

Az activation group

Az activation group feladata, az alá tartozó aktivizálható objektumok kézbentartása (részletesebben majd lejjebb).

Az activation group-okat az activator hozza létre, szükség esetén. Hasonlóképen **regisztrálni kell** őket, mint az aktivizálható objektumokat kellett. Meg kell adni:

- Az activation group-ot implementáló osztály nevét
- Az URL-okat, ahonnan az osztály letölthető
- Egy tetszőleges objektumot, ami az activation group konstruktorának lesz átadva
- Property-ket, melyek az activation group számára létrehozott VM-ben lesznek beállítva
- Az új VM létrehozásához használt programot, és az annak átadandó argumentumokat

A regisztrációkor visszakapunk egy java.rmi.activation.**ActivationGroupID**-t. Ez nagyon hasonlít egy ActivationID-hez. Ez is az activator címét és egy az activator-on belül egyedi azonosítót tartalmaz, és ez is permanens, azaz az activator újraindítása után is emlékezni fog rá.

Egy aktivizálható objektum regisztrálásánál, egy már előzőleg regisztrált activation group ActivationGroupID-jét kell megadni. Mikor az aktivizálható objektumot aktivizálni kell, **az activator** megnézi, hogy hozott-e már létre annak ActivationGroupID-jével activation group-ot. Ha igen, akkor **az új objektumot a már létező activation group-ba hozatja létre**. Ha nem, akkor először létrehozza a regisztrációkor megadott adatok alapján az új activation group-ot.

Egy activation group feladata, hogy:

1. Az activator kérésre **aktív objektumot hozzon létre**, és visszaadja az aktivizált objektum live-stub-ját.
2. **Közölje az activator-ral ha egy objektum inaktívvá vált**, vagy újra aktivizálta magát anélkül, hogy ezt az activator kérte volna. Így az activator követni tudja az aktív objektumok állapotát, és azzal szinkronban tudja tartani a live-stub-okat tartalmazó cache-ét.

3. Az activator kérésére **visszaadja egy megadott ActivationID-jű objektum aktuális live-stub-ját**, vagy jelezze, hogy ilyen ActivationID-jű aktív objektuma (már) nincs. Ez akkor szükséges, ha az activator-nak valamiért nincs meg cache-elve a live-stub, vagy nem bízik a cache tartalmának aktualitásában.
4. Közölje az activator-rel, hogy ő maga aktívvá vált, vagy hogy inaktívvá vált. A tipikus activation group implementáció létrehozásakor jelzi, hogy aktív, és ha már nincs egy aktív távolról elérhető objektuma sem, akkor közli, hogy inaktív. **Az inaktívvá vált activation group-okat az activator nem használja újra fel**, úgy veszi mintha azok VM-jükkel együtt megszűntek volna (és tényleg ez is szokott történni).

Minden activation group implementáció az absztrakt java.rmi.activation.ActivationGroup leszármazottja, ami meg **UnicastRemoteObject leszármazott**. Az **activator stub-ot tárol az activation group-jaira**, és így a DGC működésében leírtak miatt azok nem lesznek kidobva. Ha egy activation group jelenti, hogy inaktívvá vált, az activator törli a rá mutató stub-ot, így az activation group-ot már kidobhatja a DGC.

A Sun JDK 1.2 tartalmaz egy activation group implementációt, így ha az megfelel céljainknak, nem kell sajátot írnunk.

Az aktivizálható objektum

Ez szinte teljesen olyan, mint egy UnicastRemoteObject, csak ez java.rmi.activation.Activatable leszármazott, vagy annak export metódusával exportált objektum.

Az aktivizálható objektum **aktivizálása automatikusan történik (mikor egy kliens meghívja egy metódusát), de inaktivizálása már nem: az objektum mindaddig aktív marad, míg explicit módon nem kérjük inaktivizálását**. Annak kitalálása, hogy ki és mikor inaktivizálja az objektumot, az alkalmazás fejlesztőjének a dolga.

Amíg egy aktivizálható objektum aktív, az activation group őriz rá helyi referenciát, ezért nem gyűjti be a GC akkor sem, ha nincs már rá távolról referencia (tehát nem használja egy kliens sem). Mikor inaktivizáljuk az objektumot, az activation group unexportálja, és törli magából a rá mutató referenciákat, így az objektum elérhetetlen lesz és begyűjtheti a GC.

Ha az objektum inaktív lett, akkor az aktivizációs rendszer többet nem használja, úgy veszi, hogy már nem létezik. Pl, ha X ActivationID-jü objektum jelezte, hogy már inaktív, és egy kliens miatt ismét aktivizálni kellene X az ActivationID-jü objektumot, akkor az activation group új példányt készít akkor is, ha az inaktív objektum még létezik. (Ez persze csak az alapértelmezett activation group implementációra vonatkozik)

Összefoglalás

Néhány fontos tény, a teljesség igénye nélkül:

- Az activation group-okat az activator, az aktív objektumokat pedig az activation group hozza létre.
- Minden activation group-nak külön VM-je van, és ebben a VM-ben hozza létre az alá tartozó aktív objektumokat
- Mivel (tipikusan) az activtor hozza létre az activation group-ot, az ugyan azon a host-on lesz, mint ő. Ezért az activator az actvation group és az aktív objektumok (tipikusan) egy host-on vannak. (De meg lehet oldani, hogy ez ne így legyen.)
- Az aktivizálható objektum aktivizálása automatikusan történik, de inaktivizálása nem: az objektum mindaddig aktív marad, míg explicit módon nem kérjük inaktivizálását.
- Egy aktivizálható objektumot az egész hálózaton (pl. az Interneten) egyértelműen és permanensen azonosít az ActivationID-je.

- Egy activaion group-ot az egész hálózaton (pl. az Interneten) egyértelműen és permanensen azonosít az ActivationID-je.
- Az activator az aktivizálható objektumok és activation group-ok regisztrációs adatait maradandóan eltárolja. Az ActivationID-k és ActivationGroupID-k kibocsátója az activator.
- A activator, csak a regisztrációkor megadott adatokra támaszkodva, önállóan hozza létre az activation group-ok VM-jét és azokban activation group-okat.
- Az activation group, csak az activator által megadott adatokra támaszkodva, önállóan hozza lére az aktív objektumokat. Az activator a regisztráláskor megadott adatokat adja meg neki.

Az implementációs részletekkel kapcsolatban:

- Az aktivizációs rendszer, közönséges nem aktivizálható távolról elérhető objektumokból (UnicastRemoteObject-ekből) épül fel. Illetve van egy új RemoteRef implementáció is a dologban.
- Egy aktív aktivizálható objektum lényegében semmiben sem különbözik egy exportált UnicastRemoteObject-től. Az aktivizálható objektum mássága, a hozzá készült stub-ban keresendő.
- A stub egy ActivationID-t tartalmaz, aminek alapján megtudakolhatja az aktív objektum aktuális címét. Ha a stub már él (live-stub), akkor az aktív objektum közvetlenül éri el, ugyanúgy, mint egy UnicastRemoteObject-et.

2.3.4 Az aktivizációs rendszer elkészítése

Az alábbi egy aktivizációs rendszer elkészítésének egy lehetséges módja:

1. Megírjuk a távoli interfészt. Ez teljesen ugyan olyan, mint nem aktivizálható objektumoknál.
2. Megírjuk az aktivizálható objektumot implementáló osztályt. Mint nem aktivizálható objektumoknál, eltekintve a következő részletektől:
   - Az implementáció osztály java.rmi.activation.Activatable leszármazott, nem UnicastRemoteObject. (Illetve, ha nem Activatable leszármazott, akkor azzal kell majd exportálni.)
   - Szükség van egy előre kikötött paraméterekkel rendelkező konstruktora, amit az activation group meg fog hívni. Erről majd később.
   - Gondolni kell arra, hogy az objektum hogyan és mikor lesz inaktivizálva. Elképzelésünket nekünk kell implementálnunk.
3. Írunk egy java programot, ami:
   - Regisztrálja az activation group-okat és az aktivizálható objektumokat az activator-nál.
   - A visszakapott ActivationGroupID-ket és ActivatonID-ket valahogyan eltárolja, hogy későbbikében elérhetők legyenek.

Az aktivizációs rendszer üzemeltetésének egy lehetséges módja:

1. Elindítjuk az rmid-et
2. Ha ez eddig még soha de soha nem történt meg, lefuttatjuk a regisztrálásokat elvégző programot.
3. Valamilyen módon elérhetővé tesszünk a kliensek számára halott-stub-okat, melyekkel a regisztrált aktivizálható objektumokat elérhetik Pl. RMI registry-ben regisztráljuk őket.

És kész. A többit majd az rmid intézi.

Az implementáció osztály elkészítése

Ez a fejezet csak az UnicastRemoteObject-eknél leírtakhoz (2.2.1 fejezet) képesti eltéréseket tárgyalja.

Az aktivizálható objektum aktivizálásakor lesz példányosítva és exportálva. Ez úgy történik, hogy az activation group létrehoz belőle egy példányt, egy előre **meghatározott paraméterezésű konstruktor**t hívva. Ha nincs ilyen konstruktora az implementációnknak, nem tudja az aktivizálást elvégezni. A **konstruktorral szemben elvárás, hogy exportálja az objektumot**, méghozzá Activatable.exportObject-el, hogy a stub megfelelő RemoteRef-et tartalmazzon.

Tegyük fel, hogy a távoli interfészünk a Test interface. Ekkor az implementáció osztály valahogy így nézne ki:

```
import java.rmi.*;
import java.rmi.activation.*;

public class TestImpl extends Activatable implements Test {

    ... mezők

    // Ezt a konstruktort fogja hivni az activation group
    public TestImpl(ActivationID aid, MarshalledObject obj)
    throws RemoteException {
        super(aid, 0); // Ez expotálja az objektumot

        ... egyéb inicializálások
    }

    ... egyéb metódusok
}
```

Vagy, ha az implementációs osztály nem Activatable leszármazott:

```
import java.rmi.*;
import java.rmi.activation.*;

public class TestImpl implements Test {

    private ActivationID aid; // A saját ActivationID
    ... egyéb mezők

    // Ezt a konstruktort fogja hívni az activation group
    public TestImpl(ActivationID aid, MarshalledObject obj)
    throws RemoteException {
        Activatable.exportObject(this, id, 0); // Exportálni kell!

        this.aid = aid; // Eltesszük, később még kellhet

        ... egyéb inicializálások
    }

    ... egyéb metódusok
}
```

Az konstruktor ActivationID paramétere az aktivizálható objektum ActivationID-je, a MarshalledObject pedig a regisztrációkor megadott tetszőleges objektumot tartalmazza szerializált formában. Hogy miért van szerializálva arról majd pár fejezettel később.

Továbbá **gondoskodnunk kell az objektum inaktivizálásáról**. Ehhez az activation group inactiveObject metódusát kell meghívnunk. Hogy ezt mikor és hol hívjuk, ránk van bízva.

Ha Activatable leszármazott az objektumunk, akkor annak egyik metódusában így inaktivizáltathatjuk:

```
boolean sikerült = inactive(getID());
```

Ha az objektum nem activatable leszármazott, és az aid nevű mentettük az ActivationID-t:

```
boolean sikerült = Activatable.inactive(aid);
```

Az Activatable.inactive semmi mást nem tesz, mint meghívja a VM-ben működő ActivationGroup példány inactiveObject metódusát a megadott ActivationID paraméterrel. Az alapértelmezett ActivationGroup implementáció inactiveObject metódusa unexportálja az objektumot, majd töröl magából minden rá mutató referenciát, hogy a GC kidobhassa azt. Az unexportálás és így az inaktivizálás, csak akkor sikeres, ha nincs végrehatás alatt álló távoli metódushívás az unexportálandó objektumban (ezért van ott az a „sikerült” változó).

Ha az aktivizálható objektumot inaktivizáltuk, és egy kliensnek ismét szüksége lenne rá, az ActivationGroup új példányt készít a fentebb bemutatott konstruktorral.

Regisztráló program elkészítése

Először **szerezni kell egy stub-ot, amivel kommunikálhatunk az activator-al**. Az activator-nak több távoli interfésze is van, a regisztráláshoz és egyéb ezzel kapcsolatos teendőkhöz az ActivationSystem interface használható.

```
// Stub szerzése az ezen a host-on futó activator (rmid)
// ActivationSystem interfészű távolról elérhető objektumához.
ActivationSystem asys = ActivationGroup.getSystem();
```

Ez már megint valami olyasmi, mint az RMI registry stub megszerzése: egy statikus metódus minden plusz információ nélkül legenerál egy stub-ot. Ez esetben nem állandó ObjID áll a dolog hátterében. Az rmid indít egy RMI registry-t alapértelmezésben a 1098-es porton, és ott beregisztrálja magát java.rmi.activation.ActivationSystem néven. A fenti programsor egyenértékű azzal, hogy:

```
ActivationSystem asys = (ActivationSystem) Naming.lookup(
        "//127.0.0.1:1098/java.rmi.activation.ActivationSystem");
```

Következőnek, a programnak **regisztrálnia kell az activation group-okat**. Minden, az activation group előállításához szükséges információt meg kell adnunk. Az alábbi programrész egy saját ActivationGroup implementációt alkalmazó activation group-ot regisztrál:

```
// ActivationGroup regisztracios adatainak elkeszitese
java.util.Properties prop = new java.util.Properties();
prop.put("java.security.policy", "activationgroup.policy");
prop.put("java.rmi.server.codebase", "http://www.foo.com/class/");
ActivationGroupDesc gdesc1 = new ActivationGroupDesc(
        "com.foo.PatientActivationGroup", // saját A.G. impl. oszt.
        "http://www.foo.com/class/",      // innen lehet letölteni
        new MarshalledObject(new Integer(30)),  // param. kons.-nak
        prop,  // System property-k az activation group VM-jének
        null);  // az alapértelmezett group indítót használjuk

// Regisztráljuk az activator-nál
ActivationGroupID gid1 = asys.registerGroup(gdesc1);
```

A regisztrációs adatokat egy ActivationGroupDesc objektumba csomagolva küldtük el az activator-nak. Ez a következő információkat tartalmazza (sorban, ahogy a konstruktor paraméterei vannak):

- Az ActivationGroup implementáció osztály neve. Ha ilyet nem adunk meg, az alapértelmezett implementáció lesz használva
- Ha nem az alapértelmezett implementációt használjuk: URL-ok ahonnan ActivationGroup implementáció osztály letölthető

- Ha nem az alapértelmezett implementációt használjuk: Egy tetszőleges objektum, amit az ActivationGroup konstruktora paraméterként megkap. Hogy miért van MarshalledObject-ba csomagolva, arról majd később.

- System property-k, amiket az activation group VM-jében be kell állítani. Az alapértelmezett ActivationGroup implementáció RMISecurityManager-t használ, ezért ne felejtsük el megadni a security policy-t.

- A program és argumentumai, ami az activaton group VM-jét és abban az activation group-ot létrehozza. Ha null, akkor az alapértelmezett VM implementáció lesz használva (Sun JDK 1.2, Win32 esetén: a program: java.exe, a paraméter: sun.rmi.server.ActivationGroupInit). A program az előző paraméterben megadott system property-ket „-D" argumentumokként kapja meg. A létrehozandó activation group-ot leíró ActivationGroupDesc-et pedig úgy rémlik a standard bementén kapja szerializálva.

Ha az alapértelmezett ActivationGroup implementációt használnánk, így nézett volna ki az ActivationGroupDesc előállítása (csak az utolsó két paraméter marad meg):

```
ActivationGroupDesc gdesc1 = new ActivationGroupDesc(prop, null);
```

Ez a dokumentum egyébként nem foglalkozik külön a saját ActivationGroup implementációk készítésével, de az itt és a JDK API dokumentációban leírtak alapján, erre elvileg bárki képes kellene hogy legyen.

Következőnek **regisztrálja az aktivizálható objektumokat** (kettőt):

```
// Activatable objektum 1 regisztralasa
ActivationDesc adesc1 = new ActivationDesc(
        gid1,  // activation group, amiben akitválódáskór létrejön
        "foo.com.TestImpl",  // implementáció oszály
        "http://www.foo.com/class/",  // innen lehet letölteni
        new MarshalledObject("BUX"));  // paraméter konstruktornak
Test halottstub1 = (Test) Activatable.register(adesc1);

// Activatable objektum 2 regisztralasa
ActivationDesc adesc2 = new ActivationDesc(
        gid1,
        "foo.com.TestImpl",
        "http://www.foo.com/class/",
        new MarshalledObject("NASDAQ"));
Test halottstub2 = (Test) Activatable.register(adesc2);
```

ActivationDesc szerepe hasonló, mint az ActivationGroupDesc-é, csak ez egy aktivizálható objektum adatait tartalmazza. De mi az az Activatable.register, meg az a halott-stub ott? Hol az ActivationID? Az Actviatable.register(adesc1), a Sun implementációban lényegében azt csinálja, hogy:

```
ActivationID aid1 = asys.registerObject(adesc1);
Test deadstub1 =
        (Test) sun.rmi.server.ActivatableRef.getStub(adesc1, aid1);
```

Tehát regisztrálja az adesc1-et, majd a visszakapott ActivatonID és az ActivationDesc alapján generál egy halott stub példányt, és visszaadja azt. Itt tehát a stub előállítása exportálás nélkül történt. Ez azért lehetséges, mert egy halott stub csak az ActivationID-t tartalmazza, az aktív objektum címét nem. Az adesc1 átadása (gondolom) a stub osztály nevének meghatározásához szükséges („foo.com.TestImpl" + „_Stub").

Mint látható, mindkét aktivizálható objektum ugyan abban az activation group-ban (és így ugyan abban a VM-ben) fog működni, mikor aktív.

A regisztrálást és egyéb a regisztrációs adatokat módosító műveleteket biztonsági okokból csak ugyan arról a hosztról lehet elvégezni, mint ahol az activator (az rmid) fut.

Mostmár megvannak a **halott stub-ok** (azokban vannak az ActivationID-k). Ezeket valahogy **el kéne tárolni**, és a **kliensek számára hozzáférhetővé tenni**. Hogy hogyan, arról semmit nem találtam a Sun dokumentációkban, de nem nehéz ilyet kitalálni. Pl. a két stub-ot szerializálva kimentjük egy fájlba, és írunk egy másik programot, ami a beolvassa ezeket és regisztrálja egy RMI registry-ben. Az rmid-ben regisztráló programot csak egyetlen egyszer kell lefuttatni. Ha később újraindítjuk a rendszert, el kell indítani az rmid-et, és le kell futtatni az előbb említett programot, ami bepakolja a RMI registry-be a regisztráló program által kimentett stub-okat. Ne futtassuk újból az rmid-be regisztráló programot! Ha újra futtatnánk, akkor már összesen négy teljesen különálló aktivizálható objektumunk lenne, melyek két teljesen különálló activation group-ba tartoznának. Az új regisztrációk hozzáadódnának az eddigi regisztrációs adatokhoz.

És mi van, ha meg akarjuk változtatni a regisztrált objektumok adatait, vagy törölni akarunk egy regisztrációt? Két megoldásról tudok:

- A csúnya megoldás: Az rmid-el elfelejtettjük az eddigi regisztrációs adatokat (kitöröljük a log könyvtárát), majd újra lefuttatjuk a módosított regisztráló programot. Ezzel az a baj, hogy ezzel nem a régi aktivizálható objektumokat változtattuk meg, hanem azokat megszüntettük, és újakat hoztunk létre. Így egy régi stub példányt használó kliens szomorúan fogja tapasztalni, hogy az általa használt aktivizálható objektum nincs többé. Ráadásul ezzel esetleg olyan regisztrációs adatokat is töröltünk, amit nem a mi regisztráló programunk állított elő.

- A korrekt megoldás: Az activatior setActivationDesc, setActivationGroupDesc, unregisterObject és unregisterGroup távoli metódusait használjuk. Ezeknek kell paraméterként az ActivationID ill. ActivationGroupID, ezért azokat is el kell mentenünk regisztráláskor. Mármost ez az aktivizálható objektumoknál egy kicsit problémás, mert az API-t elrontották: Activatable.register-el csak stub-ot kapunk, ActivationSystem.registerObject-el meg csak ActivationID-t. Viszont semmilyen hivatalos API-val nem lehet a stub-ból kiolvasni az ActivationID-t (pedig benne van), vagy az ActivationID-ből halott stub-ot készíttetni (ezt tette a fentebb bemutatott sun.rmi.server.ActivatableRef.getStub). Márpedig a halott-stub-ra (ezt juttatjuk el valahogy a kliensekhez) és az ActivationID-re (ez kell a regisztrációs adatok manipulálásához) is szükségünk lenne. Lehetséges, de csúnya megoldások:

  - Használjuk a fentebb bemutatott nem publikus API-t, ha biztosak vagyunk benne, hogy csak Sun JDK 1.2-n és 1.3-an fog futni a regisztráló programunk (nesze neked platform függetlenség), és reménykedjünk, hogy Sun-ék a következő API verzióba kijavítják a hiányosságot.

  - Az aktivizálható objektumhoz írunk egy távolról elérhető metódust (mondjuk, legyen a neve getActivatonID), ami visszaadja az aktív objektum saját ActivationID-jét (ez nem probléma). Activatable.register-el regisztrálunk, megkapjuk a halott-stub-ot, ezt elmentjük, az ActivationID viszont most nem érdekel minket. Ha majd egyszer szükségünk lesz az ActivationID-re, a stub-al meghívjuk a getActivatonID-t. Ezzel feleslegesen aktivizáltuk az objektumot, és van egy elvileg felesleges metódusunk a távoli interfészben, de viszont megvan az ActivationID.

A rendszer üzemeltetése

Tegyük fel, hogy a rendszerünk üzemeltetéséhez két programot írtunk:

- Egy regisztráló programot, ami regisztrálja az aktivizálható objektumokat és activation group-okat, és a kapott halott-stub-okat, és ActivationGroupID-ket egy fájlba menti.

- Egy kliensek számára stub hozzáférést biztosító programot, mely a regisztráló által írt fájlban lévő stub-okat beolvassa, és megfelelő néven regisztrálja a helyi RMI registry-ben.

Ekkor a rendszer indításának lépései, az operációs rendszer minden elindulása után:

1. Elindítjuk az rmid-et. Ez a háttérben fog várakozni a kliensek kéréseire.
2. Ha ez még eddig soha nem történt meg, most lefuttatjuk a regisztrációs programot.

3. Elindítjuk az RMI registry-t
4. Lefuttatjuk a programot, ami bepakolja stub-okat az RMI registry-be.

2.3.5 Az aktivizációs rendszer működésének részletei

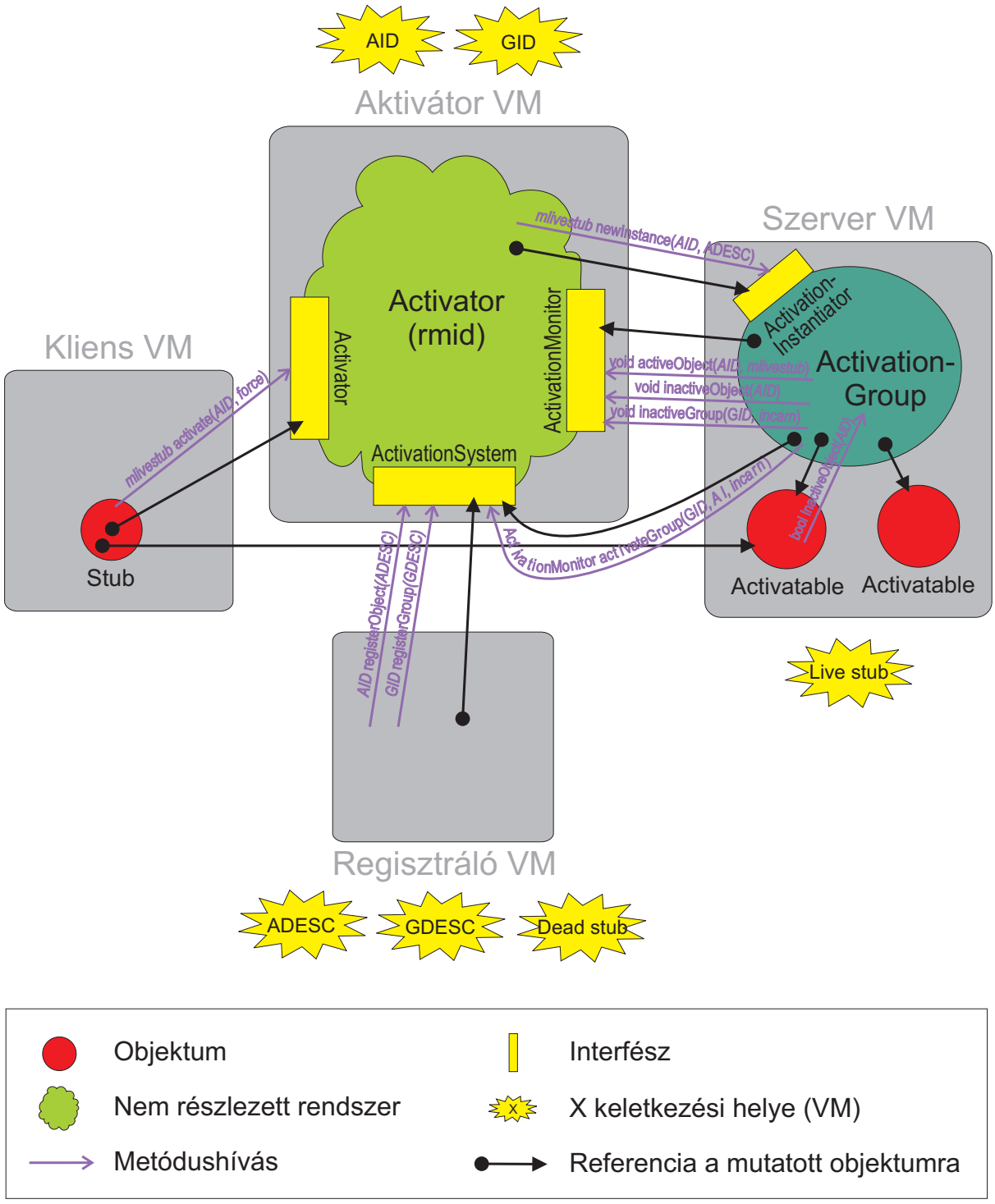
Hogy pontosan hogyan működik a rendszer, az a(z) 2.3-2. ábra és az API dokumentáció segítségével kibogarászható. Csak néhány dolgot szeretnék megjegyezni, ami az API dokumentációból nehezen hámozható ki.

A MarhsalledObject

Egy java.rmi.MarshalledObject példány, a konstruktorának megadott objektumot tartalmazza szerializált (byte folyam) formában. Ha ki akarjuk azt szedni belőle, ezt a get metódusával tehetjük meg, ekkor a szerializálva tárolt objektumot deszerializálja, és visszaadja. A MarshalledObject maga is szerializálható.

A MarshalledObject a konstruktorában megadott objektum szerializálására és deszerializálására ugyan azt a mechanizmust alkalmazza, mint az RMI a paraméterek szerializálásánál és deszerializálásánál: Ugyan úgy annotálja az osztályokat az URL-okkal, letölti szükség esetén az osztályt, és szerializáláskor az exportált objektumokat a stub-jukra cseréli.

Az ActivationGroup stub-jától eltekintve, az activator mindent olyan paramétert MarshalledObject-be csomagolva kér, aminek pontos osztályát tipikusan nem lehet előre tudni: az aktív objektumok stub-ját, regisztrációkor az ActivationDesc-ben illetve ActivationGroupDesc-ben tárolt, konstruktornak átadandó objektumot. Ezeket az activator nem csomagolja ki, így nem kell, hogy a tárolt példány osztályát elérje. Továbbá, így a DGC-t sem fogja befolyásolni, hogy az activator live-stub-okat tárol a cache-jében az aktív objektumokra.



2.3-2. ábra: Az aktivizációs rendszer egy lehetséges fölépítése és a fontosabb metódushívások. A regisztráló VM csak egyszer fut le, nem állandó tagja a rendszernek. Rövidítések: AID = ActivationID; GID = ActivationGroupID; ADESC = ActivationDesc; GDESC = ActivationGroupDesc; mlivestub = live-stub MarshalledObject-be csomagolva; incarn = újraszületés sorszáma.

Néhány megjegyzés az API-val kapcsolatban

- Az ActivationGroup implementáció példányt a statikus ActivationGroup.createGroup hozza létre az újonnan létrehozott VM-ben. A createGroup a következőket teszi:
  - Ellenőrzi az ActivationGroup class egy statikus változójával, hogy van-e már ebben a VM-ben ActivationGroup. Ha már van, kivétel lép fel.
  - Betölti egy RMIClassLoader-rel a paraméterként megkapott ActivationGroupDesc-ben szereplő activation group implementációs osztályt
  - Reflection-nel meghívja activation group osztály konstruktortát, paraméterként az ActivationGroupDesc-böl kiszedett ActivationGroupID-t és MarshalledObject-et használva.
  - Inicializálja a statikus változókat (az egyikbe az új ActivationGroup példányt írja, innen tudta az elején, hogy van e már ActivationGroup ebben a VM-ben)
  - Meghívja az activator (ActivationSystem interfész) activeGroup metódusát, amivel jelzi, hogy az új activation group aktív, elküldi annak stub-ját, és visszakapja az ActivationMonitor-ra a stub-ot. A kapott ActivationMonitor-t elraktározza.
- Az ActivationInstantiator.newInstance meghívódhat egy már aktív objektumra is, ekkor nem szabad az activation group-nak új példányt készítenie, hanem a már meglévő aktív példány stub-ját kell visszaadna.
- Az activator ActivationInstantiator.newInstance hívására elkészült objektumok esetén nem kell az ActivationMonitor.activeObject-et hívni.
- Az Activatable-nek vannak olyan konstruktorai, melyek regisztrálnak is, és rögtön aktivizálják az objektumot. Ebben a dokumentumban nem mutattam olyan gyakorlati alkalmazást, ahol ezek használhatók lennének. Csak ezek a konstruktorok hívják az ActivationMonitor.activeObject-et a többi nem.

Néhány művelet végigkövetése

Egy halott stub-nak meg kell hívnia egy távoli metódust. Ekkor a következők fognak történni:

1. A stub meghívja a benne lévő sun.rmi.server.ActivatableRef invoke metódusát.
2. Az ActivatableRef látja, hogy nincs sun.rmi.server.UnicastRef2-je az aktív objektumra, ezért meghívja a benne (mármint az ActivatableRef-ben) tárolt ActivationID.activate metódusát, az pedig meghívja az ActivationID-ben tárolt RemoteRef invoke metódusával az activator activate metódusát
3. Meghívódik az activator activate metódusa, aminek egyik paramétere az ActivationID. Ha a metódushívás második paramétere (force) true volt, akkor most ugorjunk a 4. pontra. Megnézi, hogy a paraméterként kapott ActivationID-hez van-e a cache-ében live-stub. Ha van, akkor visszatér azzal és ugorjunk a 9. pontra.
4. Az activator megkeresi az ActivationID-hez tartozó ActivationDesc-et, és megnézi hogy aktív-e az abban megadott activation group. Tegyük fel, hogy az.
5. Mivel az aktív ActivationGroup példányhoz van stub-ja, meghívja annak newInstance metódusát, paraméterként megadja az ActivationID-t és az ActivationDesc-et.
6. Meghívódik az ActivationGroup példány newInstance metódusa. Az megnézi, hogy van-e a paraméterként kapott ActivationID-jű aktív objektuma (Ezt valamilyen adatszerkezetben nyilvántartja az ActivationGroup implementáció. Pl. egy HashMap-ban tárolja az aktív objektumokat, és az ActivationID-t használja kulcsnak). Ha van, akkor visszaadja annak live-stub-ját MarshalledObject-be csomagolva és ugorjunk a 8. pontra.

7. Az activation group a paraméterként kapott ActivationDesc-et felhasználva elkészíti a példányt, annak ActivationID, MarshalledObject paraméterezésű konstruktorával. Valamilyen adatstruktúrába letárolja, az új aktív objektumot és ActivationID-jét (pl. az előbb említett HashMap-ban), majd visszatér a MarshalledObject-be csomagolt live-stub-bal.
8. Az activator megkapja MarshalledObject-be csomagolva a live-stub-ot, és beteszi azt a cache-ébe, majd visszatér a MarshalledObject-be csomagolt live-stub-al.
9. Az ActivatableRef megkapja MarshalledObject-be csomagolva a live-stub-ot, kiszedi belőle az aktív objektum UnicastRef2-jét, és bepakolja azt saját magába.
10. Az újonnan szerzett UnicastRef2 invoke metódusát használva meghívja az aktív objektum metódusát, és innentől kezdve minden ugyan úgy megy, mint UnicastRemoteObject-eknél.

Egy aktív objektum elunja magát, és úgy dönt, hogy inaktivizálódik:

1. Meghívja az Activatable.inactive-t, paraméterként megadja a saját ActivationID-jét (ezt onnan tudja, hogy a konstruktora paramétereként megadta neki az activation group)
2. Az meghívja a VM-ben működő ActivationGroup példány inactiveObject metódusát, paraméterként megadja az ActivationID-t
3. Az ActivationGroup példány inactiveObject metódusa megkeresi az ActivationID alapján a saját erre a célra szolgáló adatszerkezetében (előző példa HashMap-ja), hogy melyik objektumról van szó, és megpróbálja azt unexportálni. Ha ez nem sikerül, akkor false-al visszatér és ugorjunk a 7. pontra. Ha sikerült, kidobja a rá mutató referenciát az adatszerkezetből.
4. Az ActivationGroup meghívja az activator ActivationMonitor interfészét használva az inactiveObject metódust, paraméternek az ActivationID-t adja meg.
5. Az activator inactiveObject metódusa meghívódik, és kidobja a paraméterben megadott ActivationID-jű objektum live-stub-ját a cache-éből és visszatér.
6. Az ActivationGroup példány inactiveObject metódusa visszatér true-val.
7. Az Activatable.inactive visszatér azzal, amit az inactiveObject visszaadott (true vagy false)
8. Az objektum megnézi a visszatérési értéket, és látja hogy inaktivizálva lett vagy sem.

A DOKUMENTUMRÓL

Verzió:

Utolsó módosítás: 2001.07.14. 20:04

A legfrissebb verzió a következő címen található: http://www.webhely.hu/doksiraktar/

Szerző:

Dékány Dániel

Ha hibát találsz, vagy egyéb észrevételed van, ne fogd vissza magad: ddekany@freemail.hu.

Felhasznált irodalom:

Sun Microsystems: Java Platform 1.2 API Specification; The Java Virtual Machine Specification 2nd Edition; Java Remote Method Invocation Specification; RMI Tutorials; Sun RMI levelezőlista