Tworzenia aplikacji rozproszonej RMI

Mówiąc o aplikacjach RMI wyróżnia się w nich dwa typy obiektów:

- obiekty wywołujące metody zdalne (klienci)
- obiekty udostępniające metody zdalne (serwery)

RMI dostarcza mechanizmów, dzięki którym serwer i klient mogą komunikować się, wymieniając informacje w obie strony. Architektura aplikacji bazująca na obiektach klienta i serwera nazywana jest architekturą obiektów rozproszonych.

Lokalizacja obiektów zdalnych

Istnieją dwa mechanizmy, dzięki którym aplikacja może zlokalizować obiekty zdalne, tj. uzyskać do nich referencję:

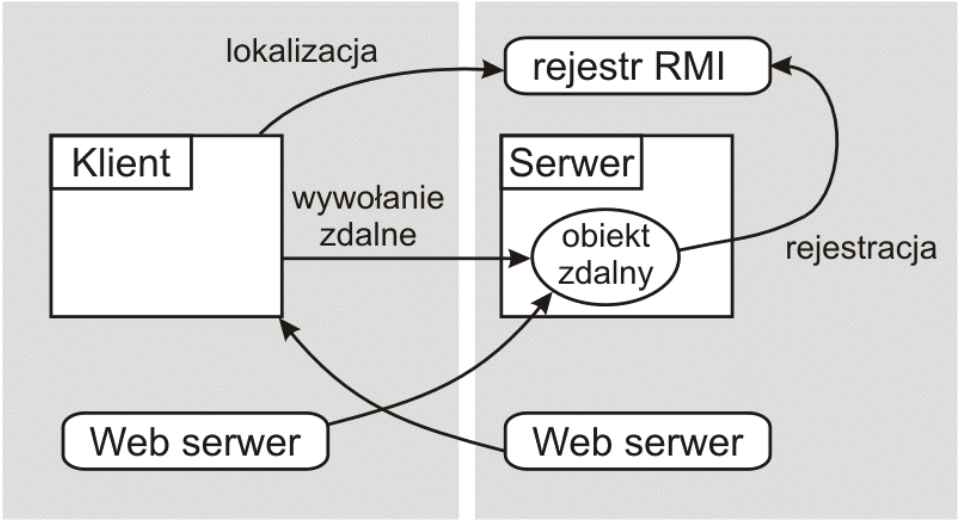
- aplikacja może zarejestrować swoje obiekty z metodami wykonywanymi zdalnie w serwisie nazw, np. w rmiregistry
- aplikacja może zwrócić i przekazać referencję do obiektu zdalnego w trybie normalnej swojej pracy

Komunikacja z obiektami zdalnymi

Szczegóły komunikacji z obiektami zdalnymi ukryte są przed użytkownikiem. Wszystkie niezbędne metody i operacje dostarcza RMI. Programista korzysta z metod obiektów zdalnych tak, jakby były to wywołania metod obiektów lokalnych.

Ładowanie kodu bajtowego klas obiektów przekazywanych zdalnie

RMI pozwala wywoływać metody zdalne, którym można przekazać obiekty jako parametry. Jest to możliwe, gdyż RMI dostarcza mechanizmu ładowania kodu bajtowego obiektów, jak również przesyłanie ich poprzez sieć.



Zalety dynamicznego ładowania kodu

RMI pozwala na ładowanie (ściąganie) kodów bajtowych klas danego obiektu, jeśli klasa ta nie jest zdefiniowana na wirtualnej maszynie odbiorcy.

Typy oraz własności obiektu, wcześniej dostępnego tylko na pojedynczej wirtualnej maszynie, mogą być transmitowane do innej, być może zdalnej, wirtualnej maszyny. RMI przesyła obiekty z ich prawdziwym typem, stąd własności obiektu nie ulegają zmianie podczas przesyłania. Pozwala to na wprowadzanie nowych typów na zdalną maszynę wirtualną, dynamicznie rozszerzając własności aplikacji.

Zdalne interfejsy, obiekty i metody

Podobnie do innych rozwiązań, rozproszone aplikacje bazujące na RMI używają klas i interfejsów. Interfejsy definiują metody, zaś klasy implementują metody tych interfejsów (obok innych metod, którymi dysponują).
W aplikacjach rozproszonych korzysta się z metod obiektów znajdujących się na różnych maszynach wirtualnych. Takie obiekty nazywane są wtedy obiektami zdalnymi. Obiekt staje się obiektem zdalnym, kiedy implementuje zdalny interfejs.

Interfejs zdalny charakteryzujący się tym, że:

- rozszerza interfejs java.rmi.Remote.
- każda metoda interfejsu zgłasza wyjątek java.rmi.RemoteException (obok własnych wyjątków)

RMI traktuje obiekty zdalne inaczej niż inne obiekty, gdy przekazywane są one z jednej maszyny wirtualnej na drugą. Zamiast robić kopię po stronie odbiorcy, RMI przekazuje zdalną namiastkę zdalnego obiektu. Namiastka pracuje jako lokalny przedstawiciel, lub pełnomocnik (proxy) obiektu zdalnego, i jest dla użytkownika (wywołującego metody zdalnego obiektu) zdalną referencją. Użytkownik wywołuje właściwie metodę namiastki, która jest odpowiedzialna za przekazanie tego wywołania do obiektu zdalnego.

Namiastka obiektu zdalnego implementuje te same metody interfejsu zdalnego, co sam obiekt zdalny. Dlatego pełnić rolę interfejsu, którym dysponuje zdalny obiekt. Jednak namiastka udostępnia tylko metody interfejsu zdalnego, które są dostępne na zdalnej maszynie wirtualnej.

Tworzenie rozproszonych aplikacji w RMI

Podstawowe kroki przy tworzeniu aplikacji rozproszonej:

- projektowanie i implementacja komponentów aplikacji rozproszonej
- kompilacja źródeł i generacja namiastek
- udostępnienie klas w sieci
- uruchomienie aplikacji

Projektowanie i implementacja komponentów aplikacji rozproszonej

Polega na określeniu architektury aplikacji i zdefiniowaniu obiektów, które powinny być zdalnie dostępne. Na krok ten składa się:

- definicja zdalnych interfejsów, zawierających metody do zdalnego wywoływania przez klientów (programy klientów będą korzystać z tych interfejsów, a nie z ich implementacji). Część z tej definicji obejmuje określenie wszystkich obiektów lokalnych, które będą użyte jako parametry w wywołaniach metod oraz określenie wartości zwracanych. Jeśli klasy parametrów nie są jeszcze zaimplementowane, powinno dostarczyć się ich definicje.

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

public interface MyInterface extends Remote {
    int metoda (int arg) throws RemoteException;
}
```

- implementacja zdalnych obiektów (obiekty zdalne muszą implementować jeden lub więcej zdalnych interfejsów. Klasy takich obiektów mogą zawierać implementacje innych interfejsów (lokalnych lub zdalnych) oraz inne metody (dostępnie lokalnie). Jeśli klasy lokalne mają być użyte jako parametry lub wartości zwracane tych metod, to muszą one być również zaimplementowane.

```
import java.rmi.*;
import java.rmi.server.*;
import compute.*;

public class MyInterfaceImpl extends UnicastRemoteObject implements MyInterface {

public MyInterfaceImpl () throws RemoteException {
     super();
}
public int metoda (int arg) {
     return arg*10;
}
```

Uwaga:
Jeśli obiekt klasy UnicastRemoteObject jest używany w rozproszonej aplikacji, to powinien on: istnieć po stronie serwera, być aktywnym, być dostępnym przez protokół TCP/IP. Wszystkie te wymagania załatwia konstruktor klasy UnicastRemoteObject.
Gdy obiekt zdalny jest instancją klasy, która po UnicastRemoteObject nie dziedziczy, wtedy klasa obiektu zdalnego powinna implementować interfejs Remote, zaś sam obiekt powinien zostać wyeksportowany przez wywołanie metody:

```
UnicastRemoteObject.exportObject(obiekt, 0);
```

- implementacja aplikacji serwerów i klientów (klienci używający zdalnych obiektów mogą być zaimplementowani w dowolnej chwili po zdefiniowaniu zdalnych interfejsów oraz po utworzeniu obiektów zdalnych w aplikacjach serwerów)

MySerwer

```
public static void main(String[] args) {
  if (System.getSecurityManager() == null) {
     System.setSecurityManager(new RMISecurityManager());
  }
  String url = "//host:1099/";
  try {
       MyInterface engine = new MyInterfaceImpl();
       Naming.rebind(name+ "MyInterface", engine);
       System.out.println("MyInterfaceImpl ok");
  } catch (Exception e) {
         System.err.println("MyInterfaceImpl excep."+ e.getMessage());
    }
}
```

MyClient

```
import java.rmi.*;
public class MyClient {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null){
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "//"+args[0]+"/MyInterface";
            MyInterface clnt =  Naming.lookup(name);
            int i =  clnt.metoda(12);
            System.out.println(i);
        } catch (Exception e) {
            ...
        }
    }
}
```

Kompilacja źródeł i generacja namiastek

Jest to proces dwuetapowy.

- Na początek kompilowane są (javac) pliki źródłowe zawierające: interfejsy zdalne, ich implementacje, klasy serwerów i klasy klientów.
- Następnie uruchamiany jest kompilator rmic, aby utworzyć namiastki obiektów zdalnych. RMI wykorzystuje namiastki zdalnych obiektów jako pośredników w klientach.

kompilacja	wynik kompilacji
javac xxx.java	xxx.class
rmic xxx	xxx_Stub.class xxx_Skel.class

Namiastki tworzą na maszynie klienta blok informacji przesyłanych do serwera, składający się z:

- identyfikator obiektu zdalnego
- opis metody, która ma zostać wywołana
- szeregowane parametry

Odbiorca po stronie serwera dla każdego zdalnego wywołania metody

- rozszeregowuje parametry
- lokalizuje obiekt, którego metoda ma być wywołana
- wywołuje metodę i jej wynik (lub wyjątek) szereguje
- wysyła szeregowane dane jako informację zwrotną na stronę klienta

Przygotowanie wdrożenia

W kroku tym pliki klas związanych ze zdalnym interfejsem, namiastki oraz inne potrzebne klasy udostępniane są w sieci (np. przez Web serwer).

Strategia dla prostego przykładu:

serwer	download	klient
MyInterface.class MyInterfaceImpl.class MyInterfaceImpl_Stub.class MyServer.class	MyInterface.class MyInterfaceImpl_Stub.class MyInterfaceImpl_Skel.class (v1.1) inne, od których zależą namiastki, ładowane przez serwer	MyInterface.class MyClient.class client.policy

client.policy (definicja portów dla połączeń RMI oraz HTTP)

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

Uruchomienie aplikacji

Przy umieszczeniu serwera i klienta na jednym komputerze:

- uruchomieniu rejestru RMI na tej samej maszynie, co serwer. Podczas uruchamiania rejestru RMI w ścieżce klas nie powinno być widać żadnych klas należących do aplikacji

```
start rmiregistry
```

- uruchomieniu aplikacji serwera (zakładamy, że działa serwer http)

```
java –Djava.rmi.server.codebase=http://localhost/download/ MyServer
```

jeśli chcemy korzystać z lokalnego katalogu (nie przez serwer http):

```
java –Djava.rmi.server.codebase=file:/F:\RMItest/ -classpath "F:\RMItest" MyServer
```

- uruchomieniu aplikacji klienta.

```
java –Djava.security.policy=client.policy MyClient
```

Przy umieszczeniu serwera zdalnie można skorzystać z adresów URL plików. Konfiguracja i uruchomienia są wtedy następujące:

client.policy (dla połączeń RMI oraz HTTP)

```
grant
{
 permission java.net.SocketPermission "*:1024-65535", "connect"
 permission java.net.SocketPermission "*:80", "connect"
 permission java.io.FilePermission "downloadDirectory", "read"
}
```

gdzie zamiast * można umieścić nazwę komputera, na którym uruchomione jest rmiregistry (jak i MyServer). Może to być np. "jakishost.com" oraz:

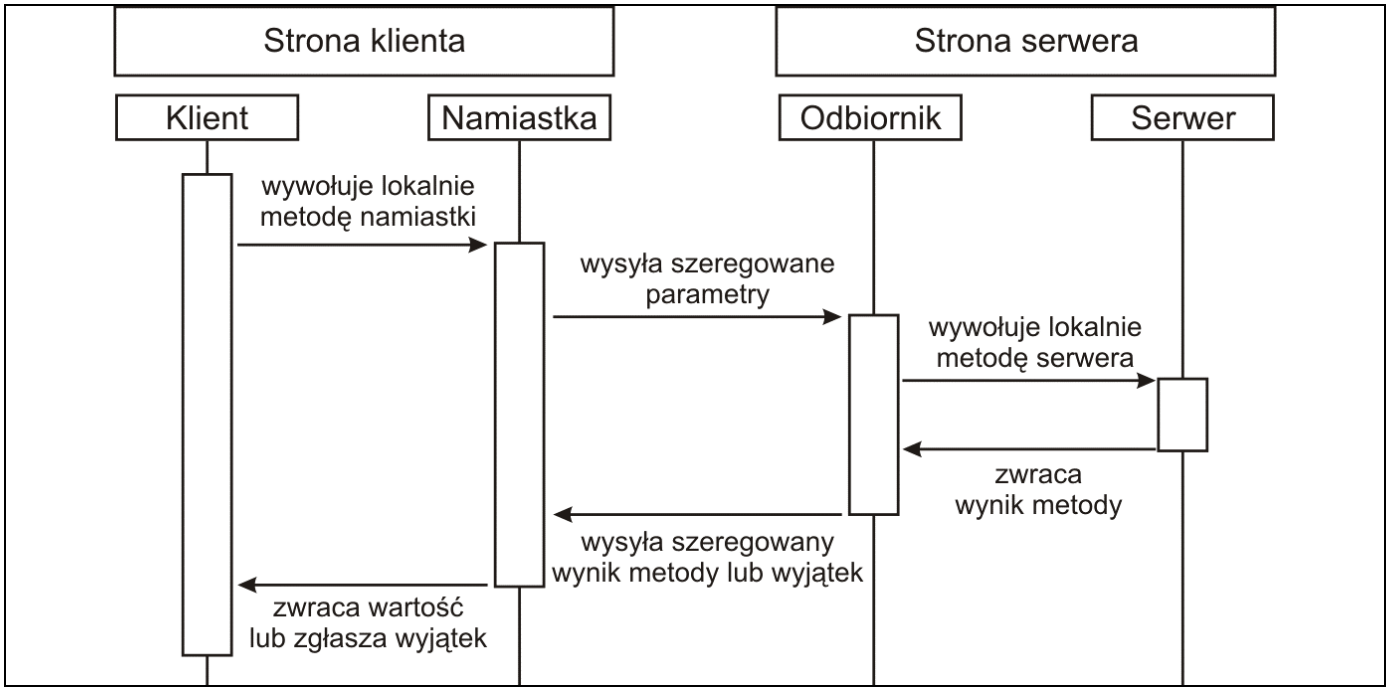
linux	windows
downloadDirectory="home/test/download/-"	*downloadDirectory*="c:\\home\\test\\download\\-"

- uruchomienie rejestru RMI
- uruchomienie serwera

```
java –Djava.rmi.server.codebase=file:///home/test/download/ MyServer&
lub
start java –Djava.rmi.server.codebase=file://c:\home\test\download/ MyServer

można też podać dodatkowe opcje:
    -Djava.rmi.server.hostname=jakishost.com
    -Djava.security.policy=java.policy
```

- uruchomienie klienta


Strona klienta
Strona serwera
Klient
Namiastka
Odbiornik
Serwer
wywołuje lokalnie
metodę namiastki
wysyła szeregowane
parametry
wywołuje lokalnie
metodę serwera
zwraca
wynik metody
wysyła szeregowany
wynik metody lub wyjątek
zwraca wartość
lub zgłasza wyjątek

java.rmi.registry

Interface Registry

All Superinterfaces:

Remote

```
public interface Registry
extends Remote
```

`Registry` is a remote interface to a simple remote object registry that provides methods for storing and retrieving remote object references bound with arbitrary string names. The `bind`, `unbind`, and `rebind` methods are used to alter the name bindings in the registry, and the `lookup` and `list` methods are used to query the current name bindings.

In its typical usage, a `Registry` enables RMI client bootstrapping: it provides a simple means for a client to obtain an initial reference to a remote object. Therefore, a registry's remote object implementation is typically exported with a well-known address, such as with a well-known `ObjID` and TCP port number (default is `1099`).

The `LocateRegistry` class provides a programmatic API for constructing a bootstrap reference to a `Registry` at a remote address (see the static `getRegistry` methods) and for creating and exporting a `Registry` in the current VM on a particular local address (see the static `createRegistry` methods).

A `Registry` implementation may choose to restrict access to some or all of its methods (for example, methods that mutate the registry's bindings may be restricted to calls originating from the local host). If a `Registry` method chooses to deny access for a given invocation, its implementation may throw `AccessException`, which (because it extends `RemoteException`) will be wrapped in a `ServerException` when caught by a remote client.

The names used for bindings in a `Registry` are pure strings, not parsed. A service which stores its remote reference in a `Registry` may wish to use a package name as a prefix in the name binding to reduce the likelihood of name collisions in the registry.

Since:

JDK1.1

See Also:

`LocateRegistry`

Field Summary

`static int`	**`REGISTRY_PORT`** Well known port for registry.

Method Summary

`void`	**`bind`**`(String name, Remote obj)` Binds a remote reference to the specified `name` in this registry.
`String[]`	**`list`**`()` Returns an array of the names bound in this registry.
`Remote`	**`lookup`**`(String name)`

	Returns the remote reference bound to the specified `name` in this registry.
`void`	**`rebind`**`(String name, Remote obj)` Replaces the binding for the specified `name` in this registry with the supplied remote reference.
`void`	**`unbind`**`(String name)` Removes the binding for the specified `name` in this registry.

java.rmi.registry

Class LocateRegistry

`java.lang.Object`
  └ **`java.rmi.registry.LocateRegistry`**

```
public final class LocateRegistry
extends Object
```

`LocateRegistry` is used to obtain a reference to a bootstrap remote object registry on a particular host (including the local host), or to create a remote object registry that accepts calls on a specific port.

Note that a `getRegistry` call does not actually make a connection to the remote host. It simply creates a local reference to the remote registry and will succeed even if no registry is running on the remote host. Therefore, a subsequent method invocation to a remote registry returned as a result of this method may fail.

Since:
JDK1.1

See Also:
`Registry`

Method Summary

`static Registry`	**`createRegistry`**`(int port)` Creates and exports a `Registry` instance on the local host that accepts requests on the specified `port`.
`static Registry`	**`createRegistry`**`(int port, RMIClientSocketFactory csf, RMIServerSocketFactory ssf)` Creates and exports a `Registry` instance on the local host that uses custom socket factories for communication with that instance.
`static Registry`	**`getRegistry`**`()` Returns a reference to the the remote object `Registry` for the local host on the default registry port of 1099.
`static Registry`	**`getRegistry`**`(int port)` Returns a reference to the the remote object `Registry` for the local host on the specified `port`.
`static Registry`	**`getRegistry`**`(String host)` Returns a reference to the remote object `Registry` on the specified `host` on the default registry port of 1099.
`static Registry`	**`getRegistry`**`(String host, int port)` Returns a reference to the remote object `Registry` on the specified `host` and `port`.

static Registry	**getRegistry**(String host, int port, RMIClientSocketFactory csf) Returns a locally created remote reference to the remote object Registry on the specified host and port.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait