Medii de proiectare și programare

2017-2018
Curs 6

Conținut curs 6

- Remote Procedure Call
- RMI, Spring Remoting
- .NET Remoting

Remote Procedure Call

- Apelul procedurilor la distanță (eng. *Remote procedure call - RPC*) este o tehnologie de comunicare între procese care permite unei aplicații să inițieze execuția unei subrutine sau a unei proceduri în alt spațiu de adrese, fără ca programatorul să scrie explicit codul corespunzător interacțiunii dintre procese.
- Programatorul scrie aproximativ același cod indiferent dacă apelează o rutină locală (din același proces) sau una la distanță (din alt proces, în alt spațiu de adrese).
- Când se folosește paradigma orientată pe obiecte RPC - apeluri la distanță sau apelul metodelor la distanță.
- RPC este adesea folosit pentru implementarea aplicațiilor client-server.
- Un apel la distanță este inițiat de client prin trimiterea unei cereri către un server la distanță pentru execuția unei proceduri cu parametrii dați. Serverul trimite un răspuns clientului, iar aplicația își continuă execuția.
- Cât timp serverul procesează cererea clientului, execuția clientului este blocată (așteaptă până când serverul a terminat procesarea cererii).

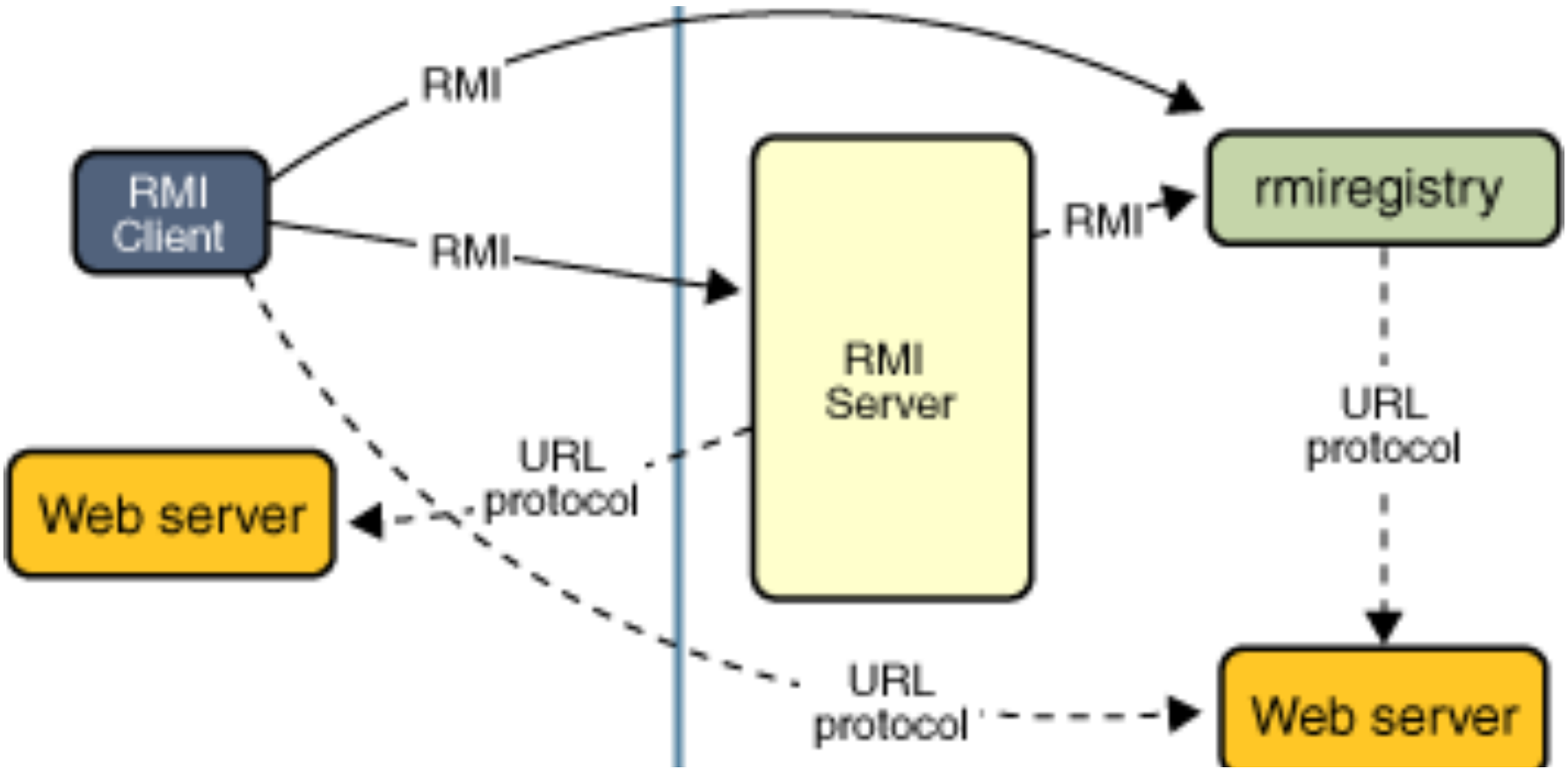
Java RMI

- Tehnologia Java Remote Method Invocation (RMI) permite unui obiect dintr-o mașină virtuală Java să apeleze metodele unui obiect dintr-o altă mașina virtuală Java.
- Aplicațiile RMI conțin cel puțin doua programe separate: *server* și *client*.
  - Programul server creează obiecte ce pot fi apelate la distanță, face referințele către aceste obiecte disponibile clienților și așteaptă ca clienții să apeleze metode asupra obiectelor.
  - Programul client obține o referință către unul sau mai multe obiecte la distanță, iar apoi apelează metode asupra acestuia (acestora).
- RMI furnizează mecanismul prin care serverul și clientul comunică și își transmit informații.

Java RMI

- *Localizarea obiectelor la distanță (remote)*. Mecanisme pentru obținerea referințelor către obiecte remote:
    - O aplicație înregistrează obiectele remote folosind *rmiregistry* - instrument oferit de RMI, bazat pe nume.
    - O aplicație poate transmite/primi referințe către obiecte remote, în urma altor apeluri la distanță.
- *Comunicarea cu obiecte remote.* Detaliile comunicării între obiecte remote și clienți sunt rezolvate de RMI.
    - Apelurile la distanță sunt asemănătoare cu apelurile de metode Java obișnuite.
- *Încărcarea claselor corespunzătoare obiectelor transmise ca și parametrii/ rezultat*. RMI permite transmiterea unor obiecte a căror clase nu erau disponbile la pornirea aplicației.

Java RMI


RMI
RMI Client
RMI
RMI
rmiregistry
RMI Server
URL protocol
URL protocol
Web server
URL protocol
Web server

RMI

- *Obiecte remote*: obiecte care conțin metode ce pot fi apelate din diferite instanțe de mașini virtuale. Un obiect devine remote dacă clasa corespunzătoare implementează o interfață *remote*, cu următoarele caracteristici:
  - O interfață remote moștenește interfața **`java.rmi.Remote`**.
  - Fiecare metodă din interfață specifică excepția java.rmi.RemoteException în signatura sa (poate să arunce și alte excepții).

```
import java.rmi.*;
public interface IServer extends Remote{
 public ResultObject method(Param1 p1, Param2 p2,…) throws
   RemoteException;
}
```

- *Obiecte locale* (obiecte non-remote): obiecte folosite în apelurile la distanță (parametrii sau rezultat).
  - Trebuie să fie serializabile.

Arhitectura unei aplicații bazate pe RMI

- *Biblioteca comună* (interfețe remote și clasele corespunzătoare parametrilor).
  - O interfață remote specifică metodele remote care pot fi apelate de clienți, parametrii acestora, tipurile parametrilor, rezultatul returnat, tipul rezultatului.
- *Server* (Implementarea interfețelor remote).
  - Obiectele remote trebuie să implementeze una sau mai multe interfețe remote.
  - Pot să implementeze alte interfețe a căror metode pot fi apelate doar local.
- *Client(Clienții)* (Implementarea clienților).
  - Clienții care folosesc obiecte remote pot fi implementați oricând după definirea interfețelor remote și “publicarea” acestora.

Pachetul `java.rmi`

- Interfața **`Remote`**: folosită pentru identificarea obiectelor a căror metode pot fi apelata dintr-o altă mașina virtuală. Orice obiect remote trebuie să implementeze direct sau indirect această interfață. Doar metodele specificate într-o interfață remote pot fi apelate la distanță.
- Clasa **`Naming`**: conține metode statice pentru păstrarea și obținerea referințelor către obiecte remote în/dintr-un registru specific.
  - Fiecare metodă a clasei **`Naming`** are un parametru de tip String cu următorul format:

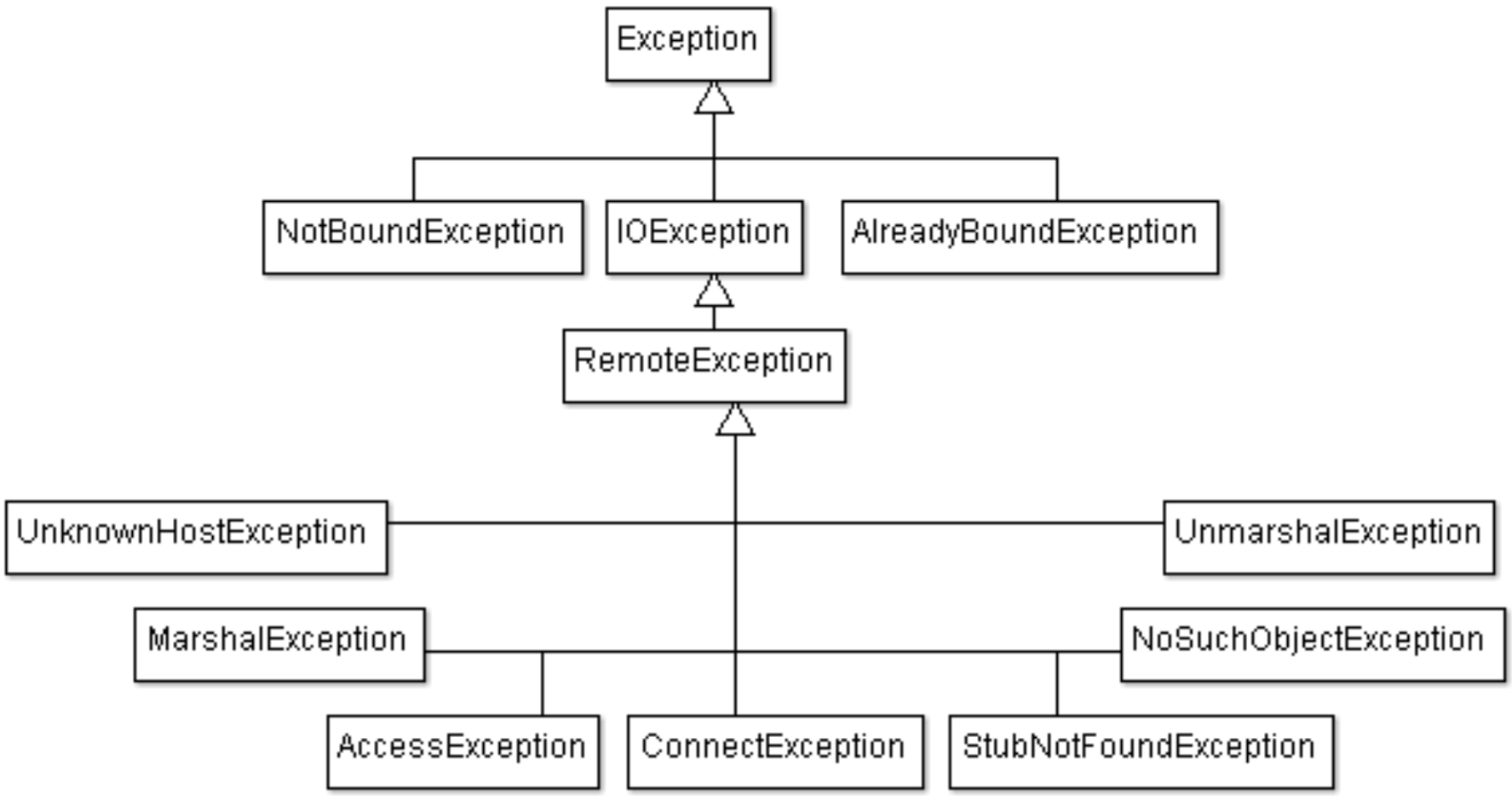
`//host:port/name`

- **`bind(String name, Remote obj)`** asociaza numele de un obiect remote.
- **`lookup(String name):Remote`** returnează referința/proxy/stub către un obiect remote asociat cu numele specificat.
- **`rebind(String name, Remote obj)`** reactualizează asocierea dintre numele specificat și obiectul remote.
- **`unbind(String name)`** distruge/șterge asocierea dintre numele specificat și obiectul remote corespunzător.

java.rmi package

- Clasa **`RMISecurityManager -`** subclasă a clasei **`SecurityManager`** folosita de aplicațiile RMI care descarcă cod. Class loader-ul RMI nu va descărca orice clase de la locații remote dacă nu a fost setat un obiect de tip SecurityManager.

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



Pachetul java.rmi.registry

- Un registru este un obiect remote care păstrează asocieri între obiecte remote și un nume specific.
- Un server își înregistrează obiectele remote folosind registrul, iar clienții le vor căuta folosind același registru.
- Când un obiect (clientul) vrea să apeleze o metodă a unui obiect remote trebuie întâi să caute obiectul remote folosind numele. Registrul va returna clientului o referință către obiectul remote, iar clientul va folosi această referință pentru apelul metodei la distanță.
- Interfața **`Registry`**: este o interfață *remote* pentru un obiect remote de tip registru care conține metode pentru păstrarea și regăsirea referințelor către obiecte remote.
  - **`bind`**, **`lookup`**, **`rebind`**, **`unbind`**
- Clasa **`LocateRegistry`**
  - folosită pentru obținerea referinței către un registru aflat la o anumită adresă (inclusiv localhost)
  - folosită pentru crearea unui obiect de tip registru care acceptă cereri de la clienti la portul specificat.

Exemplu RMI

- Definirea interfeței remote:

```
package rmi.services;

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


public interface ITransformer extends Remote {
    String transform(String text) throws RemoteException;
}
```

Exemplu RMI

- Implementarea interfeței remote:

```
package rmi.server;
import rmi.services.ITransformer;
import java.rmi.RemoteException;
import java.util.Date;

public class TransformerImpl implements ITransformer{
    public String transform(String text) throws RemoteException {
        System.out.println("Method called "+text);
        return text.toUpperCase()+(new Date());
    }
}

//for dynamically stub creation
//public class TransformerImpl extends UnicastRemoteObject
                        implements ITransformer
```

Exemplu RMI

- Crearea arhivei jar care conține clasele și interfețele comune serverului și clienților:

```
jar cvf services.jar rmi/services/ITransformer.class
```

- Compilarea clasei care implementează interfața remote folosind fișierul creat anterior.
- (Versiuni <1.5) Crearea server stub-ului folosind `rmic` :

```
rmic -classpath services.jar rmi.server.TransformerImpl
```

Se crea un fisier numit `rmi.server.TransformerImpl_stub.class`

- Crearea fișierului pentru permisiuni:

```
grant {
    permission java.security.AllPermission;
    permission java.net.SocketPermission "127.0.0.1:1024-",
   "connect,resolve";
};
```

Exemplu RMI

• Crearea clasei care pornește serverul:

```
public class StartServer {
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
         }
         try {
            String name = "Transformer";
            ITransformer engine = new TransformerImpl();
            ITransformer stub =(ITransformer)
   UnicastRemoteObject.exportObject(engine, 0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind(name, stub);
            System.out.println("Transformer bound");
        } catch (Exception e) {
            System.err.println("Transformer exception:"+e);
        }
   }
}
```

Exemplu RMI

- Pornirea serverului:
  - Start RMI registry:
    - **`Start rmiregistry`** (Windows)
    - **`Rmiregistry`** (Linux, Mac OS)
  - Copierea claselor comune într-un loc ce poate fi accesat de clienți public:
    - http://scs.ubbcluj.ro/~xyz/rmi
    - c:/temp/rmi/transformer/server/
  - Pornirea serverului folosind comanda:

```
java -cp .;<path to>services.jar
-Djava.rmi.server.codebase=file:/c:/temp/rmi/transformer/server/
-Djava.rmi.server.hostname=localhost
-Djava.security.policy=server.policy
 StartServer
```

Exemplu RMI

- Pornirea clientului:

```
public class StartClient {
  public static void main(String args[]) {
    if (System.getSecurityManager() == null)
            System.setSecurityManager(new SecurityManager());
    try {
       String name = "Transformer";
       Registry registry = LocateRegistry.getRegistry("localhost");
       ITransformer comp = (ITransformer) registry.lookup(name);
       String text="Ana are mere.";
       String resp=comp.transform(text);
       System.out.println("Result: "+resp);
    } catch (RemoteException e) {…}
     catch (NotBoundException e) {…}
     }
}
```

Exemplu RMI

- Crearea fișierului de permisiuni pentru client:

```
grant {
    permission java.security.AllPermission;
    permission java.net.SocketPermission "127.0.0.1:1024-",
  "connect,resolve";
};
```

- Execuția clientului

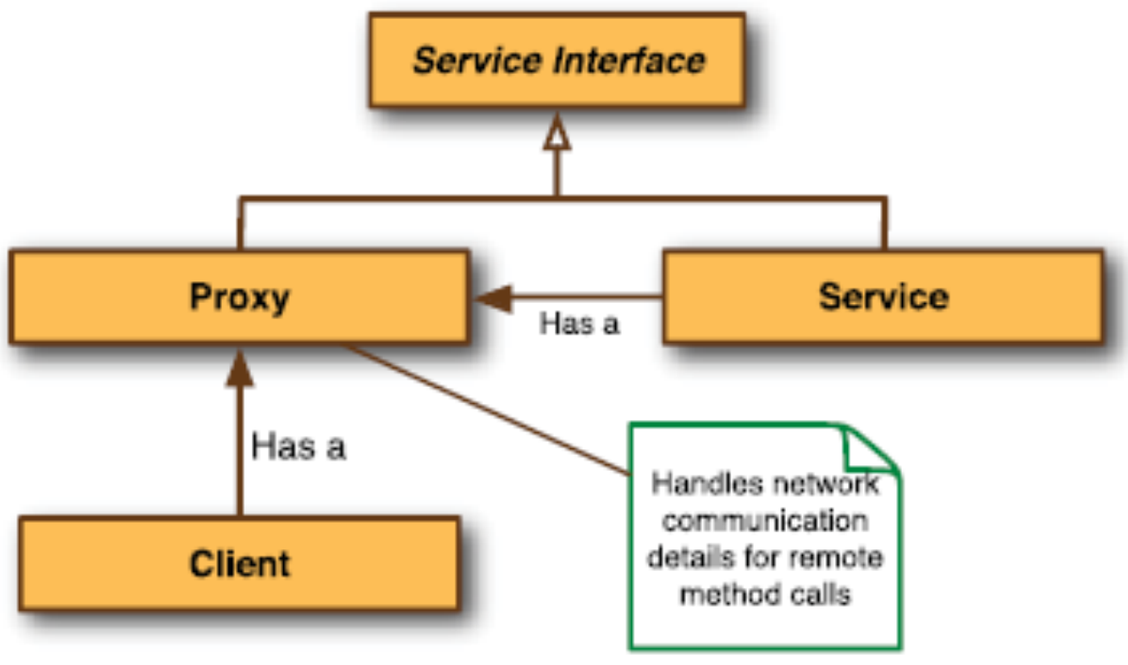
```
java -cp .;<path to>services.jar
-Djava.security.policy=client.policy StartClient
```

Spring Remoting

- Frameworkul Spring suportă diferite modele de apeluri la distanță: Java RMI (RMI), Caucho's Hessian și Burlap, și HTTP invoker specific Spring.
- În toate modelele, serviciile (obiectele remote) pot fi configurate folosind bean-uri Spring (obiecte create folosind containerul Spring).
- Se folosește un "*proxy factory bean"* care permite injectarea dependențelor către obiecte remote ca și cum ar fi obiecte locale.
- Clientul apelează metodele de la proxy ca și cum acesta ar furniza funcționalitatea cerută.
- Proxy-ul comunică cu obiectul remote în numele clientului. Se ocupă de toate detaliile conexiunii și a apelului metodelor la distanță.

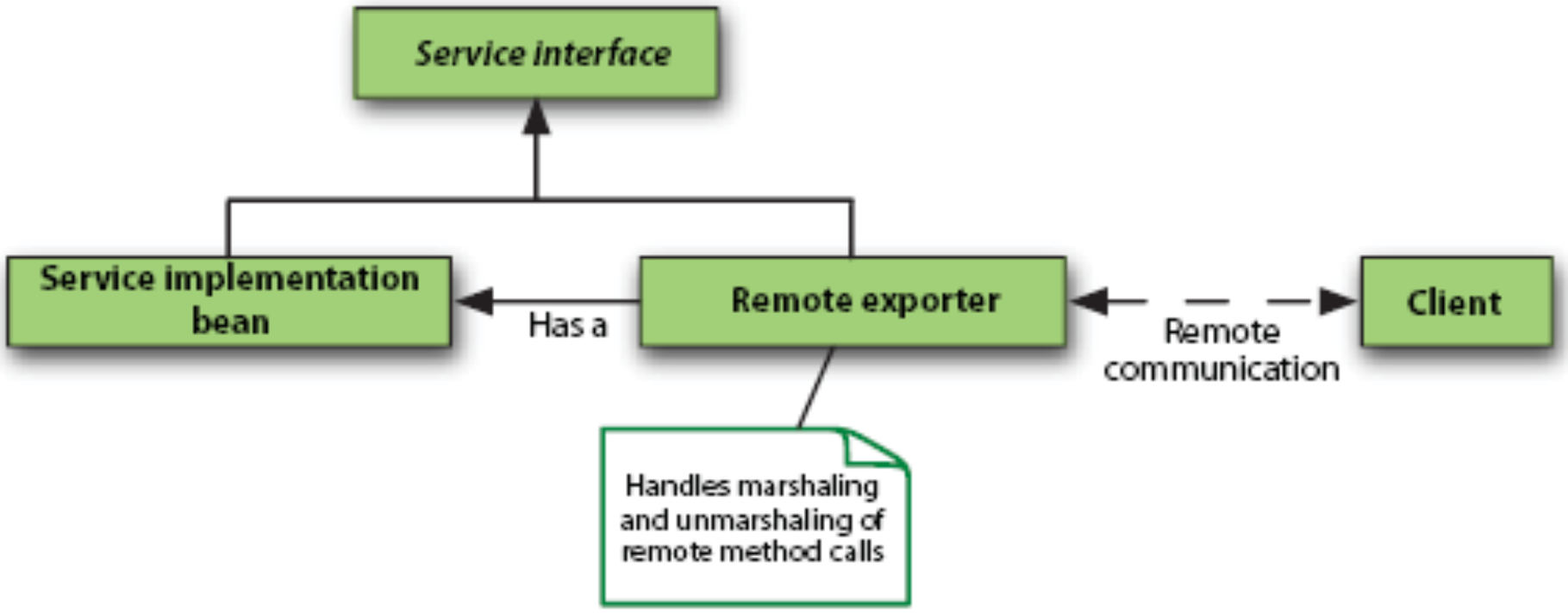
Spring Remoting

- În Spring, obiectele remote au proxy pentru a putea fi asociate codului client ca și cum ar fi beanuri Spring obișnuite.



Spring Remoting

- Beanurile Spring pot fi “exportate” ca și obiecte remote (servicii) folosind beanuri de tip remote exporter.
- Dacă apelul unei metode la distanță aruncă o excepție **`java.rmi.RemoteException`**, obiectul proxy tratează excepția și o rearuncă ca și excepție **`RemoteAccessException`** (runtime).
- Codul clientului nu este obligat să trateze excepții de tip **`RemoteAccessException.`**



Spring Remoting

Etape pentru crearea și publicarea unui obiect remote folosind RMI:

1. Crearea interfeței care moștenește **`java.rmi.Remote`**.
2. Scrierea clasei care implementează interfața, cu metode care specifică excepția **`java.rmi.RemoteException`** la signatura lor.
3. Rularea compilatorului RMI (**`rmic`**) pentru obținerea claselor stub. (v <1.5)
4. Pornirea registrului RMI pentru obiectele remote.
5. Înregistrarea obiectului remote la registru.

`RemoteException` și **`MalformedURLExceptions`** sunt adesea aruncate, chiar dacă ele indică de obicei situații din care programul nu își poate continua execuția (în blocul try-catch).

Configurarea unui obiect RMI Service în Spring

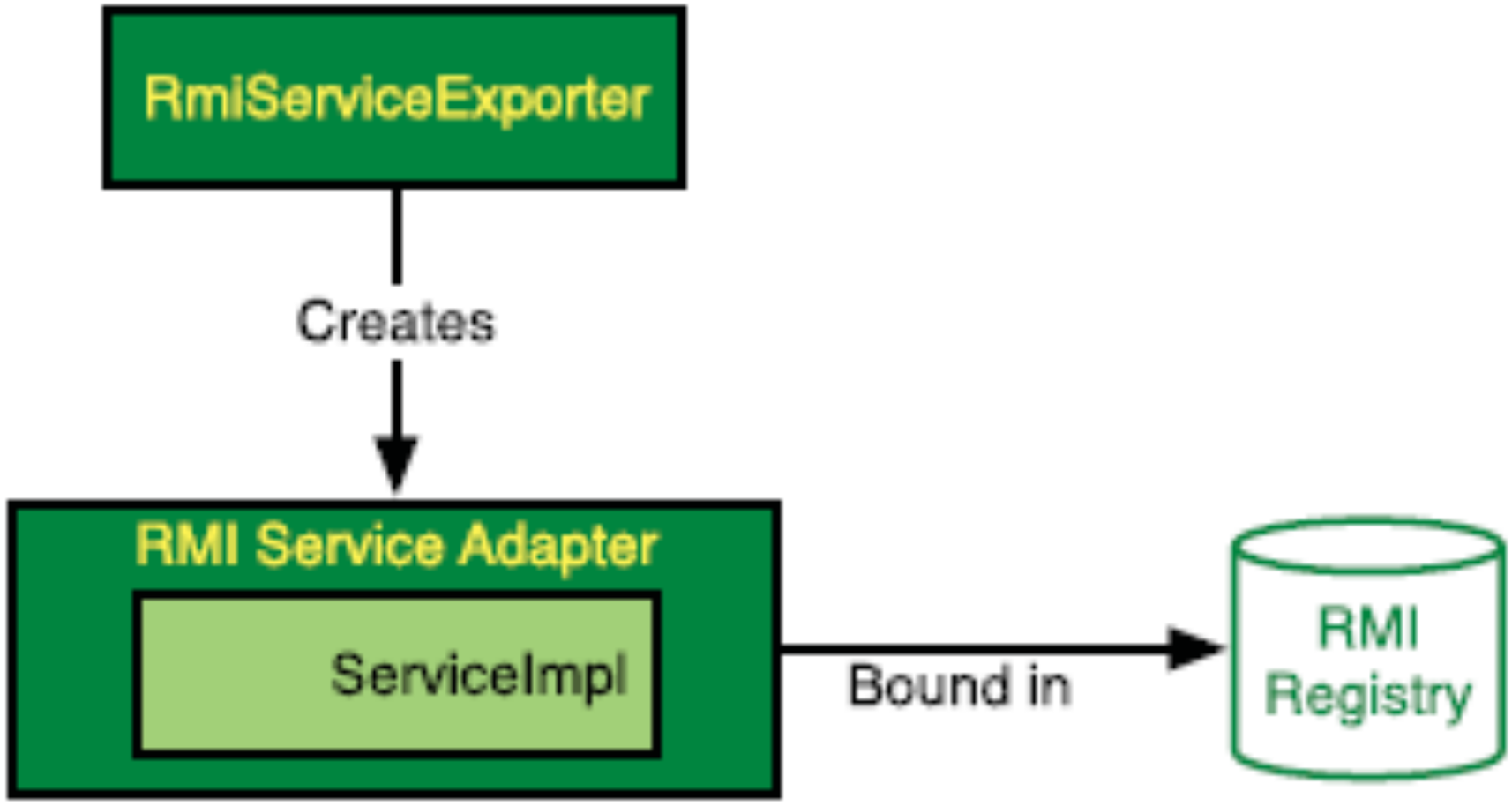
- Spring oferă o modalitate mai simplă de a crea și folosi obiecte remote RMI.
- Programatorii trebuie doar să scrie codul corespunzător funcționalității oferite de obiectul remote.
- Nu este necesară asocierea codului de clase/interfețe specifice RMI (remote interface, metode care aruncă **`RemoteException`**).

```
public interface ITransformer {
    String transform(String text);
}

import java.util.Date;
public class TransformerImpl implements ITransformer{
    public String transform(String text) {
        System.out.println("Method called "+text);
        return text.toUpperCase()+(new Date());
    }
}
```

Configurarea unui obiect RMI Service în Spring

- **`RmiServiceExporter`** “exportă” orice bean creat folosind containerul Spring ca și un obiect remote (RMI Service)
- Bean-ul respectiv este inclus (eng. *wrapped*) într-o clasa adapter. O instanță a clasei adapter este asociată numelui specificat într-un registru RMI și ea va primi cererile clienților.



Configurarea unui obiect RMI Service în Spring

```
//remote service configuration

//spring-server.xml

<bean id="transformerService"
      class="transformer.services.impl.TransformerImpl"/>

<bean class="org.springframework.remoting.rmi.RmiServiceExporter">

  <property name="serviceName" value="TransformerService"/>
  <property name="service" ref="transformerService"/>
  <property name="serviceInterface"
    value="transformer.services.ITransformer"/>
  <property name="servicePort" value="1099"/>
</bean>
```

Configurarea unui obiect RMI Service în Spring

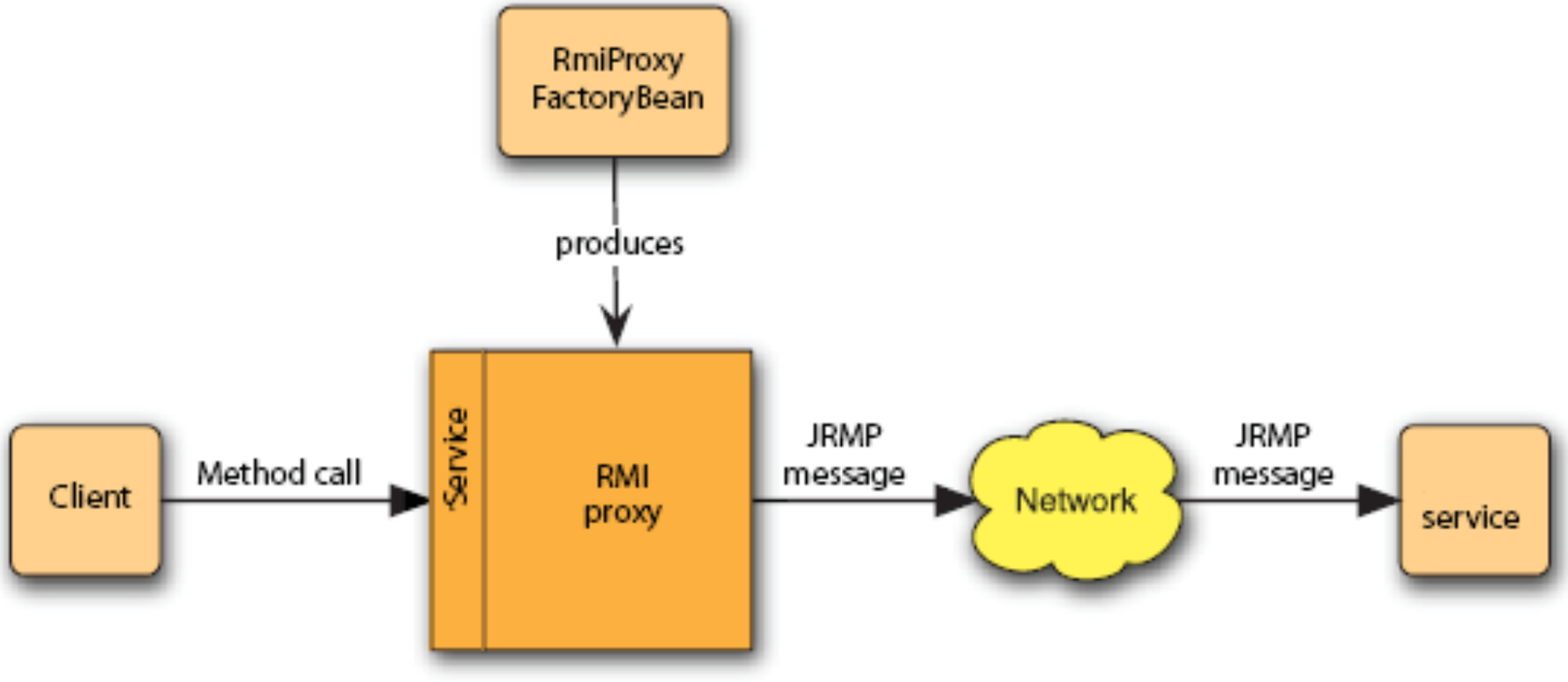
- Proprietatea **`serviceName`** specifică numele asociat obiectului remote.
- Proprietatea **`serviceInterface`** specifică interfața pe care o implementează obiectul.
- Implicit **`RmiServiceExporter`** încearcă să asocieze obiectul remote registrului RMI pe portul 1099 a mașinii locale. Dacă nu există un registru la acest port, **`RmiServiceExporter`** va porni unul.
- Dacă se dorește pornirea registrului la o altă adresă (host și port) acestea pot fi specificate folosind proprietățile **`registryPort`** și **`registryHost`**.

Obținerea referinței unui obiect remote

- Clienții Java RMI trebuie să folosească clasa **`Naming`** pentru a căuta obiecte remote într-un registru RMI.
- Clasa **`RmiProxyFactoryBean`** din Spring creează un proxy corespunzător unui obiect remote RMI.
- Obiectul proxy creat poate fi transmis (injectat) tuturor beanurilor dependente de obiectul remote.
- Codul clientului nu știe că comunică cu un obiect remote.
- Obiectul proxy prinde toate excepțiile **`RemoteExceptions`** care pot să apară și le rearuncă ca și excepții unchecked (runtime).

Obținerea referinței unui obiect remote

- **`RmiProxyFactoryBean`** creează un obiect proxy care comunică cu obiectul remote. Clientul comunică cu obiectul proxy folosind interfața remote corespunzătoare obiectului remote.
- Apelul metodelor remote este la fel cu apelul metodelor obișnuite.



Obținerea referinței unui obiect remote

```
//spring-client.xml
<bean id="transformerService“
    class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
     <property name="serviceUrl" value="rmi://127.0.0.1:1099/
      TransformerService"/>
     <property name="serviceInterface"
      value="transformer.services.ITransformer"/>
    </bean>
```

- URL-ul obiectului remote este setat folosind proprietatea **`serviceUrl`** a bean-ului **`RmiProxyFactoryBean`**.
- Interfața corespunzătoare obiectului remote este specificată folosind proprietatea **`serviceInterface`**.

Pornirea serverului

```
 //StartServer.java

public class StartServer {

    public static void main(String[] args) {

        ApplicationContext factory = new

     ClassPathXmlApplicationContext("classpath:spring-server.xml");

    }

}
```

Execuția clientului

```
 //StartClient.java
public class StartClient {
    public static void main(String[] args) {
        ApplicationContext factory = new
     ClassPathXmlApplicationContext("classpath:spring-client.xml");
       ITransformer
     transformerServ=(ITransformer)factory.getBean(“transformerService”);
        String text="Ana are mere";
        System.out.println("Sending text = " + text);
        String response=transformerServ.transform(text);
        System.out.println("Received response = " + response);
    }
}
```

.NET Remoting

- Remoting este tehnologia .NET care permite diferitor procese și componente să interacționeze.
- Tehnologia Remoting stă la baza aplicațiilor distribuite în .NET.
- Implementările fac diferența dintre *obiecte remote* și *obiecte mobile*.
- Un *obiect remote* oferă posibilitatea de a executa metode pe calculatoare diferite, de a transmite parametrii și de obține rezultatele. Obiectul remote rămâne pe server și doar o referință către acesta va fi transmisă către celelalte calculatoare.
- *Obiectele mobile* depășesc anumite “limite”. Ele sunt serializate (binar, XML, etc) într-un anumit context, iar apoi sunt deserializate în celălalt context. Clientul și serverul vor avea copii ale aceluiași obiect.

.NET Remoting

- Există două tipuri de obiecte: obiecte transmise prin referință și obiecte transmise prin valoare.
- *MarshalByRefObjects* sunt obiecte remote care sunt create și inițializate pe server. Ele rămân pe server și doar o referință *ObjRef* va fi transmisă apelanților.
- Adesea, clientul nu are codul binar al obiectelor remote în fișierele sale. Are doar codul binar al unei interfețe sau al unei clase de bază.
- Fiecare metoda, inclusiv proprietățile (set și get) vor fi executate pe server.
- Frameworkul .NET va genera automat obiecte proxy care vor redirecționa apelul către membrii obiectelor remote. Clientul apelează o metodă remote în același mod în care apelează o metodă locală.

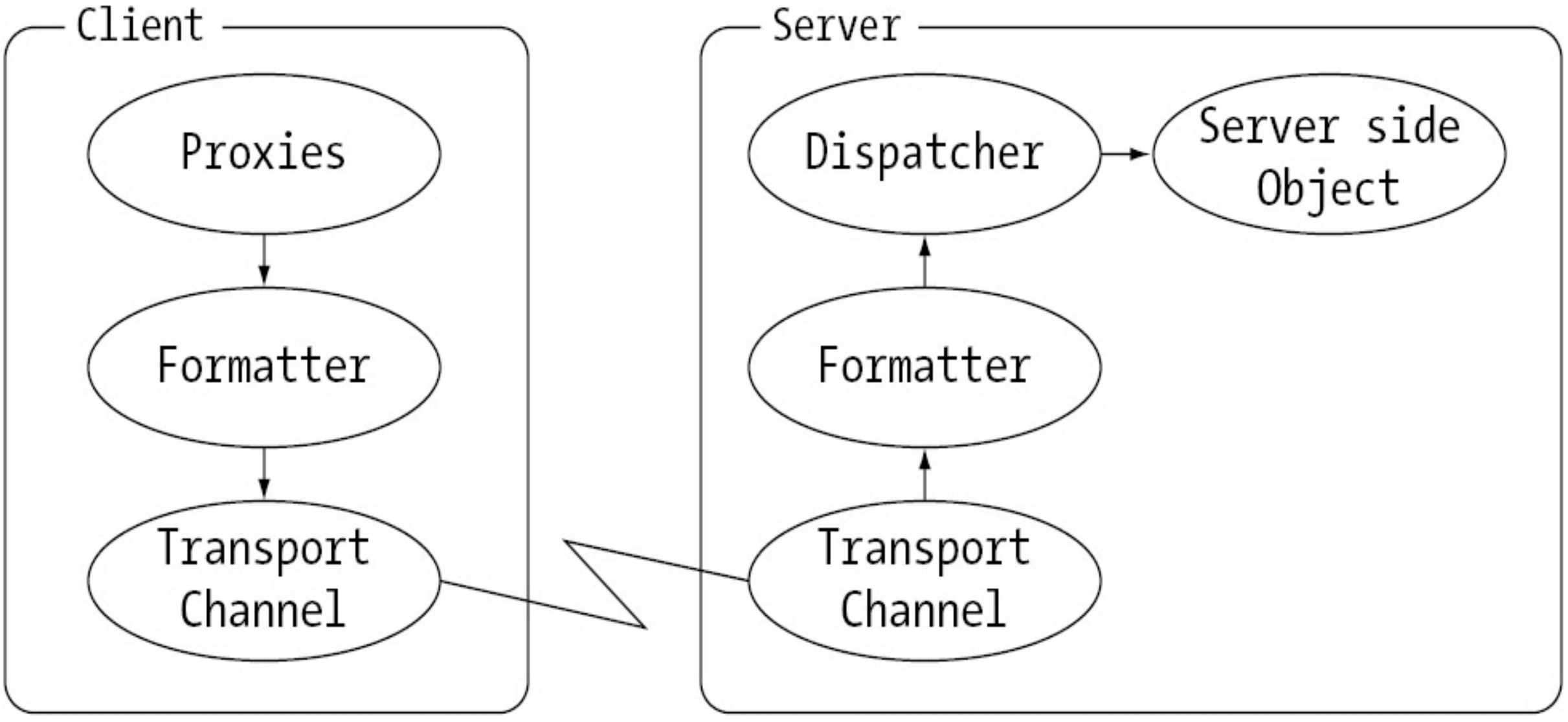
Obiecte transmise prin valoare în Remoting

- Când acest tip de obiecte sunt transmise ca și parametrii sau ca și rezultat, ele sunt serializate, iar apoi o copie va fi construită în celălalt capăt al canalului de comunicare.
- După reconstruire, nu există o notație specifică pe client/server pentru acest tip de obiecte: fiecare proces are o copie a obiectului și o vor folosi în mod independent.
- Metodele apelate asupra acestor obiecte se vor executa în același context ca și apelantul metodei.
- Singura cerință pentru ca un obiect să poată fi transmis prin valoare este să fie serializabil (atributul **`Serializable`** sau interfața **`ISerializable`**).

Observație:

Atât serverul cât și clientul trebuie să aibă codul binar corespunzător obiectelor transmise prin valoare (pentru serializare și deserializare).

Remoting


Client
Proxies
Formatter
Transport Channel
Server
Dispatcher
Server side Object
Formatter
Transport Channel

Arhitectura unei aplicații cu Remoting

- Cel puțin 3 assemblies sunt necesare pentru orice aplicație Remoting:
  - Un assembly *partajat* (.dll) care conține codul binar corespunzător obiectelor serializabile și interfețele sau clasele de bază corespunzătoare obiectelor MarshalByRefObjects.
  - Un assembly *server* (.exe) care conține codul binar corespunzător obiectelor remote (*MarshalByRefObjects*).
  - Un assembly *client* (.exe) care folosește obiectele remote (MarshalByRefObjects).

Observație

Serverul și clientul au referință către assembly-ul partajat.

System.Runtime.Remoting

- **`System.Runtime.Remoting`** conține clase și interfețe care permit programatorilor să creeze și să configureze aplicații distribuite.
- Clasa **`RemotingConfiguration`** conține metode statice pentru configurare. Metoda **`Configure`** permite programatorilor configurarea aplicației distribuite folosind fișiere în format XML.
- Clasa **`RemotingServices`** conține metode utile în folosirea și publicarea obiectelor remote.
- Clasa **`ObjRef`** păstrează informații relevante necesare activării și comunicării cu un obiect remote. Este o reprezentare serializabilă a unui obiect transmis la o locație remote folosind un canal de comunicare unde este reconstruit și poate fi folosit pentru a crea un proxy local al obiectului remote.

System.Runtime.Remoting

- Namespace **`System.Runtime.Remoting.Channels`** conține clase care sprijină și tratează canalele de comunicare folosite pentru transportul datelor când un client apelează o metodă la distanță.
- Namespace **`System.Runtime.Remoting.Channels.Http`** conține definiția canalelor care folosesc protocolul HTTP pentru transportul mesajelor și a obiectelor la/de la locații remote.
- Namespace **`System.Runtime.Remoting.Channels.Ipc`** definește un canal de comunicare pentru Remoting care folosește comunicarea între procese pe aceeași mașină (calculator) cu sistem de operare Windows.
  - Nu folosește comunicarea prin rețea și este mult mai rapid decât canalele de comunicare care folosesc protocoalele HTTP sau TCP, dar poate fi folosit doar pentru comunicarea între procese care rulează pe același calculator.

System.Runtime.Remoting

- Namespace **`System.Runtime.Remoting.Channels.Tcp`** conține canale care folosesc protocolul TCP pentru transmiterea mesajelor și a obiectelor la locații remote. Implicit, canalele TCP serializează obiectele transmise folosind formatul binar.
  - **`TcpClientChannel`** implementează un canal client care folosește protocolul TCP pentru transmiterea mesajelor (for remote calls).
  - **`TcpServerChannel`** implementează un canal server pentru apelurile la distanță folosind protocolul TCP pentru transmiterea mesajelor.
  - **`TcpChannel`** oferă un canal bidirecțional de comunicare folosind protocolul TCP
- Combină clasele **`TcpClientChannel`** și **`TcpServerChannel`** și poate fi folosit pentru transmiterea și primirea mesajelor folosind protocolul TCP.

Exemplu Remoting

- O aplicație simplă client/server:
  - Obiectul remote are o metodă care transformă un text în text cu litere mari și adaugă data și ora la care a fost primit textul conexiuni.
  - Clientul obține o referință la obiectul remote, apelează metoda și tipărește rezultatul primit.

Exemplu Remoting – assembly partajat

```
namespace RemotingServices
{
    public interface IServer{
        String transform(String txt);
    }
}
```

Va fi compilat ca și **RemotingServices.dll**

Exemplu Remoting – assembly server

```
using RemotingServices;
namespace RemotingServer
{
    class ServerImpl: MarshalByRefObject, IServer {
    public string transform(string txt)
        {
            Console.WriteLine("Processing text: "+txt);
        return txt.ToUpper() +' ' +DateTime.Now;
        }
    }
}
```

Exemplu Remoting – assembly server

```
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
namespace RemotingServer{
    class RemotingServer {
        static void Main(string[] args) {
       TcpChannel channel=new TcpChannel(55555);
            ChannelServices.RegisterChannel(channel,false);

   RemotingConfiguration.RegisterWellKnownServiceType( typeof(ServerImpl),
                                        "StringConvertor",
   WellKnownObjectMode.Singleton);
       // the server  keeps running until keypress.
            Console.WriteLine("Server started ...");
            Console.WriteLine("Press <enter> to exit...");
            Console.ReadLine();
        }
   }
}
```

Exemplu Remoting – assembly client

```
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using RemotingServices;
namespace RemotingClient{
  class RemotingClient{
     static void Main(string[] args){
        try{
           TcpChannel channel = new TcpChannel();
           ChannelServices.RegisterChannel(channel, false);
IServer server = (IServer)Activator.GetObject(typeof(IServer),
   "tcp://localhost:55555/StringConvertor");
         String txt = "Ana are mere.";
            Console.WriteLine("Calling remote method "+txt);
            String response = server.transform(txt);
            Console.WriteLine("Response:"+response);
    }catch(Exception e){…}
}}}
```

Tipuri Remoting

- Sunt posibile două tipuri de interacțiuni remote între componente:
  - Folosește obiecte serializabile care sunt transmise procesului remote (obiecte ByValue).
  - Folosește obiecte remote care permit clienților să apeleze metodele lor (MarshalByRefObjects).
- Observații:
  - Obiectele ByValue nu sunt obiecte remote. Toate metodele apelate asupra acestora sunt executate local (în același proces ca și apelantul).
  - Codul binar trebuie să fie disponibil și pe client și pe server.
  - Dacă obiectul ByValue conține referințe către alte obiecte acestea trebuie să fie la rândul lor serializabile sau obiecte remote; în caz contrar se va arunca o excepție.

Obiecte MarshalByRefObjects

- Un obiect *MarshalByRefObject* este un obiect remote care rulează pe server și acceptă apeluri de la clienți.
- Datele sale sunt păstrate în memoria serverului și metodele sale sunt executate pe calculatorul server.
- Două tipuri de obiecte *MarshalByRefObjects*: *server-activated objects* (SAOs) și *client-activated objects* (CAOs):
  - Obiecte SAO: Când un client cere o referință către un obiect SAO nu se transmite nici un mesaj serverului. Serverul va fi notificat doar când se apelează metode asupra obiectului SAO.
  - Obiecte CAO: Un obiect CAO se comportă aproape ca și un obiect obișnuit .NET. Când are loc o cerere de creare a unui obiect CAO (folosind `Activator.CreateInstance()` sau operatorul `new`), un mesaj de activare este transmis serverului care păstrează obiectul remote și un obiect remote poate fi creat. În procesul client este creat un proxy care conține un obiect `ObjRef`.

Obiecte SAO

- Când un client apelează o metodă asupra unui obiect SAO, serverul decide dacă o nouă instanță a unui obiect SAO va fi creată sau va fi folosit un obiect SAO deja creat.
- Obiectele SAO pot fi marcate ca fiind *Singleton* sau *SingleCall*.
  - *Singleton:* aceeași instanță a obiectului va fi folosită când sunt apelate metodele la distanță (într-o manieră concurentă).
  - *SingleCall:* pentru fiecare apel la distanță se va crea un nou obiect SAO care va fi distrus după terminarea execuției apelului.

```
RemotingConfiguration.RegisterWellKnownServiceType(
    typeof(<MRO>),  "<URL>", WellKnownObjectMode.SingleCall);
```

```
RemotingConfiguration.RegisterWellKnownServiceType(
        typeof(<MRO>), "<URL>", WellKnownObjectMode.Singleton);
```

Publicarea obiectelor SAO

- Instanțele obiectelor SAO sunt create dinamic în momentul unei cereri de la client (obiecte *SingleCall/Singleton*).
- Dacă un obiect deja creat trebuie publicat (nu are constructor implicit), se folosește metoda **`RemotingServices.Marshal()`**.
- După publicare, obiectul se comportă ca și un obiect SAO *Singleton*.

  Diferența este că obiectul trebuie să fi fost deja creat înainte de publicare.

```
RemoteObject obj = new RemoteObject(<actual params list>);

RemotingServices.Marshal(obj,"IdentificationString");
```

Durata de viață

- Fiecare obiect remote are asociată o durată de viată în momentul creării (eng. *lease*).
- Durata implicită este de 5 minute. La fiecare apel de metodă de către client durata de viață se mărește cu 2 minute (valoare implicită setată la configurare). Durata de viață este decrementată la anumite intervale.
- Când durata de viață ajunge 0, frameworkul caută *sponsori* înregistrați pentru acest obiect.
- Un sponsor este un obiect de pe server, client sau orice alt calculator din rețea care poate fi apelat folosind Remoting, și care poate decide dacă durata de viață pentru obiectul remote ar trebui reînnoită.
- Dacă sponsorul decide că durata de viață nu ar trebui reînnoită sau nici un sponsor nu poate fi apelat de framework, obiectul remote este marcat ca “expirat” și va fi distrus de garbage collector.
- Dacă un client are referință către un obiect remote expirat și încearcă să apeleze o metodă asupra lui se va arunca o excepție.

Durata de viață

- Pentru a schimba durata de viață, se redefinește metoda InitializeLifetimeService() a clasei MarshalByRefObject.

```
class MyRemoteClass: MarshalByRefObject, IRemoteInterface   {
  public override object InitializeLifetimeService(){
    ILease lease = (ILease)base.InitializeLifetimeService();
    if (lease.CurrentState == LeaseState.Initial){
      lease.InitialLeaseTime = TimeSpan.FromMilliseconds(10M);
      lease.SponsorshipTimeout = TimeSpan.FromMilliseconds(10M);
      lease.RenewOnCallTime = TimeSpan.FromMilliseconds(10H);
    }
        return lease;
 }
      // rest of implementation ...
}
```

Durata de viață

- Un obiect remote Singleton poate avea o durată de viață infinită dacă metoda returnează null.

```
class InfinitelyLivingSingleton: MarshalByRefObject{
   public override object InitializeLifetimeService()    {
      return null;
   }
    // ...
}
```