Osztott rendszerek

Erőforrások elérése JNDI alkalmazásával, CORBA, RMI

Simon Károly
simon.karoly@codespring.ro

JNDI alapok

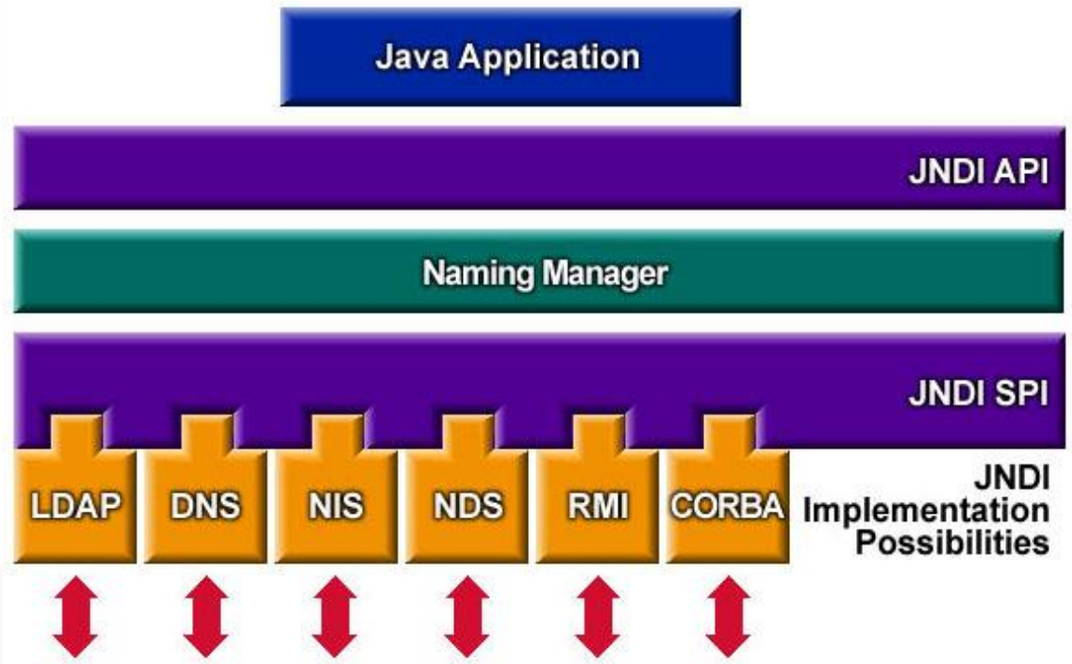
JAVA NAMING AND DIRECTORY INTERFACE

Naming Service

- Naming service: nevek hozzárendelése objektumokhoz, elérési lehetőség (objektumok/ szolgáltatások lokalizálása), információk → központosított adatelérés. Analógia: könyvtár, könyvek katalógusa (könyv-kártyák).
- Directory Service: a naming service kiterjesztése → attribútumok hozzárendelése az objektumokhoz, keresés attribútumok alapján stb.
- Bindings: nevek hozzárendelése objektumokhoz (a naming-service specifikus konvenciók betartásával).
- Közismert naming/directory szolgáltatások:
  - COS (Common Object Services) Naming: CORBA
  - DNS (Domain Name System): Internet
  - LDAP (Lightweight Directory Access Protocol): könyvtár szolgáltatások (directory services) elérése, adatok manipulációja (a DAP alternatívája) (alkalmazás példa: hálózati erőforrások elérése egyetlen azonosító/jelszó segítségével)
- Hasonlóságok: cél, bindings.
- Különbségek: alkalmazási terület, (név) konvenciók.
- JNDI: egységesített hozzáférési felületet (interfészt) biztosít a különböző naming szolgáltatásokhoz.

JNDI

- JNDI API, JNDI Service Provider Interface
- Csomagok:
  - javax.naming: Context, Name, Binding, Reference, InitialContext, NamingException stb.
  - javax.directory: DirContext, Attribute stb.
  - javax.naming.event: EventContext, NamingEvent, NamingListener stb.
  - javax.naming.ldap: LDAP specifikus interfészek és osztályok.
  - javax.naming.spi: naming/directory szolgáltatások fejlesztői számára biztosít lehetőséget, hogy saját implementációt biztosíthassanak szolgáltatásukhoz, amely így JNDI-n keresztül elérhetővé tehető.



JNDI

- Name osztály, származtatott osztályok, helper osztályok.
- Context interfész: bindings halmaz reprezentációja.
- Metódusok:
  - `void bind (String stringName, Object object)`
  - `void rebind (String stringName, Object object)`
  - `void unbind (String stringName)`
  - `Object lookup (String stringName)`
  - `void rename (String stringOldName, String stringNewName)`
  - `NamingEnumeration listBindings (String stringName)`
  - `NamingEnumeration list (String stringName)`
- Minden metódusnak van olyan változata is, amely Name objektumokat kaphat paraméterként.
- A nevek csak egyetlen objektumhoz köthetőek hozzá.
- InitialContext osztály: a kiindulási pont: a naming service meghatározása, paraméterek megadása (felhasználónév, jelszó stb.).

JNDI - példa

- Példa: kapcsolódás egy naming szolgáltatáshoz, adott binding, vagy az összes binding listázása. Alkalmazott szolgáltatás: filesystem service provider (a Sun implementációja, a fájlrendszert naming service-ként kezeli). A program létrehozza a kapcsolatot (InitialContext), felhasználva a parancssor első argumentumát, amely egy lokális könyvtár neve. Kilistázza a parancssor további argumentumaiként megadott nevekhez tartozó objektumokat, vagy ha nem kap paramétert, akkor minden név-objektum párt.

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.Binding;
import javax.naming.NamingEnumeration;
import javax.naming.NamingException;
import java.util.Hashtable;

public class Main {
       public static void main (String[] args) {
                    try {
                                  // Create the initial context. The environment
                                  // information specifies the JNDI provider to use
                                  // and the initial URL to use (in our case, a
                                  // directory in URL form -- file:///...).
```

JNDI - példa

```
Hashtable environment = new Hashtable ();
environment.put (Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
environment.put (Context.PROVIDER_URL, args[0]);
Context context = new InitialContext (environment);
// If you provide no other command line arguments,
// list all of the names in the specified context and
// the objects they are bound to.
if (args.length == 1) {
            NamingEnumeration namingEnumeration = context.listBindings ("");
            while (namingEnumeration.hasMore ()) {
                        Binding binding = (Binding) namingEnumeration.next ();
                        System.out.println (binding.getName () + " " +  binding.getObject ());
            }
      }
      // Otherwise, list the names and bindings for the specified arguments.
      else {
            for (int i = 1; i < args.length; i++) {
                        Object object = context.lookup (args[i]);
                        System.out.println(args[i] + " " + object);
            }
      }
      context.close();
   } catch (NamingException ex) {
      ex.printStackTrace();
   }
  }
}
```

CORBA

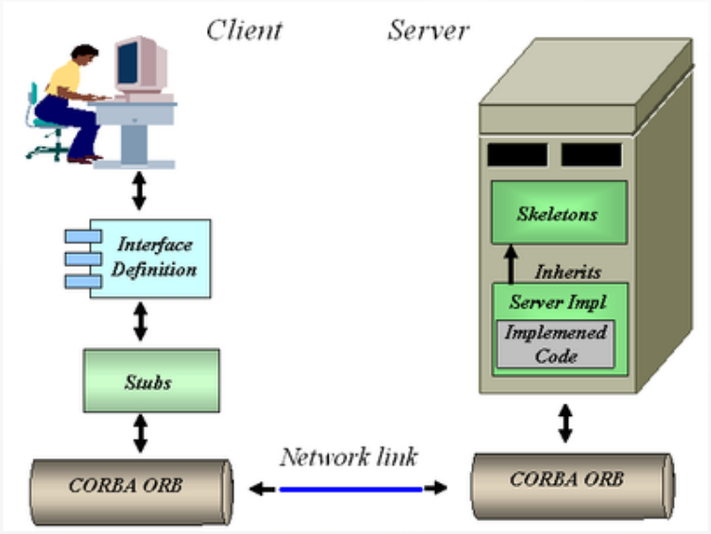
OMG, ORB, CORBA

OMG

- CORBA (**Common Object Request Broker Architecture**) az OMG (**Object Management Group**) által definiált standard, ami lehetővé teszi különböző nyelveken írt, különböző számítógépeken futó szoftverkomponensek együttműködését.
- OMG: 1989-ben alapított nemzetközi, nyílt tagságú non-profit szervezet, melynek eredeti célja standardok kidolgozása osztott objektumorientált rendszerek fejlesztéséhez. Az OMG jelenleg a modellezésre (programok, rendszerek és üzleti folyamatok) és a modell alapú standardok kidolgozására koncentrál.
- A konzorciumot 11 számítástechnikai vállalat alapította, több nagyvállalat részvételével (Hewlett-Packard, IBM, Sun Microsystems, Apple Computer, American Airlines, Data General). Mára több mint 800 tagja van és csaknem mindegyik nagyobb számítástechnikai vállalat képviselteti magát benne.
- Az OMG komponens alapú rendszerekre vonatkozó szemléletmódjának középpontjában az **Object Management Architecture** (OMA) általános referenciamodell áll. Egy OMA-elvek alapján megírt alkalmazás együttműködő osztályok és objektumok együttese, amelyek egymást az Object Request Broker (ORB) segítségével érhetik el.
- Az ORB lehetővé teszi, hogy az objektumok kéréseket (üzeneteket) küldhessenek, fogadhassanak és válaszolhassanak azokra.

CORBA

- A CORBA a kódot speciális "kötegekbe" (bundle) "csomagolja", melyek információt tartalmaznak a kód tulajdonságairól és a felhasználás/meghívás lehetőségeiről. A becsomagolt objektumok ezután a hálózaton keresztül elérhetőek lesznek más programok számára.
- A CORBA az **Interface Definition Language**-et (IDL) használja az objektumoknak megfelelő interfészek deklaráláshoz és specifikálja az interfész "leképezését" egy konkrét programozási nyelvben történő implementációba (Java, C, C++, Smalltalk, COBOL, ADA, Lisp, Python stb.) (nem standard leképezések: Perl, Visual Basic stb.)



A CORBA "szíve", az ORB felelős az objektumok közötti kapcsolat létrehozásáért és fenntartásáért. Az ORB felett az objektumok úgy létesítenek egymással kapcsolatot, mintha mindannyian egyetlen programban, egyetlen címtartományban léteznének. Az ORB feladata a köztes, gyakran igen összetett kommunikáció megvalósítása, többek között a távoli objektum hálózati helyének megtalálása, az objektumot implementáló folyamat elindítása, a hatékony adatcsere stb. Az ORB több összetevőt tartalmaz a kliens és a szerveroldalon…

Kliensoldali részek

Client IDL Stubs

- A kliens IDL kapcsolódási csonk biztosítja a kliensoldal statikus felületét a CORBA objektumok eléréséhez.

Dinamikus run-time interfész (Dynamic Invocation Interface - DII)

- A kliensoldalon a DII teszi lehetővé, hogy a CORBA objektumok a futási idő megkezdéséig ne tudjanak azokról az objektumokról, amelyeket majd használni fognak. Ezekkel az objektumokkal az ORB teremti meg a kapcsolatot. Az interfész szerver oldali részének az elnevezése Dynamic Skeleton Interface (DSI).
- A DII szemantikája megfelel az IDL leírásoknak. Egy kliens objektum generál egy futási idejű kérést, azt elküldi a kiválasztott objektumnak, átadva a szükséges paramétereket. A paramétereket egy irányban láncolt listaként kezeli a rendszer, és futás idejű típusellenőrzést hajt végre rajtuk.

Interface Repository API

- Az interfész-szótár programozói felület futási idejű hozzáférést enged az előbb említett adatokhoz, az interfész-szótárhoz.

ORB interface

- Az **ORB felület** néhány hasznos helyi szolgáltatást nyújt (pl. objektumhivatkozások karakterlánccá és visszakonvertálása, amely az objektumok kapcsolatainak tárolásához előnyös stb.)

Szerveroldali részek

Server IDL Stub

- A szerver IDL kapcsolódási felület (Server IDL Stub, váz, skeleton) a szerverobjektumok által nyújtott statikus szolgáltatásokat definiálja. A kliensoldali csonkhoz hasonlóan a vázat (szkeletont) is az IDL fordító generálja.

Dynamic Skeleton Interface (DSI)

- Futási időben képes csatlakozási információt szolgáltatni azokról az elérhető szerveroldali objektumokról, amelyek nem rendelkeznek IDL definiált statikus csonkkal.
- Az ORB a DSI-t használja föl a kérés eljuttatására ahhoz az objektumhoz, amely fordítási időben még semmit sem tud saját implementációjáról. A DSI ebben az értelemben megfelel egy típus-specifikus IDL csatlakozónak.

Object Adapter

- Az objektum-adapter a fenti két felület implementációjához szükséges, de önállóan is használható. Itt helyezkedik el az objektumok létrehozásához, metódusaik hívásához szükséges futási környezet.

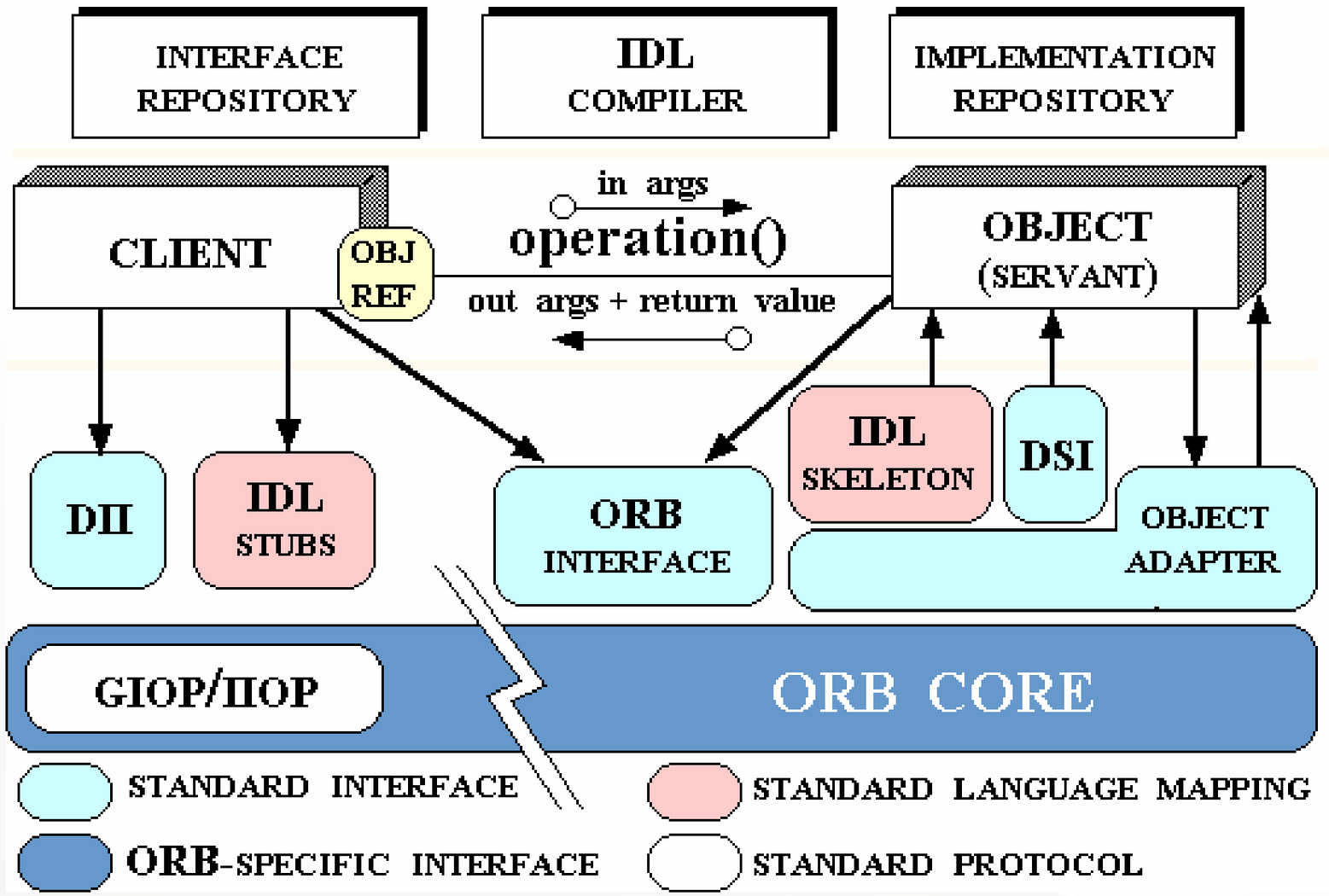
Implementation Repository

- Az implementációs szótár a megvalósított szerveroldali osztályok leírását tartalmazza, valamint információt arról, hogy melyik objektumokat példányosították és azok hogyan azonosíthatóak.

ORB interface

- Az ORB felület, amely a szerveroldalról is elérhető, megfelel a kliensoldalinak.

CORBA összetevők



Forrás: D.C. Schmidt, Overview of CORBA

Interface Definition Language

- A CORBA erősen objektumorientált szemléletű rendszer, ennek megfelelően az egyes szoftver-komponenseket osztályokként definiáljuk, és használhatunk osztályhierarchiákat, kivételkezelést stb.
- Az adatrejtésnek kiemelt szerepe van: az objektumok közötti kapcsolattartás csak az osztályok jól definiált interfészén keresztül lehetséges.
- Az interfészek meghatározására az **IDL** (Interface Definition Language) nyelvet használjuk. Az IDL hordozható, programozási nyelvtől, operációs rendszertől független nyelv. Szintaxisának gyökerei a C++ -hoz vezethetőek vissza, de a mögöttes elvek a Java által kitűzött célokhoz szorosabban kötődnek. Ennek megfelelően az IDL alaptípusainak és típuskonstrukcióinak leképezése a Java nyelvre viszonylag egyszerű.
- Az IDL deklaratív nyelv, azaz támogatja típusok, konstansok, adatelemek, metódusok, kivételek deklarációját, de nem tartalmaz procedurális elemeket, azaz nem foglalkozik azzal, hogy az eljárásokat hogyan kell implementálni.
- Az IDL segítségével deklarált interfészeket később megvalósítjuk valamilyen konkrét programozási nyelvben.

IDL-Java

- Az IDL nyelven deklarált interfészek konkrét megvalosítása Java-ban két lépésben történik: az IDL nyelven leírt interfészekből az IDL-fordító készít Java kódot, amelyet ki kell egészítenünk a metódusok - szintén konkrét nyelvi - megvalósításával.
- Az IDL-Java fordító IDL interfészforrást olvas és Java kódot hoz létre.
- Az IDL interfész-ből létrehozott Java interfész mellett generálásra kerülnek a csonkok (stub), vázak (szkeletonok), és egyéb állományok is.
- Az IDL-Java fordítókat az egyes ORB gyártók biztosítják termékeikhez. A Java 2 Platform, Standard Edition tartalmazza az **idlj** IDL fordítót.

IDL alaptípus	Java típus
`char,wchar`	`char`
`short`	`short`
`long`	`int`
`long long`	`long`
`unsigned short`	`short`
`unsigned long`	`int`
`unsigned long long`	`long`
`float`	`float`
`double`	`double`
`octet`	`byte`
`boolean`	`boolean`
`string,wstring`	`java.lang.String`

IDL paramétertípusok Java leképezése

CODESPRING
OUR CODE YOUR SUCCESS

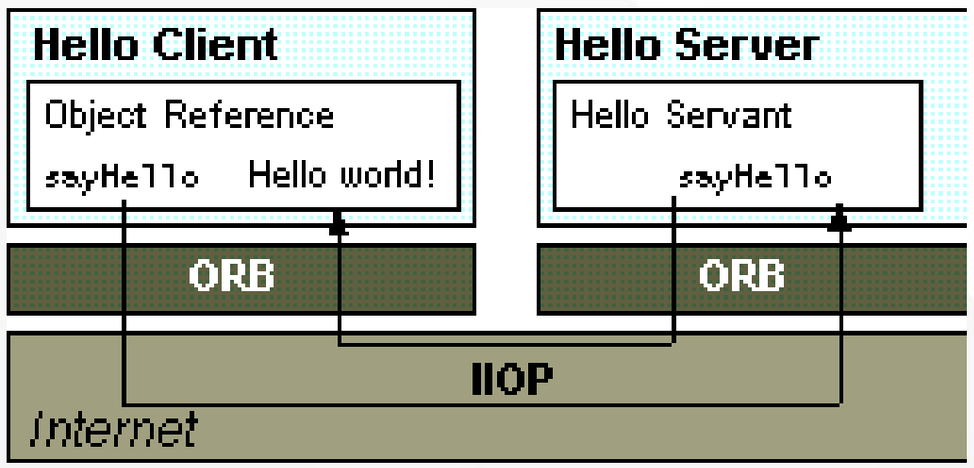
IDL típus	in	inout	out	visszatérő érték
short	short	ShortHolder	ShortHolder	short
long	int	IntHolder	IntHolder	int
long long	long	LongHolder	LongHolder	long
unsigned short	short	ShortHolder	ShortHolder	short
unsigned long	int	IntHolder	IntHolder	int
unsigned long long	long	LongHolder	LongHolder	long
float	float	FloatHolder	FIoatHolder	float
double	double	DoubleHolder	DoubleHolder	double
boolean	boolean	BooleanHolder	BooleanHolder	boolean
char	char	CharHolder	CharHolder	char
octet	byte	ByteHolder	ByteHolder	byte
any	Any	Any	Any	Any
enum	int	IntHolder	IntHolder	int
referencia	Object Ref	típusHolder	típusHolder	ObjectRef
struct	class	class	class	class
union	class	class	class	class
string	String	StringHolder	StringHolder	String
sequence	class	class	class	class
array	array	array	array	array

CORBA – további lehetőségek

- Szerver visszahívási (callback) lehetőséget is ad a CORBA. A hagyományos kliens-szerver elképzelés szerint mindig a kliens a kezdeményező, aki aktivizál valamilyen szerver folyamatot. Ha a kliens nem fordul a szerverhez, az nem képes információt szolgáltatni. A visszahívási technika lehetőséget ad arra, hogy a (klienssel párhuzamosan futó) szerverfolyamat legyen a kezdeményező, és az hívja meg a kliens egy metódusát (ez a metódus Java kliens esetében legtöbbször egy külön szálon belül van implementálva).
- Nem csak statikusan - előre definiált műveletekkel - kommunikálhatunk a CORBA objektumok között, hanem használhatjuk a dinamikus hívási felületet (Dynamic Invocation Interface) is. A DII segítségével a kliens képes futási időben kiválasztani azokat a szerveroldali objektumokat és metódusokat, amelyekkel együtt kíván működni. Ez azt jelenti, hogy a kliens fordítási idő helyett futási időben is képes az IDL interfészt lekérdezni, metódusok között válogatni, egy kiválasztott metódus leírását megszerezni, majd a metódust meghívni.
- Bár a Java ezt nem támogatja, használható a többszörös öröklés

Java-CORBA HelloWorld

- A kliens oldali applikáció tartalmaz egy a távoli (remote) objektumra mutató referenciát. A referenciához tartozik egy stub metódus, ez továbbítódik az ORB-hoz, amely továbbítja a kérést a szerverhez.
- A szerver oldalon az ORB szkeleton kódot alkalmaz a távoli hívás fordításához, egy lokális metódushívásba alakítva azt. A szkeleton implementáció specifikus formátumba fordítja le a hívást és a paramétereket. A metódus visszatérésekor a szkeleton átfordítja a visszatérített eredményt vagy hibát és az ORB-on keresztül visszaküldi azt a kliensnek.
- A kommunikáció a TCP/IP alapú, OMG által definiált IIOP (Internet Inter-ORB Protocol) protokollon keresztül történik.



HelloWorld példa lépései

- A Hello.idl interfész létrehozása:

```
module HelloApp {
interface Hello {
   string sayHello();
   oneway void shutdown();
   };
};
```

- A Hello.idl leképezése Java-ra az idlj fordító segítségével:

```
idlj -fall Hello.idl
```

A –fall opció alkalmazása miatt a szerver oldali szkeletonok is legenerálódnak (egyébként csak a kliens oldali rész).

- A létrejött HelloApp könyvtár 6 állományt fog tartalmazni, közöttük a Hello.java állományt:

```
//Hello.java
package HelloApp;
/**
* HelloApp/Hello.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hello.idl
*/
public interface Hello extends HelloOperations,
          org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {        }
```

- A module leképezése package, az interface leképezése interface.

HelloWorld példa lépései

- Az interfészben deklarált műveletek leképezése egy külön állományt hoz létre:

```
//HelloOperations.java
package HelloApp;
/**
* HelloApp/HelloOperations.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hello.idl
*/
public interface HelloOperations {
                String sayHello ();
                void Shutdown ();
        } // interface HelloOperations
```

- A **HelloPOA.java** absztrakt osztály egy adatfolyam alapú szerver szkeleton, amely az alap CORBA funkcionalitásokat biztosítja.

  Az org.omg.PortableServer.Servant kiterjesztése, és implementálja az InvokeHandler valamint a HelloOperations interfészeket. A HelloServant szerver osztály a HelloPOA osztálynak lesz a kiterjesztése.

- A **_HelloStub.java** osztály, a kliens stub az alapvető kliens oldali CORBA funkcionalitásokat biztosítja.

  Az org.omg.CORBA.portable.ObjectImpl kiterjesztése és implementálja a Hello.java interfészt.

HelloWorld példa lépései

- A **HelloHelper.java** osztály járulékos funkcionalitásokat biztosít, például a CORBA referenciák megfelelő típusokba történő konvertálásához szükséges narrow() metódust. A Helper osztály felelős a CORBA adatfolyamokba történő írásért/olvasásért, valamint az Any típus kezeléséért.
- A **HelloHolder.java** final osztály tartalmaz egy publikus Hello példányt. Az Input/Output streamek kezeléséért felelős, írásra és olvasására delegálhat adatokat a Helper megfelelő metódusaihoz. Implementálja az org.omg.CORBA.portable.Streamable interfészt.
- **Portable Object Adaptor (POA)**: A J2SE 1.4 –től kezdve az idlj alapértelmezetten (a –fall opció alkalmazásával) POA-t alkalmaz. Ennek előnyei közé tartozik, hogy az objektumok implementációja „hordozható” különböző ORB termékek között, valamint lehetővé teszi, hogy egyetlen szerver oldali servant objektum több szerver oldali objektum párhuzamos kezelésére legyen alkalmas.
- A következő lépés a **HelloClient.java** kliens és a **HelloServer.java** szerver osztályok létrehozása.

HelloClient.java

```
import HelloApp.*;                    // the package containing our stubs
import org.omg.CosNaming.*;           // HelloClient will use the Naming Service
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;               // All CORBA applications need these classes

public class HelloClient {
     static Hello helloImpl;
     public static void main (String[] args) {
       try{
            // create and initialize the ORB
            ORB orb = ORB.init (args, null);

            // get the root naming context
            org.omg.CORBA.Object objRef = orb.resolve_initial_references ("NameService");

            // Use NamingContextExt instead of NamingContext. This is
            // part of the Interoperable naming Service.
            NamingContextExt ncRef = NamingContextExtHelper.narrow (objRef);

            // resolve the Object Reference in Naming
            String name = "Hello";
            helloImpl = HelloHelper.narrow (ncRef.resolve_str (name));
            System.out.println ("Obtained a handle on server object: " + helloImpl);
            System.out.println (helloImpl.sayHello ());
            helloImpl.shutdown ();
       } catch (Exception e) {
            System.out.println ("ERROR : " + e) ;
            e.printStackTrace (System.out);
       }
     }
}
```

HelloServer.java

```
import HelloApp.*;                    // The package containing our stubs
import org.omg.CosNaming.*;           // HelloServer will use the naming service

// The package containing special exceptions thrown by the name service
import org.omg.CosNaming.NamingContextPackage.*;

import org.omg.CORBA.*;    // All CORBA applications need these classes
// Classes needed for the Portable Server Inheritance Model
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.util.Properties;          // Properties to initiate the ORB

//the class for the servant object
class HelloImpl extends HelloPOA {
     private ORB orb;
     public void setORB(ORB orb_val) {
             orb = orb_val;
     }
     // implement sayHello() method
     public String sayHello() {
             return "\nHello world !!\n";
     }
     // implement shutdown() method
     public void shutdown() {
             orb.shutdown(false);
     }
}
```

HelloServer.java

```
public class HelloServer {
  public static void main(String[] args) {
    try {
            // create and initialize the ORB
            ORB orb = ORB.init (args, null);
            // get reference to rootpoa & activate the POAManager
            POA rootpoa = POAHelper.narrow (orb.resolve_initial_references("RootPOA"));
            rootpoa.the_POAManager ().activate ();
            // create servant and register it with the ORB
            HelloImpl helloImpl = new HelloImpl ();
            helloImpl.setORB (orb);
            // get object reference from the servant
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference (helloImpl);
            Hello href = HelloHelper.narrow (ref);
            // get the root naming context
            org.omg.CORBA.Object objRef = orb.resolve_initial_references ("NameService");
            // Use NamingContextExt (is part of the Interoperable Naming Service specification).
            NamingContextExt ncRef = NamingContextExtHelper.narrow (objRef);
            // bind the Object Reference in Naming
            String name = "Hello";
            NameComponent[] path = ncRef.to_name (name);
            ncRef.rebind (path, href);
            System.out.println ("HelloServer ready and waiting ...");
            // wait for invocations from clients
            orb.run ();
    } catch (Exception e) {
            e.printStackTrace ();
    }
    System.out.println("HelloServer Exiting ...");
  }
}
```

HelloWorld példa futtatása

- Az Object Request Broker Daemon (orbd) indítása:
  `start orbd -ORBInitialPort 1050 -ORBInitialHost localhost`

- A HelloServer indítása:
  `start java HelloServer -ORBInitialPort 1050 -ORBInitialHost localhost`

- A HelloClient kliens indítása:
  `java HelloClient -ORBInitialPort 1050 -ORBInitialHost localhost`

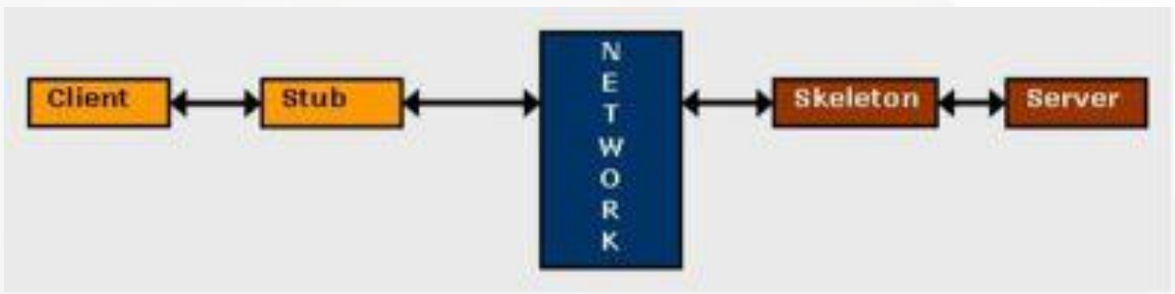
- A kliens a konzolon a HelloWorld! feliratot jeleníti meg

RMI

JAVA REMOTE METHOD INVOCATION API

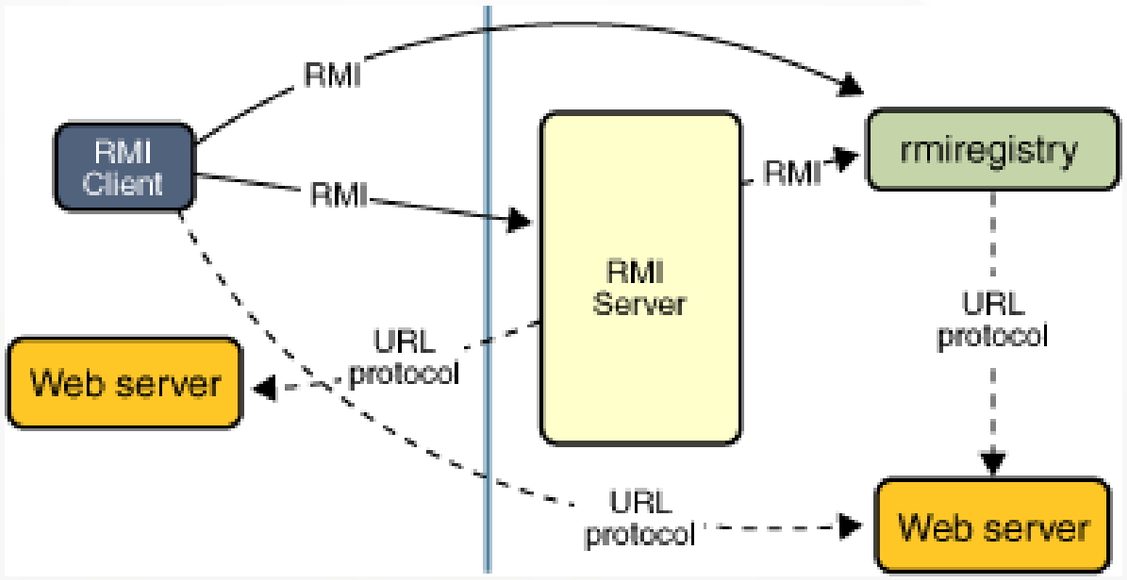
RMI

- Java **RMI** API (Java **Remote Method Invocation** API): távoli (remote) objektumok metódusainak meghívását lehetővé tevő fejlesztői interfész (API).
- Kétfajta implementációja ismert: az első a Java alapú JRMP (Java Remote Method Protocol), amely a különböző JVM-ek közötti kommunikációt teszi lehetővé, a másik (aktuálisan már szélesebb körben alkalmazott) CORBA alapú RMI-IIOP (RMI over IIOP), mely funkcionalitásait CORBA implementáción keresztül biztosítja.
- A Java csomag neve: java.rmi
- Az RMI alkalmazások két részből állnak: a szerver (remote) objektumokat példányosít, majd várja és fogadja a kliensek kéréseit. A kliens referenciákat kér a remote objektumokhoz való hozzáféréshez, majd meghívja ezen objektumok szükséges metódusait. Az RMI alapú alkalmazások tehát osztott objektum alkalmazások (distributed object application).



RMI

- Distributed object application működése:
  - A remote objektumok lokalizálása: referenciák kérése a remote objektumokhoz, pl. az RMI registry (az RMI naming service) segítségével, vagy más remote metódushívásokon belül.
  - Kommunikáció a remote objektumokkal: az RMI-n keresztül valósul meg, a fejlesztő számára hasonló az egyszerű metódushívásokhoz.
  - A remote objektumoknak megfelelő osztálydefiníciók betöltése.



Forrás: oracle.com

- A szerver a registry segítségével egy névvel asszociálja a remote objektumot.
- A kliens az objektum neve alapján megkeresi az objektumot a registry segítségével, majd meghívja annak valamelyik metódusát.
- Egy webserver biztosítja az osztály-definíciók beolvasását.

RMI

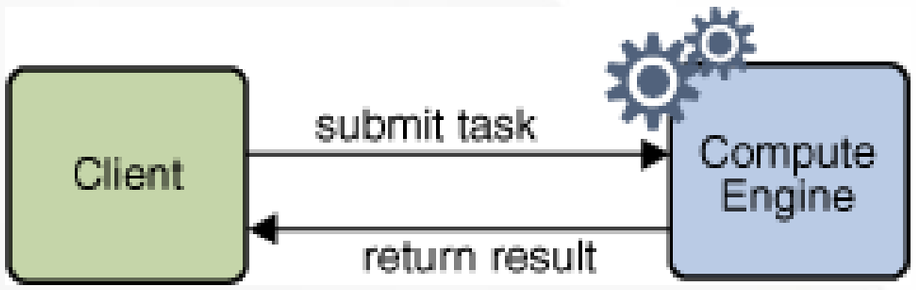
- Az RMI egyik előnye a dinamikus kódbetöltés: akkor is betölthető egy objektum osztálydefiníciója, ha a kliens JVM-en belül nem definiált az illető osztály. Ilyen módon az alkalmazások viselkedése dinamikusan kiegészíthető (új típusok bevezetésével).
- Remote interfészek, objektumok és metódusok:
  - Egy objektum a remote interfész implementálásával válhat remote objektummá. A remote interfész a java.rmi.Remote interfész kiterjesztése, és minden interfészen belüli metódus java.rmi.RemoteException kivételt generálhat.
  - A remote objektumok esetében a kliens (fogadó) JVM nem készít másolatot az objektumról, csak egy annak megfelelő stub-ot hoz létre. A stub lesz az illető objektum lokális reprezentációja, proxy, amely tulajdonképpen a kliens számára remote referencia – a kliens ezen keresztül hívhatja meg a remote metódusokat.
  - A stub a remote objektum által implementált interfészek mindegyikét implementálja, tehát bármelyik típusba konvertálható (cast).
- Egy RMI alkalmazás fejlesztésének lépései:
  - Az osztott alkalmazás komponenseinek megtervezése és implementációja.
  - A forráskódok fordítása.
  - Az osztályok elérhetővé tétele hálózaton keresztül.
  - Az alkalmazás futtatása.

A fejlesztés lépései

- A komponensek tervezése és implementálása:
  - Az applikáció architektúrájának megtervezése: a lokális és remote objektumok azonosítása.
  - A remote interfészek létrehozása: a remote interfészek deklarálják azokat a metódusokat, amelyeket a kliens meghívhat távolsági (remote) metódushívások segítségével. A kliens program az interfészeken keresztül kommunikál, nem az azokat implementáló osztályok példányain keresztül. A remote interfészek tervezése magába foglalja a paraméterként átadott és a metódusok által visszatérített objektumok típusának meghatározását.
  - A remote objektumok osztályainak létrehozása: a remote objektumoknak megfelelő osztályok implementálják a remote interfészeket (ezeken kívül további, csak lokálisan elérhető interfészeket is implementálhatnak).
  - A kliens megvalósítása.
- A forráskód fordítása: egyszerűen a javac segítségével (Java Platform SE 5.0 megjelenése előtt az rmic fordítót is használni kellett a stubok létrehozásához, ez ma már nem szükséges).
- Az osztályok elérhetővé tétele a hálózaton keresztül (általában webszerverek segítségével).
- Az alkalmazás futtatása: egyaránt szükséges a szerver és RMI registry, majd a kliens futtatása.

RMI példa

- RMI példa: „Compute Engine” (számítási motor): remote objektum, amely a kliensektől különböző feladatokat (task) kap, elvégzi a feladatokat és visszafordítja a megfelelő eredményeket (mivel a feladatok elvégzése a szerveren történik, egy ilyen alkalmazás lehetővé teheti több kliens hozzáférését egy nagykapacitású szerverhez, vagy speciális hardware-ekhez).
- Új task létrehozása bármikor lehetséges - csak annyi az elvárás, hogy az illető task implementálja a megfelelő interfészeket. Az RMI dinamikusan tölti be a taskokat a számítási motort futtató JVM-be, majd futtathatja azokat anélkül, hogy bármilyen *a priori* ismeretekkel rendelkezne az illető task-ot implementáló osztályról.
- Az ilyen dinamikus kódbetöltésre képes alkalmazásokat viselkedés alapú (behavior based) alkalmazásoknak is nevezik.



- A szerver kódját egy interfész és egy osztály alkotja. Az interfész deklarálja a kliens által meghívható metódusokat, az osztály biztosítja az implementációt

RMI példa – szerver

- A szerver oldali Compute interfész egyetlen metódust tartalmaz - ez teszi lehetővé, hogy a kliens átküldjön a szervernek egy bizonyos task-ot:

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface Compute extends Remote {
          <T> T executeTask(Task<T> t) throws RemoteException;
}
```

- A Compute Engine –nek még szüksége van a Task interfészre – ilyen típusú az executeTask metódus paramétere. Az interfész egyetlen metódust tartalmaz:

```
package compute;
public interface Task<T> {
          T execute();
}
```

  A T típusparaméter – a visszafordított érték típusa
- Az executeTask is rendelkezik egy saját T típusparaméterrel, amely lehetővé teszi, hogy a paraméterként kapott Task objektum által visszatérített eredményt visszaküldje a kliensnek.
- A Compute Engine bármilyen taskot (számítást) elvégez, amennyiben az illető task implementálja a Task interfészt. Az interfészt implementáló osztályok tartalmazhatnak bármilyen, a számítások elvégzéséhez szükséges adatokat, metódusokat.

RMI példa – szerver

- Az RMI a Java Serialization mechanizmusát használja az objektumok JVM-ek közötti érték szerinti küldéséhez.
- A remote metódusok argumentumai és a visszafordított típusok lehetnek remote vagy lokális objektumok, vagy primitív adattípusok. A lokális objektumok osztályainak meg kell valósítaniuk a Serializable interfészt.
- A remote objektumok átadása referencia segítségével történik, a referencia a stub, a kliens oldali proxy, amely megvalósítja a remote objektum osztálya által implementált összes interfészt.
- A lokális objektumok átadása másolatok segítségével történik: a static vagy transient adattagokon kívül minden lemásolódik.
- A remote objektumok referencia általi átadása azt eredményezi, hogy remote metódushívások az eredeti objektum állapotában is tükröződnek (a nem remote objektumok esetében másolat készül – itt a változtatások csak ezeket a kliens oldali másolatokat érintik).
- A remote objektumok átadásánál a kliens számára csak a remote interfészek által deklarált metódusok láthatóak.

A ComputeEngine osztály

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;
public class ComputeEngine implements Compute {
    public ComputeEngine () {
            super ();
    }
    public <T> T executeTask (Task<T> t) {
            return t.execute ();
    }
    public static void main (String[] args) {
            if (System.getSecurityManager () == null) {
                    System.setSecurityManager (new SecurityManager ());
            }
            try {
                    String name = "Compute";
                    Compute engine = new ComputeEngine ();
                    Compute stub = (Compute) UnicastRemoteObject.exportObject (engine, 0);
                    Registry registry = LocateRegistry.getRegistry ();
                    registry.rebind (name, stub);
                    System.out.println ("ComputeEngine bound");
            } catch (Exception e) {
                    System.err.println ("ComputeEngine exception:");
                    e.printStackTrace ();
            }
    }
}
```

RMI példa – szerver

- Az RMI szerverprogram először példányosítja a remote objektumokat, majd exportálja azokat az RMI runtime-nak, ettől kezdve azok fogadhatnak a kliensektől érkező remote metódushívásokat. Ez a leírt „setup” procedúra, amely lehet a remote objektum metódusa, vagy egy külső osztály része, a következő lépéseket hajtja végre:
  - Security manager létrehozása és telepítése.
  - Egy vagy több remote objektum létrehozása és exportálása.
  - A remote objektumok regisztrálása az RMI registry (vagy más naming service) segítségével.
- Az első lépés a SecurityManager telepítése, amely megvédi a rendszert a nem megbízható kódok betöltésétől – hiánya azt eredményezi, hogy az RMI csak a lokális classpath-ből tölt be kódokat.
- A következő lépés a remote objektum (esetünkben ComputeEngine) létrehozása, majd exportálása az UnicastRemoteObject osztály exportObject statikus metódusával, melynek második paramétere a TCP port meghatározása (a 0 érték anonymous portot jelent – a konkrét portot futási időben azonosítja a rendszer - gyakran alkalmazzák), ezen keresztül fogadhat az objektum a kliensektől érkező remote metódushívásokat.
- A kliensnek szüksége lesz a remote objektumra mutató referenciára – ezt az objektum nevének és az RMI registry-nek a segítségével szerezheti meg.

RMI példa – szerver

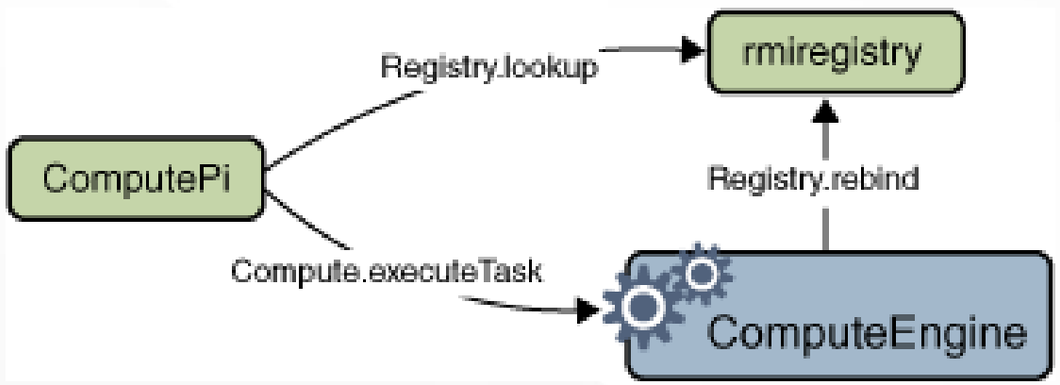
- A LocateRegistry osztály definiál az aktuális registry lekéréséhez, vagy új registry létrehozásához alkalmazható statikus metódusokat (a registry mindig egy adott IP címhez és porthoz van hozzárendelve – azonos host-on futó szerverek megoszthatnak egymás között registry-ket, vagy mindenik létrehozhatja/használhatja a sajátját).
- Az alapértelmezett rmiregistry port: 1099.
- Az objektum nevének regisztrálása a registry rebind metódusával történik, ez gyakorlatilag egy a lokális host-on futó registrynek címzett remote metódushívás (RemoteException-t eredményezhet). A rebind metódushívásnál paraméterként átadódik a remote objektumnak megfelelő stub is.
- Biztonsági okokból egy alkalmazás csak a lokális registryn belül köthet le vagy törölhet objektumokat (bind/unbind/rebind) – a registry csak lokálisan futó alkalmazás által módosítható, de bárhonnan lekérdezhető.
- A regisztrálás után az objektum elérhetővé válik, nem szükséges külön szál létrehozása a kérések, metódushívások fogadásához, mivel a kliensek a registryn keresztül érik el az objektumot. Az objektum nem törlődhet (a garbage collector nem törli) mindaddig, amíg nem töröljük a registry-ből (unbind) ÉS ameddig kliensek rámutató referenciával rendelkeznek.

A ComputeEngine osztály

```
package engine;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;
public class ComputeEngine implements Compute {
    public ComputeEngine () {
            super ();
    }
    public <T> T executeTask (Task<T> t) {
            return t.execute ();
    }
    public static void main (String[] args) {
            if (System.getSecurityManager () == null) {
                    System.setSecurityManager (new SecurityManager ());
            }
            try {
                    String name = "Compute";
                    Compute engine = new ComputeEngine ();
                    Compute stub = (Compute) UnicastRemoteObject.exportObject (engine, 0);
                    Registry registry = LocateRegistry.getRegistry ();
                    registry.rebind (name, stub);
                    System.out.println ("ComputeEngine bound");
            } catch (Exception e) {
                    System.err.println ("ComputeEngine exception:");
                    e.printStackTrace ();
            }
    }
}
```

RMI példa – kliens

- Példánkban a kliens egy konkrét task-ot definiál: a PI (π) értékét fogja kiszámolni egy bizonyos pontossággal. Ehhez két osztályra lesz szükségünk a ComputePi osztály megkeresi a Compute objektumot, majd meghívja annak executeTask metódusát átadva egy konkrét task-ot. Esetünkben a Pi osztály implementálja a task-ot (a számítást – esetünkben a PI kiszámítását).



- A ComputePi osztály tartalmazza a main metódust, a parancssor argumentumaként megkapja az RMI registry-t és szerver alkalmazást futtató host nevét, valamint a PI kiszámításánál elvárt pontosságot.
- A tény, miszerint a paraméterként átadott task a PI kiszámítását végzi, irreleváns a szerver számára – hasonlóan bármilyen más task is átadható.

A ComputePi osztály

```
package client;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String[] args) {
            if (System.getSecurityManager() == null) {
                    System.setSecurityManager (new SecurityManager ());
            }
            try {
                    String name = "Compute";
                    Registry registry = LocateRegistry.getRegistry (args[0]);
                    Compute comp = (Compute) registry.lookup (name);
                    Pi task = new Pi(Integer.parseInt (args[1]));
                    BigDecimal pi = comp.executeTask (task);
                    System.out.println (pi);
            } catch (Exception e) {
                    System.err.println ("ComputePi exception:");
                    e.printStackTrace ();
            }
    }
}
```

A Pi osztály – I.

```
package client;
import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {
    private static final long serialVersionUID = 227L;

    /** constants used in pi computation */
    private static final BigDecimal FOUR = BigDecimal.valueOf (4);
    /** rounding mode to use during pi computation */
    private static final int roundingMode = BigDecimal.ROUND_HALF_EVEN;
/** digits of precision after the decimal point */
private final int digits;

/**
* Construct a task to calculate pi to the specified precision.
*/
public Pi (int digits) {
       this.digits = digits;
}
/**
* Calculate pi.
*/
public BigDecimal execute () {
       return computePi (digits);
}
```

A Pi osztály – II.

```
/**
* Compute the value of pi to the specified number of
* digits after the decimal point. The value is
* computed using Machin's formula:
*
*    pi/4 = 4*arctan(1/5) - arctan(1/239)
*
*and a power series expansion of arctan(x) to
* sufficient precision.
*/
public static BigDecimal computePi (int digits) {
     int scale = digits + 5;
     BigDecimal arctan1_5 = arctan (5, scale);
     BigDecimal arctan1_239 = arctan (239, scale);
     BigDecimal pi = arctan1_5.multiply (FOUR).subtract ( arctan1_239).multiply (FOUR);
     return pi.setScale(digits, BigDecimal.ROUND_HALF_UP);
}
```

A Pi osztály – III.

```
/**
* Compute the value, in radians, of the arctangent of the inverse of the
* supplied integer to the specified number of digits after the decimal point.
* The value is computed using the power series expansion for the arc tangent:
* arctan(x) = x - (x^3)/3 + (x^5)/5 - (x^7)/7 + (x^9)/9 ...
*/
public static BigDecimal arctan (int inverseX, int scale) {
    BigDecimal result, numer, term;
    BigDecimal invX = BigDecimal.valueOf (inverseX);
    BigDecimal invX2 = BigDecimal.valueOf (inverseX * inverseX);
    numer = BigDecimal.ONE.divide (invX, scale, roundingMode);
    result = numer;
    int i = 1;
    do {
            numer = numer.divide (invX2, scale, roundingMode);
            int denom = 2 * i + 1;
            term = numer.divide (BigDecimal.valueOf (denom), scale, roundingMode);
            if ((i % 2) != 0) {
                    result = result.subtract (term);
            } else {
                    result = result.add (term);
            }
            i++;
    } while (term.compareTo (BigDecimal.ZERO) != 0);
    return result;
 }
}
```

RMI példa – kliens

- A forráskódok fordítása után, mivel néhány osztály dinamikusan (runtime) lesz betöltve, és hálózati hozzáférés szükséges, az osztályokat (az azokat tartalmazó csomagokat – lehetőleg külön jar állományok) egy hálózaton keresztül elérhető helyre kell másolnunk.
- Az RMI nem használ speciális protokollokat, a Java által támogatott URL protokollokat (pl. HTTP) alkalmazza. Az osztályok RMI-n keresztüli letöltéséhez természetesen speciális web-szerverekre van szükség. Például:

  http://java.sun.com/javase/technologies/core/basic/rmi/class-server.zip

- A Security manager részére létre kell hoznunk a kliens és szerver oldali policy fájlokat (server.policy és client.policy):

```
grant codeBase "file:/home/ann/src/" {
  permission java.security.AllPermission;
};
```

```
grant codeBase "file:/home/jones/src/" {
  permission java.security.AllPermission;
};
```

- Ezután az RMI registry indítása következik, Windows alatt:

  `start rmiregistry` vagy **`start rmiregistry <port>`**

- A szerver indítását követheti a kliens indítása.
- Eclipse RMI plugin: http://www.genady.net/rmi/v20/index.html