RMI - Remote Method Invocation

Cíl cvičení:

Cílem je seznámit se s procedurální komunikací Java RMI a procvičit si použití vývojového prostředí Eclipse.

Cvičení zahrnuje ukázkový příklad *Hello*, jeho zdrojové texty najdete na www stránce předmětu X36DSV (tedy tam, odkud jste získali tento text). Popis mechanismu Java RMI a jeho podpůrných prostředků je v příloze tohoto textu, stejně jako podrobný popis překladu a spuštění ukázkového programu.

Postup:

Pro editaci a překlad zdrojového textu komponent aplikace využijte prostředí Eclipse. Spuštění generátoru stubů *rmic* a programu *rmiregisty* je možné z příkazové řádky, podobně je rozumné (s ohledem na poměrně vysoké paměťové nároky systému Eclipse) spouštět z příkazové řádky i odladěný server.

Samostatná práce:

Součástí cvičení je individuální návrh jednoduché aplikace, dovolující například realizovat přístup ke vzdálené tabulce (seznam knih, výsledky hokejových zápasů), zjistit informace o vzdáleném systému (např. výpis aktuálního adresáře) nebo zajistit jednoduchý výpočet.

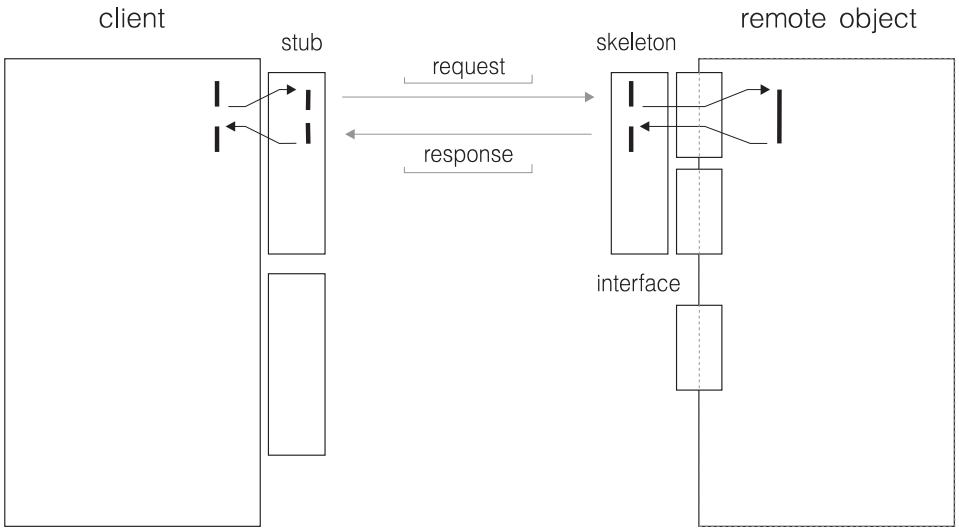
Pro svou aplikaci se můžete pokusit o analýzu RMI protokolu (s využitím programu tcpdump nebo ethreal).

A Procedurální komunikace - RMI

Při programování distrubuovaných aplikací v jazyce Java můžeme využít proprietární (tedy vázaný na tento programovací jazyk) procedurální komunikační nástroj - *RMI - Remote Method Invocation*.

Výhodou java RMI proti jiným formám procedurální komunikace je jednoduchost, a možnost přenášet objekty (včetně jejich kódu). Nevýhodou je použitelnost omezená na jazyk Java.

Strukturu mechanismu RMI si lze popsat na obr. 1.



Obrázek 1: Komunikace RMI

Mechanismus RMI zajišťuje komunikaci mezi *klientem* (klientskou částí aplikace), kterým může být libovolný objekt jazyka Java (přesněji vlákno výpočtu, které interpretuje kód nějakého objektu) a *serverem*, kterým může být libovolný *vzdálený objekt (Remote Object)* realizující určitou obsluhu. Ten se liší od běžného objektu (z kterého je odvozen) v některých předefinovaných metodách; základním rozdílem je umožnění vzdáleného přístupu pro klienta.

Komunikaci mezi klientem a vzdáleným objektem definuje *rozhraní RMI*. To je výčtem metod, které vzdálený objekt pro klienta zajišťuje, a které musí být vzdáleným objektem implementovány. Vzdálený objekt může současně implementovat více různých rozhraní RMI.

Klient může získat odkaz na zpřístupněný vzdálený objekt prostřednictvím systémové aplikace *rmiregistry* dovolující registrovat vzdálený objekt na daném počítači pod zvoleným jménem, další možností je získat odkaz jako výsledek jiného vzdáleného volání.

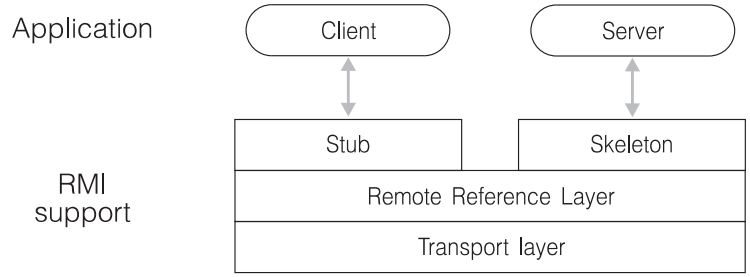
Klient spolupracuje s lokálním zástupcem vzdáleného objektu - ten označujeme jako *stub*. Aktivaci metod rozhraní RMI (označovanou jako *Remote Method Invocation*) budeme označovat jako *vzdálená volání*. Na rozdíl od běžné aktivace metody mohou vzdálená volání překlenout hranici mezi logickými stroji JVM (Java Virtual Machine); běh klienta a serveru na různých strojích však není podmínkou.

Při vzdáleném volání jsou předávány parametry a výsledek jako posloupnosti oktetů. O transformaci běžné aktivace metody na takovou posloupnost (používáme zde termínů *marshalling* a *unmarshalling*) se stará *stub* na straně klienta. Odpovídající stub na straně vzdáleného objektu (ten převádí posloupnost zpět na volání) je označován jako *stub serveru* nebo *skeleton*. Stuby mohou předávat primitivní typy, vzdálené objekty (přesněji odkazy na ně) a objekty (jako hodnoty), které implementují rozhraní *java.io.Serializable* (a jsou tedy převeditelné na posloupnost oktetů). Pokud parametr nebo výsledek vzdáleného volání nepatří do některé z uvedených skupin, končí vzdálené volání výjimkou.

Schéma na obr. 1 kromě synchronního charakteru volání RMI zdůrazňuje skutečnost, že skeleton je svázán s třídou popisující vzdálený objekt (zahrnuje všechna implementovaná rozhraní) a že klient může spolupracovat s více vzdálenými objekty jedné nebo více tříd. O vytvoření stubu a skeletonu se postará generátor *rmic*.

Pozn:
Skeleton pro vzdálený objekt je generován programem *rmic* pouze při volbě staršího RMI protokolu (RMI 1.1). V případě v současnosti implicitní volby poněkud odlišného protokolu RMI 1.2 je generován pouze stub (klienta). V případě nutnosti podpořit i klienty pracující s protokolem RMI 1.1 máme k dispozici kompatibilní mód.

Mechanismus komunikace mezi klientem a vzdáleným objektem je nejviditelnější vrstvou RMI systému, celkovou strukturu RMI systému si můžeme popsat oblíbeným schématem vrstev - obr. 2.



Obrázek 2: Architektura systému RMI

Pod vrstvou komunikace mezi stubem klienta a skeletonem vzdáleného objektu se skrývá velice zajímavá vrstva označovaná jako *Remote Reference Layer*. Ta je zodpovědná za *sémantiku* vzdálených volání, tedy za způsob zajištění proti výpadku komunikace a havárii výpočtu vzdáleného objektu (sémantiky *"at-most-once"*, *"at-least-once"* a *"exactly- once"*), za způsob synchronizace klienta a vzdáleného objektu (standardem je *synchronní* vyvolání vzdálené metody, *asynchronní* vyvolání není v současném systému podporováno) a za způsob implementace vzdáleného objektu (trvalá nebo dočasná *aktivita* vzdáleného objektu, správa *persistentních* vzdálených objektů, *replikační strategie* a *mobilita*).

Konečně, celý RMI mechanismus je podporován *transportní službou* zodpovědnou za navazování a správu komunikačních spojení. K podpoře RMI mechanismu konečně patří i registrace vzdálených objektů (v rámci jednoho stroje) schopných přijímat RMI volání - program *rmiregistry*.

Kromě dosud uvedených prvků, které přímo podporují spolupráci součástí distribuované aplikace, zahrnuje RMI systém prostředky podporující uvolňování paměti po vzdálených objektech (distribuované rozšíření čističe paměti - *garbage collectoru*), dynamické zavádění definic tříd (pro stuby, skeletony a vzdálená rozhraní) nejenom z lokálních systémů souborů (*java.lang.ClassLoader*), ale i ze sítě (*java.rmi.server.RMIClassLoader*) a ochranu systému i aplikace (*java.rmi.RMISecurityManager*).

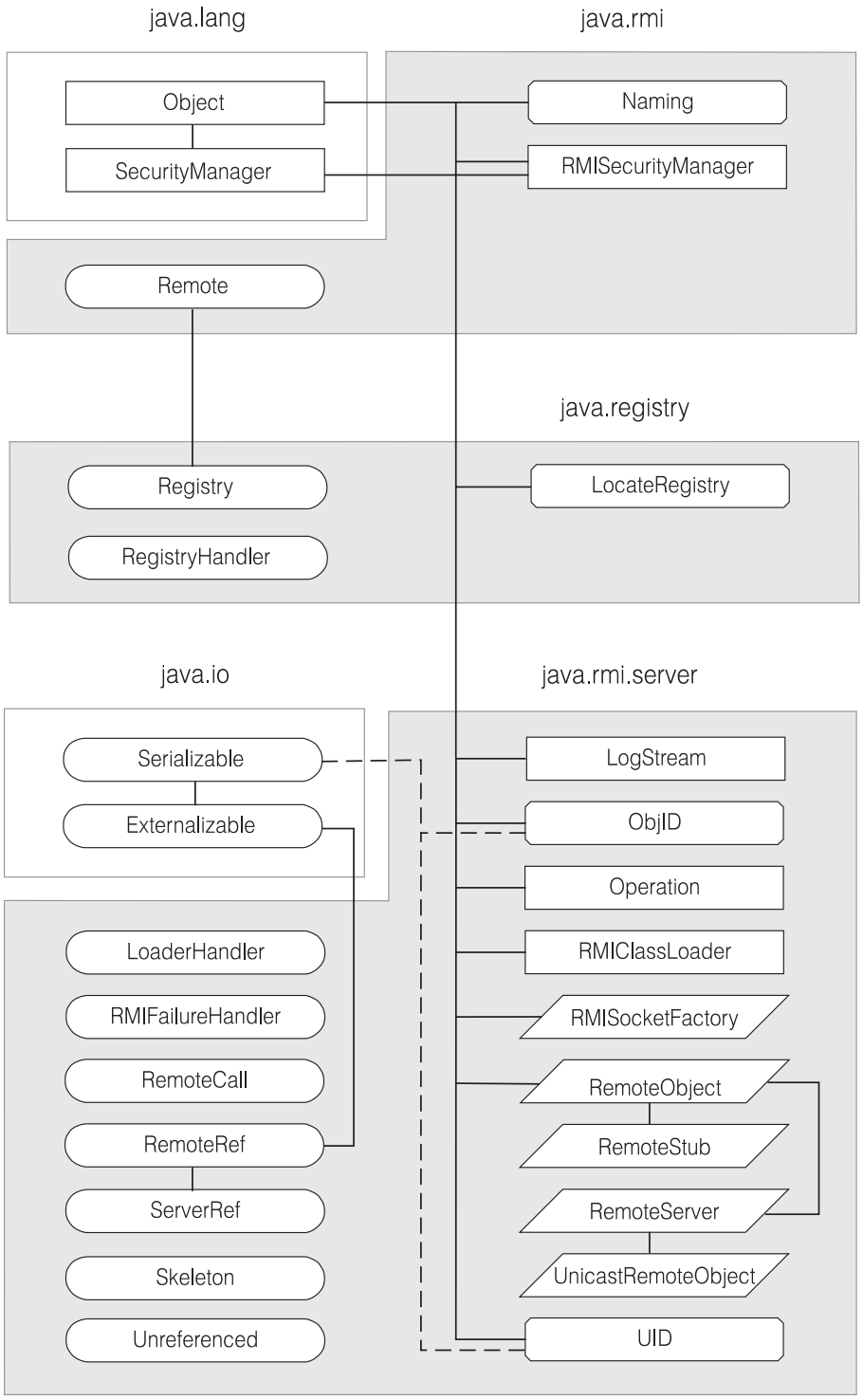
Ale vraťme se nyní k základnímu mechanismu RMI, ten je podporován třídami systémových knihoven *java.rmi*, *java.rmi.server*, *java.rmi.registry* a *java.rgi.DGM*. Strukturu těchto knihoven uvádí obr. 3. Popisu jednotlivých tříd této knihovny se nyní budeme věnovat podrobněji.

A.1 Rozhraní RMI

Rozhraní RMI pro konkrétní aplikaci, zpřístupňující metody vzdáleného objektu, odvodíme z rozhraní *java.rmi.remote*. Rozhraní *java.rmi.remote* má velice stručnou signaturu:

```
package java.rmi;
public interface Remote { }   ,
```

nezahrnuje tedy žádné veřejně přístupné metody.



Obrázek 3: Třídy RMI

Rozhraní RMI aplikace, z rozhraní *java.rmi.remote* odvozené, musí být veřejně dostupné (*public*). Každá z jeho metod musí mít v seznamu výjimečných ukončení (klauzule *throws*) uvedenu výjimku *java.rmi.RemoteException*, která je základem pro všechny výjimky související s RMI komunikací. To dovolí klientské části aplikace reagovat specifickým způsobem na výpadky v síťové komunikaci nebo na problémy vzniklé na vzdáleném objektu/serveru.

Na parametry a výsledek metod rozhraní RMI jsou kladena určitá, již uvedená, omezení. Nejdůležitějším je zřejmě skutečnost, že objekty lokálně dostupné klientské části aplikace jsou předávány *hodnotou* (kopírovány, do kopií pouze nejsou zahrnuty položky se specifikacemi *static* a *transient*). Vzdálené objekty jsou naproti tomu identifikovány *odkazem*, odkaz musí být navíc směrován na určité rozhraní RMI vzdáleného objektu a ne na vzdálený objekt samotný.

Jako ilustrační příklad definice rozhraní RMI si uvedeme uživatelské rozhraní bankovního účtu dovolující uložit (metoda *deposit*) nebo vyzvednout (metoda *withdraw*) zadaný obnos a zjistit (metoda *balance*) okamžitý stav účtu.

```
package bankaccount;
public interface BankAccount extends java.rmi.Remote {
   public void deposit(float amount)
          throws java.rmi.RemoteException;
   public void withdraw(float amount)
          throws OverDrawnException, java.rmi.RemoteException;
   public float balance()
          throws java.rmi.RemoteException;
}   .
```

Klientská část aplikace musí být schopna získat odkaz na RMI rozhraní vzdáleného objektu identifikovaného doménovým jménem nebo IP adresou počítače a výlučným jménem tohoto objektu na tomto počítači. Vytvoření potřebné vazby vzdáleného objektu na jméno a získání odkazu při zadání jména podporují metody třídy *Naming*. Třída má signaturu:

```
public final class Naming {
   public static Remote lookup(String name) throws NotBoundException,
            MalformedURLException, UnknownHostException, RemoteException;
   public static void bind(String name,Remote obj) throws AlreadyBoundException,
            MalformedURLException, UnknownHostException, RemoteException;
   public static void rebind(String name,Remote obj) throws
            MalformedURLException, UnknownHostException, RemoteException;
   public static void unbind(String name) throws NotBoundException,
            MalformedURLException, UnknownHostException, RemoteException;
   public static String[] list(String name) throws
            MalformedURLException, UnknownHostException, RemoteException;
}   .
```

Použití metod třídy Naming si můžeme ilustrovat na jednoduchých ukázkách. O registraci pod zadaným jménem (*"xyz"*) žádá vzdálený objekt *remObj* voláním metody *bind* (nebo *rebind*, pokud chceme předefinovat předchozí vazbu jména na jiný vzdálený objekt):

```
Naming.rebind("xyz",remObj);   .
```

Vyhledání tohoto serveru (budeme předpokládat, že k jeho registraci došlo na počítači *java*) klientem pak může mít formu:

```
BankAccount remObj = (BankAccount)Naming.lookup("//java/xyz");   .
```

Systém RMI vyžaduje použití vhodné bezpečnostní strategie, která nám zaručí, že objekt (například klientský applet načtený ze sítě prohlížečem) bude mít přístup k systémovým prostředkům (lokálním souborům, komunikačním kanálům) vhodně omezený. Konkrétní bezpečnostní strategii pro RMI aplikace volíme výběrem objektu třídy *RMISecurityManager* statickou metodou *System.setSecurityManager*.

```
System.setSecurityManager(new RMISecurityManager());   .
```

V systému zahrnutá třída *java.rmi.RMISecurityManager* (je odvozená ze základní třídy *java.lang.SecurityManager*) povoluje jen nejnutnější funkce potřebné pro přenos RMI odkazů, serializovaných reprezentací objektů a pro vlastní RMI volání. Implicitně použitý *java.lang.SecurityManager* dovoluje přístup jen k lokálním souborům. Rozšíření povolených funkcí oproti strategii *RMISecurityManager* lze dosáhnout definováním vlastní odvozené bezpečnostní strategie.

A.2 Vzdálené objekty

Třídy definující vzdálené objekty odvozujeme ze tříd *java.rmi.server.RemoteObject*, *java.rmi.server.RemoteServer* a *java.rmi.server.UnicastRemoteObject*. Termínem *vzdálený objekt* označujeme takto získaný objekt, bez ohledu na jeho skutečné umístění.

Třída *RemoteObject* modifikuje základní metody třídy *Object* pro vzdáleně přístupné objekty. Má signaturu

```
package java.rmi.server;
public abstract class RemoteObject
      implements java.rmi.Remote, java.io.Serializable {
   public int HashCode();
   public boolean equals(Object obj);
   public String toString();
}  .
```

Třída *RemoteServer* odvozená ze třídy *RemoteObject* vytváří základ pro implementaci metod vzdálených objektů. Signatura zahrnuje několik služebních metod:

```
package java.rmi.server;
public class RemoteServer extends RemoteObject {
   public static String getClientHost() throws ServerNotActiveException;
   public static void setLog(java.io.OutputStream out);
   public static java.io.PrintStream getLog();
}
```

Metoda *getClientHost* dovoluje zjistit počítač klienta, který vyvolal právě prováděnou metodu; výjimka *ServerNotActiveException* indikuje situaci, kdy akce nebyla vyvolána vzdáleným voláním. Metoda *setLog* dovoluje zvolit výstupní kanál pro monitorování (parametrem *null* lze záznam vypnout), metoda *getLog* předává tento kanál ke zpracování.

Třída *RemoteServer* slouží jako základ, nad nímž lze vystavět mechanismus umožňující specifickou formu komunikace se vzdáleným objektem. Lze tak podpořit práci s replikovanými objekty, s objekty persistentními a podobně. V současnosti (verze 1.1.4) je běžně podporována pouze synchronní komunikace s jednoduchým objektem třídou *UnicastRemoteServer*. Ta má signaturu:

```
package java.rmi.server;
public class UnicastRemoteServer extends RemoteServer {
   protected UnicastRemoteObject() throws java.rmi.RemoteException;
   public Object clone() throws java.lang.CloneNotSupportedException;
   public static void exportObject(java.rmi.Remote obj)
         throws java.rmi.RemoteException;
}
```

Konstruktor *UnicastRemoteServer* vytváří vzdálený objekt. Výsledný objekt je zpřístupněn (*exported*) a může být použit jako parametr nebo výsledek vzdáleného volání. Metoda *clone* vytváří kopii vzdáleného objektu (třída *RemoteObject* totiž neimplementuje rozhraní *java.lang.Clonable*). Metoda *exportObject* zpřístupňuje objekt, který byl vytvořen jinak než konstruktorem *UnicastRemoteServer*. Použití nezpřístupněného objektu pro vzdálené volání vyvolá výjimku *java.rmi.server.StubNotFoundException*.

Jako příklad si uvedeme definici vzdáleného objektu BankAccountImpl implementujícího naše rozhraní bankovního účtu BankAccount:

```
package bankaccount;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class BankAccountImpl extends UnicastRemoteObject implements BankAccount {
   public void deposit(float amount) throws java.rmi.RemoteException {
   }
   public void withdraw(float amount)
         throws OverDrawnException, java.rmi.RemoteException {
   }
   public float balance() throws java.rmi.RemoteException {
   }
}
```

Třídy, které jsme si zatím uvedli vytváří kostru RMI aplikací a u jednoduchých aplikací s nimi bez problémů vystačíme. Chceme-li podrobně porozumět činnosti mechanismů skrývajících se za kódem generovaným překladačem *rmic* neobejdeme se bez pochopení některých podpůrných rozhraní a systémových tříd. Těm jsou věnovány následující odstavce.

Instance tříd implementující rozhraní *RemoteRef* odkazují na vzdálené objekty (na straně klienta). Protějšky těchto odkazů na straně vzdálených objektů jsou popsány třídami implementujícími rozhraní *ServerRef*. Jednotlivá volání na straně klienta zajišťují instance třídy implementující rozhraní *RemoteCall*, na straně serveru se o volání metod stará rozhraní *Skeleton*. Třída *RemoteStub* je kostrou všech klientských stubů.

Rozhraní *LoaderHandler* podporuje vzdálené dynamické přisestavování tříd, rozhraní RMIFailureHandler dovoluje informovat o problémech při přípravě TCP kanálu (přípravě socketů tříd Socket a ServerSocket). Konečně, rozhraní *java.rmi.server.Unreferenced* indikuje, že na objekt již neexistuje žádný vzdálený odkaz. Jeho metoda *unreferenced* je aktivována při zrušení posledního ze vzdálených odkazů (to může být za dobu existence vzdáleného objektu i vícekrát).

Třída *RMISocketFactory* vytváří sockety pro RMI spojení. Funkce jejích metod *createSocket* a *createServerSocket* je poměrně zajímavá: pokouší se otevřít běžný TCP kanál, pokud neuspěje (např. pokud navázání TCP spojení nedovolí bezpečnostní brána (firewall) zkouší spojení HTTP protokolem po vlastním kanále, a pokud ani zde neuspěje, vytváří spojení přes implicitní kanál HTTP (TCP:80) a využívá funkce POST tohoto protokolu. Třída *RMIClassLoader* se stará o vzdálené dynamické přisestavování tříd. Třídy *UID* a *ObjID* vytvářejí jedinečné identifikátory v rámci jednoho virtuálního stroje. Třída *Operation* uchovává informace o metodách RMI rozhraní. Na závěr, třída *LogStream* podporuje monitorování chyb.

A.3 Příklad - jednoduchá aplikace klient-server

Jako reálný příklad použití technologie RMI si uvedeme jednoduchou aplikaci tvořenou klientem, který umí požádat o zopakování textového řetězce doplněného o pozdravení ("Hello"), a vzdálený objekt, který na požádání zaslaný řetězec doplněný o pozdravení ("Hello") vrátí.

Prvním krokem při programování aplikace RMI je definice RMI rozhraní, v našem případě toto rozhraní zahrnuje jedinou metodu, která vrací textový řetězec.

```
public interface Hello extends java.rmi.Remote {
   String sayHello(String s) throws java.rmi.RemoteException;
}
```

Server, který rozhraní *Hello* implementuje a je typicky rozšířením některé z podtříd *java.rmi.server.RemoteObject*, musí kromě konstruktoru a vlastního kódu metody *sayHello* definovat bezpečnostní strategii a svou službu zaregistrovat.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloImpl extends UnicastRemoteObject implements Hello {
   private String greeting = "Hello";
   public HelloImpl(String s) throws RemoteException {
      super();
      greeting = s;
   }
   public String sayHello(String s) throws RemoteException {
      return greeting+" "+s+"!";
   }
   public static void main(String args[]) {
      try {
         System.setSecurityManager(new RMISecurityManager());
      }
      catch(Exception e) {
         System.out.println(e); return;
      }
      try {
         HelloImpl obj = new HelloImpl("Salut");
         Naming.rebind("//localhost/HelloServer",obj);
         System.out.println("HelloServer bound in registry");
      }
      catch(Exception e) {
         System.out.println(e);
      }
   }
}
```

Explicitní konstruktor třídy *RemoteObject* (dovoluje zaměnit formu pozdravení) je uveden spíše pro ilustraci (implicitní konstruktor je aktivován automaticky). Výjimku *java.rmi.RemoteException* konstruktoru může vyvolat chyba v komunikačním systému indikovaná při instalaci vzdáleného objektu.

Prvním krokem statické metody *main* třídy definující vzdálený objekt/objekty je instalace objektu *RMISecurityManager* (případně jiného objektu odvozeného ze třídy *SecurityManager*, ale spíše ze třídy *RMISecurityManager*). Bez něj nelze zavést RMI objekty (stuby, skeletony) ani lokálně, *RMISecurityManager* dovoluje vzdálené zavádění stubů pro parametry a výsledek. Teprve potom lze instalovat jeden nebo více vzdálených objektů.

Navíc, pokud pracujeme s novější verzí Java SDK (počínaje 1.2), musíme komponentám distribuovaného programu poskytnout soubor definující omezení kladená na použité komunikační kanály. Příkladem takového souboru je následující soubor *java.policy*:

```
grant
      permission java.net.SocketPermission "*:1024-65535","connect,accept";
      permission java.net.SocketPermission "*:80","connect";
;
```

Aby bylo možné vzdálený objekt zpřístupnit klientovi, musíme mu umět poskytnout odkaz na tento objekt. Takovou službu poskytuje aplikace *rmiregistry*, metoda *Naming.rebind* v našem příkladě zpřístupňuje stub objektu *HelloImpl* pod jménem *HelloServer* na počítači *localhost*. Formát jména ve volání se řídí pravidly pro vytváření URL odkazů s tím, že nemusíme definovat protokol (tedy *rmi*), jméno počítače (jde-li o *localhost*) a můžeme využít implicitní port 1099, na němž je služba *rmiregistry* běžně dostupná (plný formát URL odkazu by pro náš příklad byl *rmi://localhost:1099/HelloServer*).

Klient, který metodu *sayHello* volá musí nejprve zjistit odkaz na vzdálený objekt třídy *HelloImpl*, v našem příkladě vytvořením URL odkazu a jeho použitím jako parametru metody Naming.lookup.

```
import java.rmi.*;
public class HelloClient {
   static String message = "";
   public static void main(String[] args) {
      try {
         Hello obj = (Hello)Naming.lookup("//localhost/HelloServer");
         message = obj.sayHello("client");
      }
      catch(Exception e) {
         System.out.println(e);
      }
      System.out.println(message);
   }
}
```

Po získání odkazu na stub vzdáleného serveru (v proměnné *obj*) už můžeme používat metody vzdáleného objektu běžným způsobem, pouze se musíme postarat o zpracování případných výjimek vyvolaných problémy při komunikaci.

Pro doplnění našeho přehledu si ještě uvedeme klienta, který má formu appletu zaváděného HTTP protokolem ze serveru WWW.

```
import java.awt.*;
import java.rmi.*;
public class HelloApplet extends java.applet.Applet {
   String message = "";
   public void init() {
      try {
         Hello obj = (Hello)Naming.lookup("//"+getCodeBase().getHost()
                           +"/HelloServer");
         message = obj.sayHello("client");
      }
      catch(Exception e) {
         System.out.println(e);
      }
   }
   public void paint(Graphics g) {
      g.drawString(message,25,50);
   }
```

Spíše jako poznámku si uveďme, že při vyhledávání odkazu na stub je zde nutné uvést jméno počítače, z něhož byl zaveden applet, protože standardní AppletSecurityManager nedovolí přístup k lokálním prostředkům.

Příslušná WWW stránka, která dovolí náš applet zavést a spustit může mít například tvar:

```
<HTML>
<title>Hello World</title>
<center><h1>Hello World</h1></center>
The message from the HelloServer is:
<p>
<applet codebase="../.."
   code="examples.hello.HelloApplet"
   width=500 height=120>
</applet>
</HTML>
```

Je zřejmé, že na počítači, na kterém běží server naší aplikace musí být v tomto případě funkční HTTP server, který poskytne uvedenou WWW stránku.

Pokud jde o vytvoření aplikace, prvním krokem je překlad potřebných zdrojových textů. V našem případě, kdy jsou zdrojové soubory *Hello.java*, *HelloImpl.java* a *HelloClient.java* uloženy v podadresáři *Hello* aktuálního adresáře (nebo workspace Eclipse), bude mít volání překladače tvar:

```
javac Hello.java HelloImpl.java HelloClient.java
```

Uvedený příkaz předpokládá zpřístupnění překladače *javac* v systémové proměnné *PATH* (to platí i pro další binární soubory jako jsou *java*, *rmic* a *rmiregistry*). Překládané soubory máme v podadresáři *hello* adresáře aktuálního. Knihovny překladače musí být zpřístupněné systémovou proměnnou *CLASSPATH*, případně lze cestu k nim zadat explicitně parametrem *-classpath*. Pokud chceme přeložené soubory aplikace ukládat do jiného adresáře, než kde jsou soubory zdrojové, musíme takový adresář specifikovat parametrem *-d*.

Po překladu musíme vygenerovat soubory *HelloImpl_Stub.class* a *HelloImpl_Skel.class* (pro verzi RMI 1.1). Příslušný příkaz pro překladač *rmic* může mít tvar:

```
rmic -d . -keep HelloImpl
```

Parametrem *-d* opět můžeme definovat adresář, kam budou uloženy generované soubory, parametrem *-keepgenerated* žádáme i o vytvoření jejich textové formy (*HelloImpl_Stub.java* a *HelloImpl_Skel.java*). Textovou formu vygenerovaného stubu a skeletonu se na závěr našeho příkladu uvedeme.

V dalším kroku se musíme případně postarat o uložení vytvořených *bytekódů* - souborů *.class* do adresářů na počítači klienta i serveru (pokud je tam neuložil přímo překladač *javac* a generátor *rmic*). Před spuštěním serveru aplikace musí na tomto počítači běžet aplikace *rmiregistry*, její instalaci, např. na portu 2001, lze zajistit příkazem

```
rmiregistry 2001 &
```

Při neuvedení čísla portu použije *registry* implicitní port *1099*. Potom již pouze stačí spustit server příkazem:

```
java -Djava.security.policy=java.policy HelloImpl
```

a klienta příkazem:

```
java -Djava.security.policy=java.policy HelloClient
```