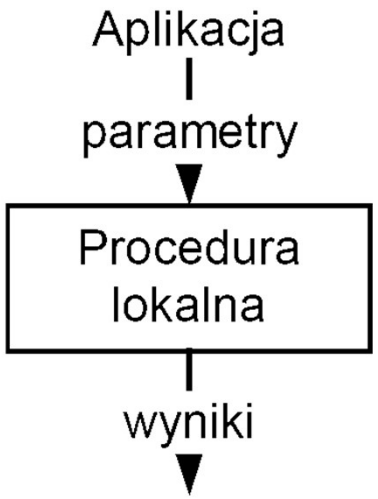
Zaawansowanie programowanie obiektowe

Wykład 7b
Zdalne wywoływanie procedur w środowisku Java.

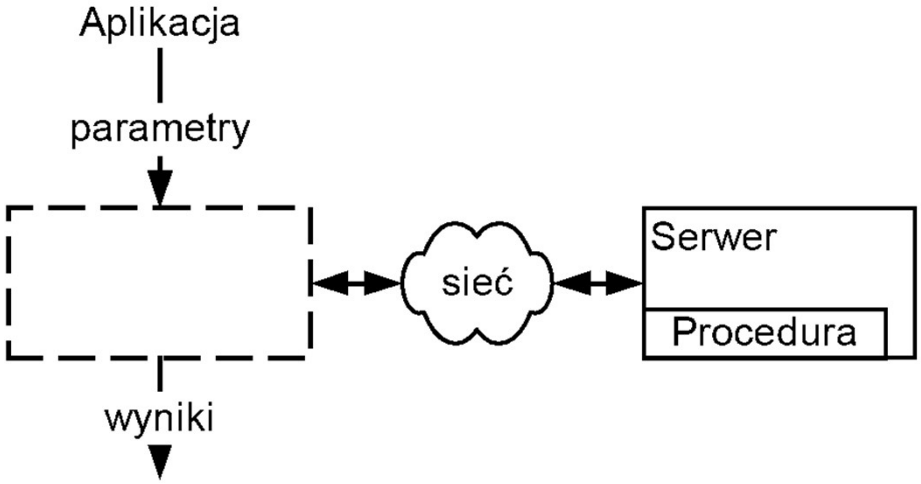
dr inż. Wojciech Bieniecki
mgr inż. Michał Paluch
wbieniec@kis.p.lodz.pl
http://wbieniec.kis.p.lodz.pl

Idea zdalnego wywoływania procedur

zdalne wywołanie procedur – ang Remote Procedure Call (RPC)



Lokalne wywołanie procedury



Paradygmat wywoływania zdalnych procedur
Birel i Nelson (1984)

Model klient serwer pozwala na rozwiązanie szerokiej klasy problemów posiada jednak ograniczenia:

- Odwołanie się do wejścia / wyjścia (receive / send)
- Występuje problem reprezentacji danych (systemy heterogeniczne)
- Model zorientowany na dane

Idea RPC

Możliwe problemy:

- Procedura wywołująca i wywoływana działają na różnych maszynach w różnych przestrzeniach adresowych.
- Problem z użyciem wskaźników.
- Maszyny mogą mieć różne systemy reprezentacji danych
- Występuje problem obsługi sytuacji awaryjnych

Łącznikami aplikacji klienta i serwera są:

- Stopka klienta (pieniek) (*ang. client stub)* – reprezentuje serwer po stronie klienta
- Stopka serwera (szkielet) (*ang. server stub*) – reprezentuje klienta po stronie serwera. Wywołanie procedury zostaje zastąpione wywołaniem stopki klienta.

Jak działa zdalne wywołanie procedury

1. Aplikacja klienta wywołuje stopkę klienta
2. Stopka klienta buduje komunikat i przechodzi do jądra OS
3. Jądro przesyła komunikat do jądra maszyny odległej
4. Stopka serwera rozpakowuje parametry i wywołuje serwer
5. Serwer wykonuje zdalną procedurę i zwraca wynik stopce serwera
6. Stopka pakuje wyniki w komunikat i przesyła do maszyny klienta
7. Jądro klienta przekazuje komunikat stopce klienta
8. Stopka klienta rozpakowuje wynik i zwraca go klientowi

Cechy mechanizmu RPC

Zalety:

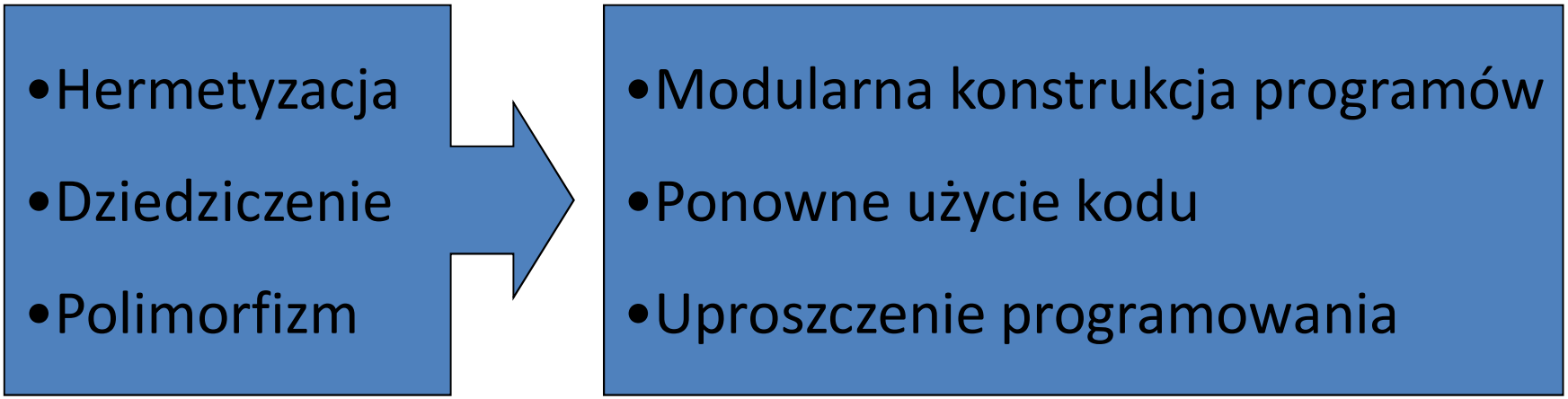
- Prostota
- Duża wydajność
- Rozpowszechnienie standardu

Wady

- Brak wsparcia serwerów RPC dla wielowątkowości
- Słabe techniki autoryzacji klienta
- Brak zabezpieczenia przed konfliktem numerów programów
- W interfejsach tylko funkcje jednoargumentowe
- Trudności w użyciu wskaźników do przekazywania parametrów
- Wsparcie tylko dla języka C

Obiektowy model przetwarzania

Cechy obiektowego modelu przetwarzania:



Obiektowy model przetwarzania

Motywacja:

Dostarczyć klientowi mechanizm obiektowego RPC

Obiektowe RPC + usługi = rozproszona platforma przetwarzania rozproszonego

Klient zna tylko interfejs obiektu, bez jego implementacji

Przykłady:

– CORBA (ang. *Common Object Request Broker Architecture*) – OMG
– DCOM (ang. *Distributed Component Object Model*) – Microsoft
– SOAP (*ang. Simple Object Access Protocol*) – Microsoft
– .NET Remoting – Microsoft
– Ninf, NetSolve/GridSolve (gridy)
– Java RMI (*ang. Remote Method Invocation*) – Sun Microsystems
– Globe – Uniwersytet Vrije

Java RMI

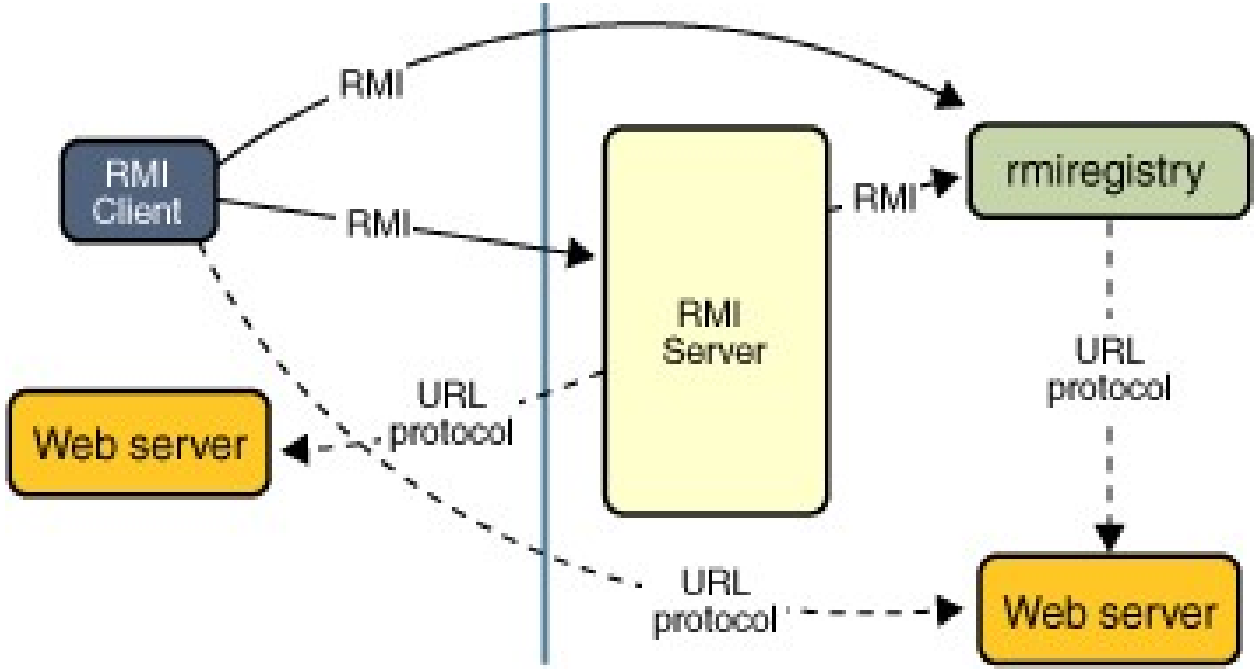
Remote Method Invocation – Zdalne wywoływanie metod

Obiektowe RPC dla Java

Mechanizm jest prosty w porównaniu do CORBA

Produkt firmowy Sun/Oracle

System RMI pozwala na to, aby obiekt działający na jednej maszynie wirtualnej wywoływał metody obiektu działającego na innej maszynie wirtualnej Javy



Tworzenie aplikacji RMI

1 Zdefiniuj interfejs zdalny

2 Implementacja interfejsu

(.java)

3 javac

(.class)

4 rmic

Pieniek klienta (.class)

Szkielet serwera (.class)

Klasa serwera (.class)

8 Implementacja klienta

uses

(.java)

9 javac

(.class)

10 Uruchom klienta

5 Uruchom rejestr RMI

6 Uruchom obiekty serwera

7 Rejestruj obiekty zdalne

KLIENT

SERWER

Przykład RMI: Tworzymy interfejs

Interfejs deklaruje metody (i ich argumenty), które będą zdefiniowane w klasie implementującej.

```
public interface myRMIInterface extends java.rmi.Remote
{
  public java.util.Date getDate() throws java.rmi.RemoteException;
}
```

Każdy interfejs zdalny musi spełniać dwa warunki:

Musi rozszerzać interfejs java.rmi.Remote przez dziedziczenie pośrednie lub bezpośrednie.

Wszystkie jego metody muszą deklarować klauzulą throws możliwość rzucenia wyjątku java.rmi.RemoteException.

Tworzymy klasę implementującą interfejs myRMIInterface

W klasie musimy przepisać nagłówki wszystkich metod zadeklarowanych w interfejsie i wypełnić ich ciała

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

public class myRMIImpl extends UnicastRemoteObject
implements myRMIInterface
{
   public java.util.Date getDate()
   {
      return new java.util.Date();
   }
// …
```

Obiekt zdalny

Aby metody obiektu mogły być zdalnie dostępne, musi on spełniać dwa warunki:

Implementować jakiś interfejs zdalny – to już mamy.

Być wyeksportowany

Eksport:

Dziedziczenie z java.rmi.server.RemoteServer,
(java.rmi.server.UnicastRemoteObject)

Użycie metody
static RemoteStub exportObject(Remote obj)

Eksport można wykonać w konstruktorze obiektu

Cofnięcie eksportu

static boolean unexportObject(Remote obj, boolean force)

Tworzymy klasę implementującą interfejs myRMIInterface – c.d.

```
//. . .
  public myRMIImpl(String name) // konstruktor
  throws RemoteException
  {
     super();
     try
     {
       Naming.rebind(name, this);
     }
     catch(Exception e)
     {
       System.out.println("Exception occurred: " + e);
     }
  }
}
```

Tworzenie serwera

W tym przykładzie serwer będzie miał dwa zadania

Zainstalowanie managera bezpieczeństwa RMI (RMISecurityManager)

Utworzenie instancji obiektu zdalnego myRMIImpl

Obiekt zdalny zostanie zainstalowany w rejestrze RMI i będzie dostępny pod nazwą myRMIImplInstance.

Klient będzie mógł uzyskać dostęp do obiektu poprzez adres URL: rmi://<adres IP serwera>/myRMIImpl

Tworzenie serwera

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

public class myRMIServer
{
  public static void main(String[] argv)
  {
    System.setSecurityManager(new RMISecurityManager());
    try
    {
       myRMIImpl ob = new myRMIImpl("myRMIImplInstance");
    }
    catch (Exception e)
    {
       System.out.println("Exception occurred: " + e);
    }
  }
}
```

Tworzenie klienta

W tym przykładzie:
-klient połączy się z serwerem
-uzyska dostęp do instancji zdalnego obiektu
-Wywoła metodę zdalną obiektu zdalnego
-Odbierze wynik działania metody

Przed połączeniem się z serwerem (instrukcja naming.lookup())
klient powinien również zainstalować managera bezpieczeństwa
RMI

Tworzenie klienta

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.Date;
public class myRMIClient{
  public static void main(String[] argv){
    System.setSecurityManager(new RMISecurityManager());
    try{
      myRMIInterface myServerObject = (myRMIInterface)
      Naming.lookup("rmi://"+argv[0]+"/myRMIImplInstance");
//invoke method on server object
      Date d = myServerObject.getDate();
      System.out.println("Date on server is " + d);
    }
    catch(Exception e){
      System.out.println("Exception occured: " + e);
      System.exit(0);
    }
    System.out.println("RMI connection successful");
  }
}
```

RMI i zarządzanie pamięcią

Główny problem – rozproszone zwalnianie pamięci (*distributed garbage collection*).

Rozwiązanie – licznik referencji dla każdego obiektu zdalnego

Protokół RMI zapewnia odpowiednią jego aktualizację. Obiekty zdalne mogą być zniszczone dopiero, kiedy nie ma do nich ani zdalnych, ani lokalnych referencji.

Aby serwer np. kończył sam pracę, kiedy nie jest już potrzebny, przydatna może być informacja o zniknięciu ostatniej zdalnej referencji. W takiej sytuacji serwer powinien implementować interfejs Unreferenced, zawierający tylko jedną, bez parametrową metodę unreferenced(), którą można wypełnić wedle uznania, a która jest wywoływana przez RMI w momencie stwierdzenia opróżnienia listy referencji do danego obiektu.

```
class Server … implements Unreferenced

…

void unreferened()

{

}
```

Serializacja obiektów

Metody zdalne mogą przekazywać przez sieć obiekty, np.

```
public Obiekt2 metoda(Obiekt1 obiekt)
```

RMI wykorzystuje w tym celu mechanizm serializacji.
Każdy obiekt, który ma być przekazywany przez RMI powinien implementować (implements) interfejs java.io.Serializable

Niestandardowa serializacja wymaga zdefiniowania metod w tym obiekcie:

```
private void writeObject(java.io.ObjectOutputStream out)
        throws IOException;

private void readObject(java.io.ObjectInputStream in)
        throws IOException, ClassNotFoundException;
```

Rejestracja serwerów
Fabryki obiektów

rmiregistry – uruchamia rejestr serwerów na maszynie

Identyfikacja serwisu:

```
[rmi:]//<komputer>[:<port>]/<nazwa usługi>
```

Funkcje używane do rejestracji serwera

```
public static void bind(String name, Remote obj)
public static void unbind(String name)
public static void rebind(String name, Remote obj)
public static String[] list(String name)
public static Remote lookup(String name)
```

mechanizm fabryki: rejestrowany jest jeden obiekt, którego metody tworzą na żądanie odpowiednie obiekty - serwery różnych usług i zwracają referencje do nich

Przykład z użyciem fabryki – interfejsy

Głównym obiektem zdalnym będzie Factory.
Obiekt ten będzie uruchamiał i odłączał obiety Server i Client

```
1  //interfaces.java
2  interface Factory extends java.rmi.Remote
3  {
4    Server get() throws java.rmi.RemoteException;
5  }
6
7  interface Server extends java.rmi.Remote
8  {
9    void hello(Client me) throws java.rmi.RemoteException;
10   void forget() throws java.rmi.RemoteException;
11   String say() throws java.rmi.RemoteException;
12 }
13 interface Client extends java.rmi.Remote
14 {
15   String what() throws java.rmi.RemoteException;
16 }
17
18
```

Implementacja – fabryka

theFactory pełni tu rolę serwera. Rejestruje obiekt zdalny Fabryka.
Obiekt ten ma funkcję get, która na żądanie utworzy drugi obiekt zdalny – theServer

```
1  //implementations.java
2  import java.lang.*;
3  class theFactory implements Factory
4  {
5    theFactory() throws java.rmi.RemoteException
6    {
7      java.rmi.server.UnicastRemoteObject.exportObject(this);
8    }
9    public Server get() throws java.rmi.RemoteException
10   {
11     return new theServer();
12   }
13
14
15   public static void main(String args[])
16   {
17     try {
18       theFactory maker = new theFactory();
19       java.rmi.Naming.rebind("//127.0.0.1/Fabryka",maker);
20       System.out.println("Fabryka zarejestrowana");
21     }
22     catch (Exception e) {
23       System.out.println("Nie zarejestrowano fabryki! Wyjatek: "+ e.getMessage() );
24     }
25   }
26 }
```

Implementacja – theServer

```
28  class theServer implements Server
29  {
30    Client he;
31    theServer() throws java.rmi.RemoteException
32    {
33      he = null;
34      System.out.println("Nowy serwer!");
35      java.rmi.server.UnicastRemoteObject.exportObject(this);
36    }
37    public void hello(Client me) throws java.rmi.RemoteException
38    {
39      System.out.println("Rejestracja klienta");
40      he = me;
41    }
42    public void forget() throws java.rmi.RemoteException
43    {
44      System.out.println("Wyrejestrowanie klienta");
45      he = null;
46    }
47    public String say() throws java.rmi.RemoteException
48    {
49      try {
50        String text = he.what(); // Callback! To tak proste w RMI.
51  // Tak naprawdę he mogłoby byc argumentem!
52        System.out.println(text);
53        return "Witaj! " + text;
54      }
55      catch (Exception e) {
56        System.out.println("Blad komunikacji:" + e.getMessage());
57      }
58      return null;
59    }
60  }
```

Implementacja – theClient

```
62  class theClient implements Client
63  {
64    Server textmaster;
65    String tekst;
66    theClient() throws java.rmi.RemoteException
67    {
68      tekst = null;
69      textmaster = null;
70      java.rmi.server.UnicastRemoteObject.exportObject(this);
71    }
72    public String what() throws java.rmi.RemoteException
73    {
74      System.out.println("-- Pytanie! Wysylam tekst...");
75      return tekst;
76    }
77
```

Implementacja – theClient

```
78    public static void main(String args[])
79    {
80      try {
81        theClient me = new theClient();
82        me.tekst = "Test RMIkrofonu, 1, 2, 3...";
83        Factory dom = (Factory)java.rmi.Naming.lookup("//localhost/Fabryka");
84        System.out.println("Jest fabryka.");
85        me.textmaster = dom.get();
86        System.out.println("Jest serwer. Podanie referencji...");
87        me.textmaster.hello(me);
88        System.out.println("Praca...");
89        System.out.println( me.textmaster.say() );
90        System.out.println("Kasowanie referencji...");
91        me.textmaster.forget();
92        System.out.println("Koniec!");
93        me.textmaster = null;
94        me = null;
95        System.exit(0);
96      }
97      catch (Exception e) {
98        System.out.println("Niepowodzenie!: " + e.getMessage() );
99      }
100   }
101 }
```

Uruchomienie aplikacji

Kompilacja

```
javac *.java
rmic theFactory theClient theServer
```

Uruchomienie serwera (na jednej stacji

```
rmiregistry
java theFactory
```

Uruchomienie klienta (na drugiej stacji):

```
java theClient
```

Podsumowanie

RMI pozwala na szybkie i łatwe tworzenie aplikacji rozproszonych

RMI wykorzystuje wyłącznie język Java

Aby zdalnie wywołać metodę nie trzeba jej konwertować do interfejsów IDL.

W DCOM lub CORBA ta konwersja jest konieczna.

Ze względu na silny związek z Javą RMI jest w stanie współdziałać z dziedziczeniem. Nie jest to możliwe w DCOM ani CORBA, ponieważ te rozwiązania są zazwyczaj statyczne.

Parametry metody zdalnej i odbierana wartość mogą być zwykłymi obiektami Javy - niemożliwe w modelu DCOM i CORBA.
Java-RMI obsługuje Distributed Garbage Collection, który działa tak jak lokalny odśmiecacz na każdej z maszyn wirtualnych.

RMI działa wolniej niż RPC i nawet wolniej niż CORBA

Wywołania RMI są oczywiście blokujące, jak zwykle w RPC, jendak wbudowana wielowątkowość pozwala na uruchomienie innych funkcji programu w trakcie obliczeń na maszynie zdalnej