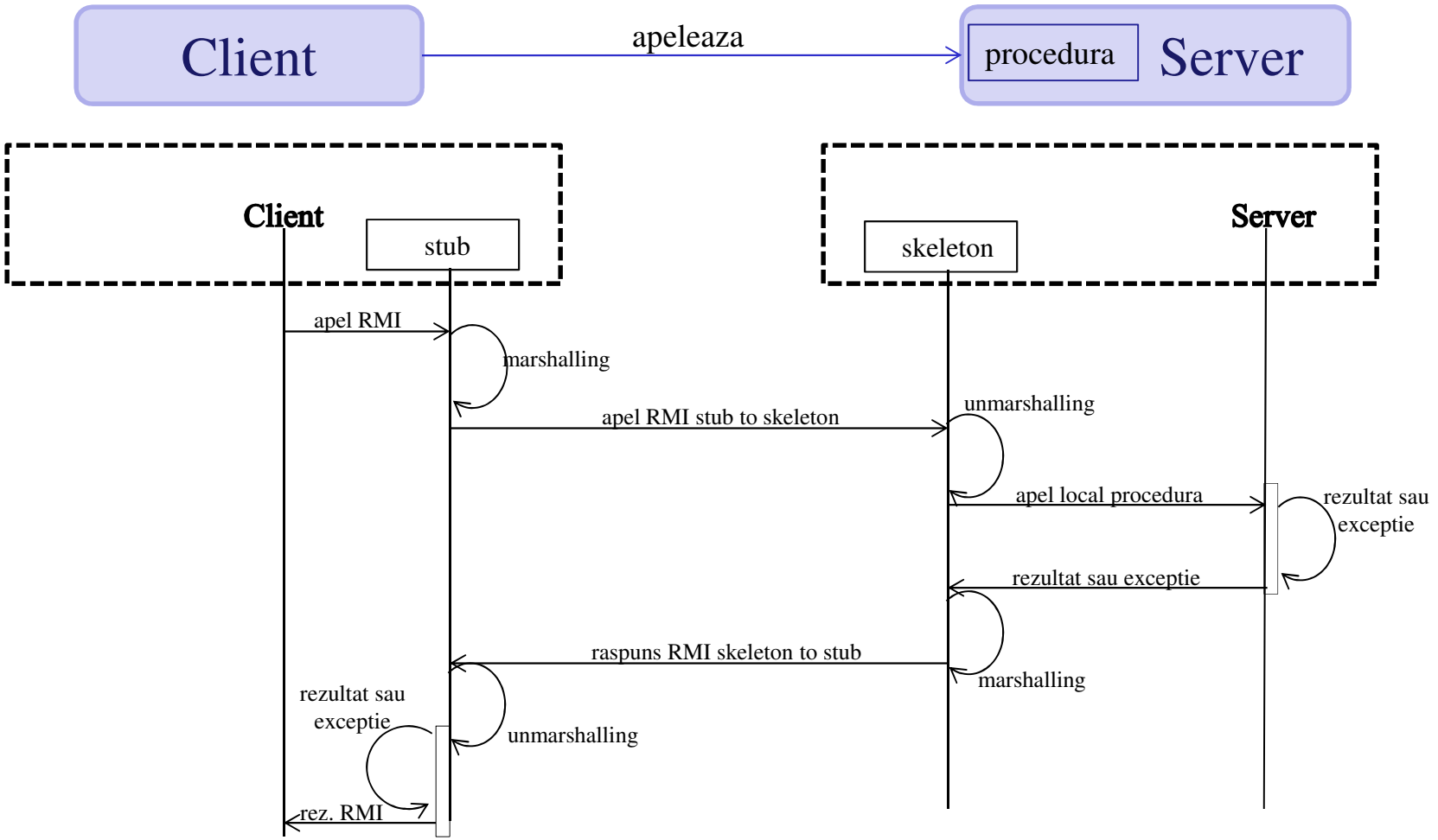
RMI: Remote Method Invocation

- RMI – remote method invocation – invocarea de metode la distanta - reprezinta o extindere a conceptului de RPC – Remote Procedure Call – utilizarea apelului de procedura – pentru comunicarea intre procese care se executa pe calculatoare diferite



RMI: Arhitectura si utilizare

- **Arhitectura RMI**: este structurata pe trei nivele:
  - Nivelul superior: stub si skeleton;
  - Nivelul median: referinte la distanta;
  - Nivelul de baza: transport.

- **Utilizarea RMI:** Un obiect la distanta (remote object, numit si server in continuare) pune la dispozitie metode pentru a fi apelate de pe alte masini virtuale;
- Obiectele accesibile la distanta sunt similare obiectelor obisnuite dar nu sunt identice;
- Utilizarea RMI presupune urmatorii pasi:
  - Definirea unei interfete prin care va fi apelat serverul (remote object);
  - Definirea unei clase care reprezinta implementarea interfetei (instantierea ei – este un obiect la distanta);
  - Pe baza celor doua (interfata si implementarea interfetei) se pot obtine stub-ul si skeleton-ul (utilizand un program numit **rmic**);
  - Obiectul la distanta trebuie sa fie inregistrat la un serviciu de nume cunoscut atat de server cat si de client. Obiectul gaseste serverul (primeste stub-ul corespunzator);
  - Clientul poate sa apeleze metodele obiectului la distanta utilizand stub-ul;

RMI: Exemplu de implementare

SimpleCat.java

```
import java.io.*;
public class SimpleCat implements Serializable {
    private static final long serialVersionUID = 101L;
    String name;
    SimpleCat(String name){
        this.name = name;
    }
    void Print(){
        System.out.println(name);
    }
    String getName() {
        return name;
    }
}
```

SimpleCat este o clasa obisnuita; dorim sa obtinem informatii despre un obiect obtinut prin instantierea acestei clase; informatia va fi numele returnat de getName()

Pentru a obtine informatiile de pe alta masina va trebui sa implementam un server.
Serverul folosit pentru obtinerea informatiilor de pe alta masina este CatServer. Acesta implementeaza o interfata CatServerInterface (care extinde java.rmi.Remote)

CatServerInterface.java

```
public interface CatServerInterface extends java.rmi.Remote {
    String getCatName() throws java.rmi.RemoteException;
}
```

CatServer.java

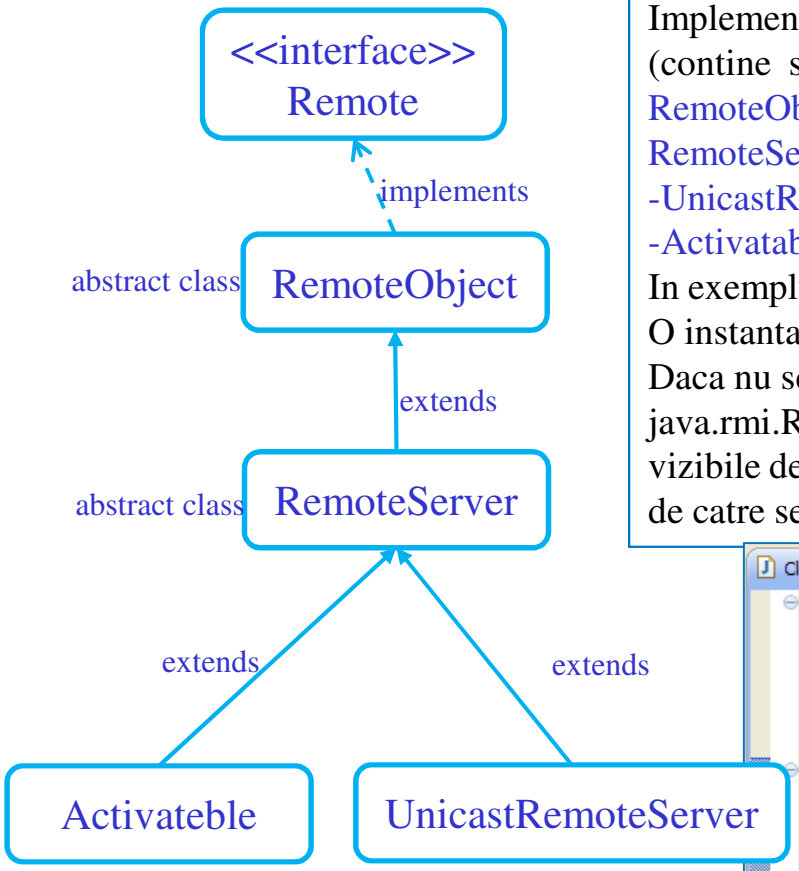
```
import java.rmi.*;

public class CatServer extends UnicastRemoteObject implements CatServerInterface {
    private static final long serialVersionUID = 102L;

    SimpleCat sc = null;
    CatServer(SimpleCat sc) throws java.rmi.RemoteException {
        super();
        this.sc =sc;
    }

    public String getCatName() throws RemoteException {
        return sc.getName();
    }
}
```

RMI: Exemplu de implementare



Implementarea interfetei java.rmi.Remote este o clasa abstracta, RemoteObject (contine si redefinirea metodelor hashCode(), equals(), toString() din Object. RemoteObject este la randul ei extinsa de clasa abstracta RemoteServer. RemoteServer este extinsa de doua clase:
-UnicastRemoteServer
-Activatable
In exemplul nostru, CatServer extinde UnicastRemoteServer
O instanta a lui CatServer este un obiect care poate accepta apeluri la distanta. Daca nu se reuseste construirea obiectului, va genera o exceptie de tip java.rmi.RemoteException. CatServer poate avea si metode locale dar singurele vizibile de la distanta sunt cele descrise in interfata de tip Remote implementata de catre server (si este cunoscuta de client).

Accesul la metoda getCatName() se face dintr-un obiect considerat client: ClientRMICat

ClientRMICat.java

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;

public class ClientRMICat {

    static public void main(String argv[]){
        System.setSecurityManager(new RMISecurityManager());
        String where = "rmi://localhost:5555/CatServer";

        try {
            Remote robj = Naming.lookup(where);
            System.out.println("Reference to server obtained " + robj.getClass());
            CatServerInterface csi = (CatServerInterface) robj;
            String s = (String)csi.getCatName();
            System.out.println("Cat name obtained: " + s);
        }
        catch (Exception e){
            System.out.println("Error" + e);
            System.exit(0);
        }
    }
}
```

RMI: Exemplu de implementare

Daca clientul si serverul sunt pe masini diferite se pune problema cum poate clientul sa obtina o referinta la server pe baza careia sa aiba acces la metodele acestuia.
Se utilizeaza un obiect numit registry, rolul acestuia fiind dublu: gestionar de obiecte si registru de nume.
Codul pentru registry se executa pe masina pe care se afla serverul. Clientii utilizeaza clasa Naming din implementarea RMI pentru a comunica cu acel obiect.
Metoda Naming.bind() face legatura intre un nume si un server (reprezentat de un obiect aflat la distanta). Clasa Naming utilizeaza notatia de tip URL pentru accesarea obiectelor aflate la distanta. Forma cea mai generala pentru accesarea unui obiect la distanta este: rmi://server:port/nume. Portul implicit este 1099. Clasa LocateRegistry implementeaza createRegistry() si getRegistry(), pentru a creea, respectiv a accesa registri.
Prin apelul metodei Naming.lookup() rezulta un obiect (stub) care va permite folosirea metodelor obiectului la distanta.
Apelul metodei la distanta este: csi.getCatName() unde csi este referinta obiectului aflat la distanta. Clientii trebuie sa cunoasca interfetele implementate de obiectele la distanta (implementarea propriu-zisa nu este importanta, executia facandu-se evident la distanta.

Legatura intre server si serviciul de nume se face, in exemplul nostru, prin programul CatServerStart. Acesta instantiaza un server si face apelul la serviciul de nume pentru a anunta noul server.
Daca exista un server, trebuie creeate clasele de tip stub si skeleton. Aceasta se face cu comanda (in cazul nostru):
>rmic CatServer

Pornirea serviciului de nume:
>start rmiregistry [port_number]

CatServerStart.java

```
import java.rmi.*;

public class CatServerStart {
    public static void main(String argv[]) throws java.rmi.RemoteException {
        SimpleCat sc = new SimpleCat("Romanian Cat");
        System.setSecurityManager(new RMISecurityManager());
        try {
            CatServer cs = new CatServer(sc);
            Naming.rebind("rmi://localhost:5555/CatServer",cs);
            System.out.println("Server CatServer announced");
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

RMI: Exemplu de implementare (2)

CatServerInterface.java

```
public interface CatServerInterface extends java.rmi.Remote {
    String getCatName() throws java.rmi.RemoteException;
    SimpleCat getCat() throws java.rmi.RemoteException;
}
```

Extindem interfata CatServerInterface adaugand o metoda pentru a returna un obiect de tip SimpleCat

CatServer.java

```
import java.rmi.*;

public class CatServer extends UnicastRemoteObject implements CatServerInterface {
    private static final long serialVersionUID = 102L;

    SimpleCat sc = null;
    CatServer(SimpleCat sc) throws java.rmi.RemoteException {
        super();
        this.sc =sc;
    }

    public String getCatName() throws RemoteException {
        return sc.getName();
    }

    public SimpleCat getCat() throws RemoteException {
        return sc;
    }
}
```

Serverul CatServer va implementa metoda aditionala care returneaza un obiect de tip SimpleCat

RMI: Exemplu de implementare (2)

Clientul, modificat pentru a accesa metoda getCat, care returneaza un obiect de tipul SimpleCat

ClientRMICat.java

```
import java.rmi.Naming;

public class ClientRMICat {
    private static int GET_CAT = 1;
    static public void main(String argv[]){
        System.setSecurityManager(new RMISecurityManager());
        String where = "rmi://localhost:5555/CatServer";
        try {
            String s = null;
            SimpleCat sc = null;
            Remote robj = Naming.lookup(where);
            System.out.println("Reference to server obtained: " + robj.getClass());
            CatServerInterface csi = (CatServerInterface) robj;
            if(1 == GET_CAT) {
                sc = csi.getCat();
                System.out.println("Reference to object passed by remote object obtained: " + sc.getClass());
            }
            else {
                s = (String)csi.getCatName();
                System.out.println("Value of string from remote object obtained: " + s);
                System.out.println("Cat name obtained: " + s);
            }
        }
        catch (Exception e){
            System.out.println("Error" + e);
            System.exit(0);
        }
    }
}
```

RMI: Exemplu de implementare (3)

Interfata pentru clase care efectueaza calcule

CompInterface.java

```
public interface CompInterface {
    int doComputation(int x, int y);
}
```

Exemplu de clasa care efectueaza calcule

CompOne.java

```
import java.io.*;
public class CompOne implements Serializable, CompInterface {
    private static final long serialVersionUID = 1L;
    public int doComputation(int x, int y) {
        return x + y;
    }
}
```

Implementarea unui client care primeste un calculator de la server

ClientRMIComp.java

```
import java.rmi.Naming;
import java.rmi.RMISecurityManager;
import java.rmi.Remote;

public class ClientRMIComp {
    static public void main(String argv[]) {
        CompInterface ci = null;
        System.setSecurityManager(new RMISecurityManager());
        try {
            Remote robj = Naming.lookup("ServerComp");
            System.out.println("Server reference obtained: " + robj.getClass());
            ServerCompInterface sci = (ServerCompInterface) robj;
            ci = sci.getComputation();
            System.out.println("Reference to " + ci + "obtained");
        }
        catch (Exception e) {
            System.out.println("Error " + e);
            System.exit(0);
        }
        int z = ci.doComputation(3, 5);
        System.out.println("Result is " + z);
    }
}
```

Interfata pentru server, returneaza o interfata de calcul

ServerCompInterface.java

```
public interface ServerCompInterface {
    public CompInterface getComputation()
                    throws java.rmi.RemoteException;
}
```

Server, implementeaza interfata de calcul, in acest caz returneaza un obiect de tip CompOne

ServerComp.java

```
import java.rmi.RemoteException;

public class ServerComp extends UnicastRemoteObject
                        implements ServerCompInterface {
    private static final long serialVersionUID = 2L;
    ServerComp() throws RemoteException {
    }
    public CompInterface getComputation() throws RemoteException {
        CompInterface c = new CompOne();
        return c;
    }
}
```

RMI: Incarcarea dinamica a claselor

Incarcarea dinamica a claselor

Incarcarea dinamica a datelor este un concept important in Java. Applet-urile presupun incarcarea dinamica a datelor.
Elementele care concura la realizarea RMI si pasii parcursi sunt:

- •Se face legatura intre un nume (prin care se va referi serviciul) si o adresa (URL) la care serviciul poate fi accesat. Aceasta adresa este de tipul: **rmi://adresa_server:port/nume**.
- •Clientul "stie" interfata implementata de obiectul accesat la distanta dar are nevoie de referinta (stub-ul) corespunzator numelui de serviciu (obiectului accesat la distanta);
- •Serviciul de nume transmite clientului stub-ul clasei. Daca definitia clasei se gaseste local (este in CLASSPATH) atunci clientul va incarca clasa local. Daca nu, va incerca sa foloseasca proprietatea codebase pentru obiectul la distanta;
- •Daca este nevoie, clientul poate solicita incarcarea clasei obiectului aflat la distanta;
- •Se obtine stub-ul obiectului aflat la distanta si al altor clase utilizate; Stub-ul functioneaza ca un proxy pntru obiectul aflat la distanta.

Pot exista una dintre urmatoarele situatii:

- •Totul local;
- •Incarcare dinamica la client;
- •Incarcare dinamica la server;
- •Client bootstrapat. Codul pentru client este incarcat dinamic de catre un mic program de tip incarcator;
- •Server bootstrapat. Codul pentru servereste incarcat dinamic de catre un mic program de tip incarcator;

RMI: Avantaje utilizare; probleme de performanta

Avantaje utilizare RMI

-RMI este complet object oriented - poate sa transmita ca argumente obiecte si poate sa obtina ca rezultate obiecte.
-RMI permite transmiterea de "behaviors" de la server la client si invers. Altfel spus, modificarea implementarii unei clase trebuie facuta numai intr-un singur loc (pe client sau pe server).
-Politici de securitate standard sau configurate de utilizator permit verificarea codului;
-Implementarea este simpla;
-Posibilitatea conectarii la sisteme vechi. Utilizand interfata nativa JNI, se poate converti o interfata de server scrisa in orice alt limbaj la o interfata Java. Utilizand interfata JDBC, se poate face legatura cu orice baza de date existenta;
-RMI – 100% pure Java. Scriind aplicatii care utilizeaza numai RMI, JDBC, JNI, ele vor fi portabile la nivel de cod sursa si pot fi compilate pe orice platforma Java.
-Exista mecanism de garbage collection (limitat, nu poate rezolva structuri circulare).
-Se poate utiliza cu multithreading.

Probleme de performanta in utilizarea RMI

Performanta scazuta, datorita implementari non-native, in comparatie cu alte implementari RPC.
-Crearea unui obiect la distanta care va fi referit la distanta presupune suplimentar creearea stub si skeleton;
-Transferul de parametri presupune marshalling si unmarshalling; accesul la distanta introduce de asemenea un overhead;
-Durata apelului depinde de latenta retelei in care se executa operatia;
Se poate optimiza consumul de resurse prin scrierea unui mecanism propriu de serializare.