



Lập trình phân tán theo chủ đề Remote Method Invocation

Cao Tuấn Dũng

Trịnh Tuấn Đạt

Bộ môn CNPM



Nội dung

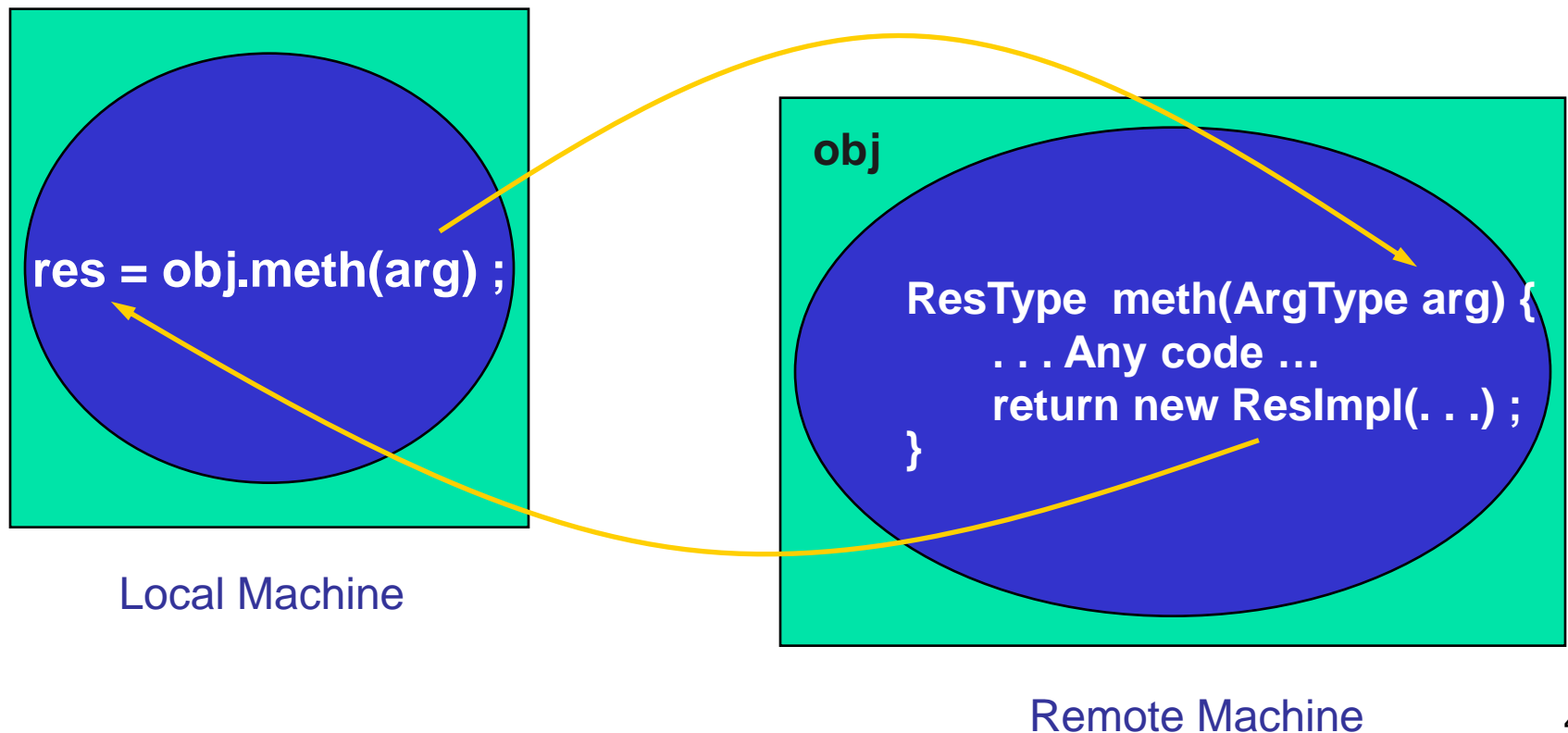
- 1. Giới thiệu chung
- 2. Các thành phần cơ bản
- 3. Các bước xây dựng
- 4. Một số tính năng khác



1. Giới thiệu

Mô hình đối tượng phân tán

- Mã chạy tại máy cục bộ giữ một tham chiếu xa (*remote reference*) tới một đối tượng obj trên máy từ xa.





1.1. RMI là gì

- RPC giữa các Java Objects
- Sự phát triển của RPC
 - ◆ RPC không hướng đối tượng
 - ◆ CORBA (Object-oriented)
 - ◆ RMI (Object-based)
- ◆ Sự cài đặt mô hình đối tượng phân tán trên Java

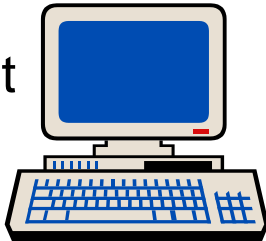


Một số đặc điểm

- RMI cho phép các đối tượng ở 1 JVM thực hiện các phương thức của đối tượng nằm ở 1 JVM khác.
- Hai JVMs có thể nằm ở cùng 1 máy hoặc ở 2 máy khác nhau trong cùng 1 network.
- Đối tượng thực thi 1 phương thức từ xa như thể đối tượng từ xa ở cục bộ (Tức là truy cập qua classpath cục bộ)

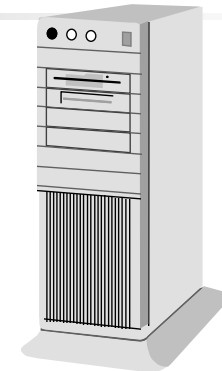
RMI

Client



Internet

Server



Gọi phương thức
cục bộ của Stub

Client
Code

Stub

Trả về giá trị
hoặc exception

Gửi đối số
nhị phân hóa

Gửi kết quả
nhị phân hóa
hoặc exception

Gọi phương thức của
đối tượng từ xa cục bộ

RMI
"Run-time"
System

Remote
Object

Trả về giá trị
hoặc exception



Khác biệt

- Cơ chế hỗ trợ an toàn
- Quản lý ngoại lệ (lỗi do mạng) qua *RemoteException*
- Dựa trên Java
- Giàu ngữ nghĩa hướng đối tượng
 - *object serialization*
 - *dynamic class loading*



1.2. Tại sao sử dụng RMI?

- Cách truyền thống triệu gọi phương thức từ xa:
 - Cách truyền thống là sử dụng Stream Sockets.
 - Stream sockets có thể được sử dụng để truyền các kiểu dữ liệu nguyên thủy (Strings, numbers, ...) hoặc các objects (nếu nó thực thi giao diện Serializable) giữa 2 máy trên mạng.
 - Stream sockets luôn luôn truyền dữ liệu theo giá trị.



1.2. Tại sao sử dụng RMI?

- Giả sử có ứng dụng client/server, mà client muốn cập nhật đối tượng Account trên server.
 - Client sẽ mở kết nối với server và request đối tượng theo tên.
 - Server sẽ gửi 1 bản sao của đối tượng tới client thông qua 1 ObjectOutputStream.
 - Client nhận đối tượng, cập nhật và gửi lại đối tượng đã sửa đổi về server.
 - Server sẽ thay đổi đối tượng của mình bằng đối tượng mới gửi từ client.



1.2. Tại sao sử dụng RMI?

- Nếu nhiều clients cùng yêu cầu cập nhật 1 đối tượng trên server, sẽ có số lượng tương ứng các đối tượng truyền trên network.
 - Mỗi client sẽ nhận bản sao đối tượng từ server truyền về
 - Các sửa đổi từ client cho các đối tượng sẽ chưa có ảnh hưởng chừng nào client chưa gửi lại đối tượng cho server
 - → Làm tăng lưu thông trên mạng.
- Vấn đề khác:
 - Server đổi địa chỉ → client phải biết trước khi truy cập vào 1 đối tượng trên server



1.2. Tại sao sử dụng RMI?

- Nếu client có thể gọi phương thức của 1 đối tượng mà không cần download đối tượng → tăng đáng kể hiệu năng.
- Nếu client có thể truy cập đối tượng từ xa mà không quan tâm ở server nào → đối tượng là “location transparent”.



1.2. Tại sao sử dụng RMI?

- RMI đáp ứng được các yêu cầu trên:
 - RMI: sử dụng các classes & interfaces trong gói `java.rmi` and `java.rmi.server` packages.
 - Cho phép truy cập đối tượng trên server (không dùng giá trị).
 - Khi client có tham chiếu tới đối tượng từ xa, có thể gọi 1 phương thức của đối tượng đó như thể ở cục bộ.
 - Mọi sửa đổi tới đối tượng qua tham chiếu đối tượng từ xa sẽ có tác dụng ngay trên server.
 - Các bản sao đối tượng không bao giờ truyền trên network.



RMI và các công nghệ khác

- CORBA (Common Object Request Broker Architecture)
 - CORBA hỗ trợ chuyển giao đối tượng giữa bất cứ ngôn ngữ ảo nào.
 - Đối tượng được đặc tả trong IDL
 - Phức tạp, dễ mắc lỗi
- RMI thuần túy cho ngôn ngữ Java
 - Đơn giản hơn CORBA

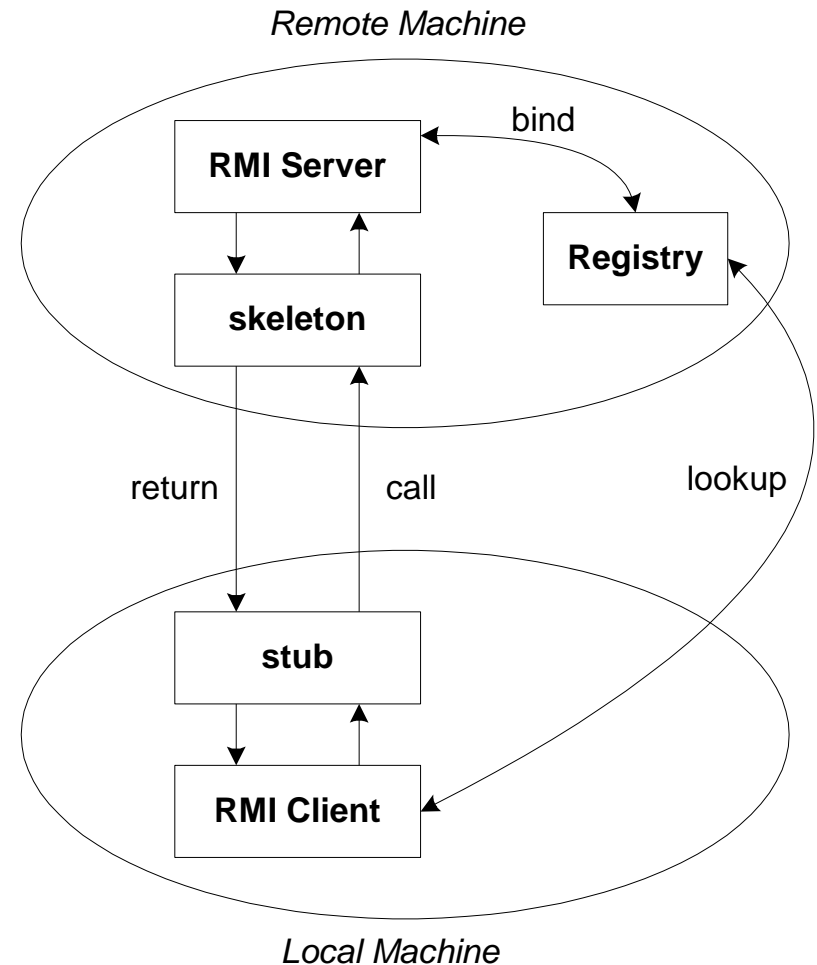


Thuật ngữ

- Đối tượng từ xa (**remote object**): đối tượng nằm ở máy tính từ xa
- Đối tượng client (**client object**): đối tượng yêu cầu – gửi thông điệp tới đối tượng khác
- Đối tượng server (**server object**) đối tượng nhận yêu cầu
- “client” và “server” có thể dễ dàng hoán đổi vai trò
- The **rmiregistry** là server đặc biệt chịu trách nhiệm tìm đối tượng theo tên
- **rmic** là chương trình dịch tạo **stub** (client) và **skeleton** (server) classes

Kiến trúc tổng quát

- Server đăng ký tên (bind) với registry
- Client tìm tên server trên registry để thiết lập tham chiếu từ xa.
- Stub nhị phân hóa tham số cục bộ gửi tới skeleton, Skeleton triệu gọi phương thức từ xa và truyền kết quả lại cho stub.





2. Các thành phần chính của ứng dụng RMI

- Giao diện từ xa - Remote interface
- Stub và Skeleton
- Đối tượng từ xa- Remote object



2.1 Remote Interface

- Giao diện của Java
 - ◆ đặc tả các phương thức có thể được truy nhập từ xa.
- Đối tượng của lớp cài đặt giao diện loại này trở thành **remote object**
- Là "Hợp đồng" giữa RMI client và remote object (RMI server)



The java.rmi.Remote interface

- RMI cung cấp interface Remote, trong gói java.rmi.
 - Nếu một lớp muốn đối tượng của nó có thể truy cập từ xa, lớp đó phải thực thi giao diện Remote.
 - interface Remote: rỗng, không có phương thức nào
 - Thông báo cho Java run time biết đối tượng của lớp tương ứng là truy cập được từ xa.
 - Để tạo 1 lớp có phương thức truy cập được từ xa, phương thức đó phải được khai báo trong 1 interface extends interface Remote .



Sử dụng RMI interface

- Các bước cần làm là:
 1. Khai báo 1 interface extends interface `java.rmi.Remote`
 2. Khai báo các phương thức cần được truy cập từ xa trong interface đó:
 - Tất cả các phương thức này phải ném ra biệt lệ `java.rmi.RemoteException`
 3. Định nghĩa 1 lớp thực thi interface trên.



Extends RMI

- Khai báo 1 interface extends interface `java.rmi.Remote` :

```
public interface MyInterface  
    extends java.rmi.Remote
```



Khai báo các phương thức cần được truy cập từ xa trong interface đó

- public interface:

```
MyInterface extends java.rmi.Remote{  
    //Declare all methods here  
    public String getMessage() throws  
        java.rmi.RemoteException;  
    //More methods  
}
```



Định nghĩa 1 lớp thực thi interface trên

```
Public class MyClass implements MyInterface
{
    //Variable definitions/Declarations
    //Constructors
    //Other methods
    public String getMessage()throws RemoteException
    {
        return new String("Hello there!");
    }
}
```



Ví dụ khác

- MessageWriter.java khai báo interface:

```
import java.rmi.* ;
```

```
public interface MessageWriter extends Remote {  
    void writeMessage(String s) throws RemoteException ;  
}
```

- Phương thức từ xa:
 - **writeMessage()**.



2.2 Stub và Skeleton

- Công cụ (rmic) sinh
 - ◆ RMI stub
 - ◆ RMI skeleton (tùy chọn)
- Từ mã cài đặt trên RMI server (không phải từ RMI interface)
- Được tạo tự động cho các JDK phiên bản sau



Stub và Skeleton

- RMI Stub

- ◆ Nằm ở không gian địa chỉ client
- ◆ Đại diện cho đối tượng từ xa (RO) với client
 - Đóng vai trò **proxy** của remote object
 - **Cài đặt Remote interface**
 - Client gọi đến method của RMI Stub
- ◆ Kết nối với remote object
- ◆ Gửi đối số và nhận kết quả
 - Thực hiện marshaling và unmarshaling



Stub và Skeleton

- RMI Skeleton
 - ◆ Nằm ở không gian địa chỉ server
 - ◆ Nhận đối số từ client và trả kết cho client
 - Thực hiện marshaling và unmarshaling
 - ◆ Xác định phương thức của đối tượng từ xa được gọi
 - ◆ Từ JDK 1.3, RMI Skeleton được sinh tự động



Đối tượng từ xa (Remote Object)

- Cài đặt remote interface
- Cần phải được "xuất khẩu" sang một trong hai kiểu (bằng cách kế thừa):
 - ◆ Non-activatable (extends *java.rmi.server.UnicastRemoteObject*)
 - ◆ Activatable (extends *java.rmi.server.Activatable*)
- ◆ Quy ước: hậu tố **"Impl"**



RMI Server

- RMI Server: 1 Application
 - Tạo các remote objects
 - Tạo các tham chiếu cho các remote objects
 - Đợi Client gọi các phương thức của các remote objects
- RMI Server: Java Class. Phương thức main:
 1. Tạo 1 instance của remote object
 2. Export remote object (tùy chọn)
 3. Kết nối instance ở trên tới 1 tên trong RMI registry



Cách thức export remote object?

- Để export 1 remote object, sử dụng phương thức
`java.rmi.server.UnicastRemoteObject.exportObject`
`(Remote, int);`
- Cho phép nhận request từ client, `exportObject()` có 2 tham số
 - 1 thể hiện của remote object
 - Cổng TCP.
- Cùng 1 cổng có thể nhận nhiều lời gọi từ nhiều remote objects. Cổng mặc định cho RMI là 1099.
 - Nếu tham số vào là 0 → sử dụng cổng mặc định

```
Payment stub = (Payment)  
UnicastRemoteObject.exportObject(robj, 0);
```



Cách thức export remote object? (Cách 2)

- Tạo 1 lớp server extend lớp *java.rmi.server.UnicastRemoteObject*, dùng constructor để export object và định nghĩa số hiệu cổng

```
public class Server extends
    java.rmi.server.UnicastRemoteObject
    implements aRemoteInterface{
    public Server(int port) {
        super(port) ;
    }
    ....
    Naming.bind(uniqueName, this) ;
    ....
}
```



Ví dụ: MessageWriterImpl.java

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

public class MessageWriterImpl extends
UnicastRemoteObject

implements MessageWriter {

    public MessageWriterImpl() throws
RemoteException {
    }

    public void writeMessage(String s)
throws RemoteException {
        System.out.println(s) ;
    }
}
```




Java RMI registry

- Java RMI registry: là một remote object, ánh xạ tên với các remote objects.
- Các phương thức của lớp *LocateRegistry* dùng để lấy 1 registry cho 1 host cụ thể hoặc host và port.
- Phương thức *bind()* và *rebind()* gắn 1 tên/định danh duy nhất với tham chiếu của 1 remote object.



Các bước xây dựng một hệ thống RMI

1. Định nghĩa remote interface
2. Viết các (lớp) remote object bằng cách thực thi giao diện remote interface.
3. Viết Server và Client.
4. Biên dịch mã nguồn Java.
5. Sinh các stub và skeleton.
6. Khởi động RMI registry.
7. Chạy đối tượng Server.
8. Chạy Client.



3. Ví dụ 1

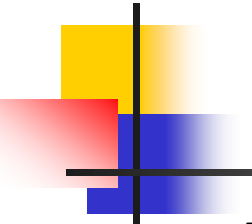
Tính tổng hai số



Bước 1: Định nghĩa Remote Interface

- Giao diện từ xa

```
/* SampleServer.java */  
import java.rmi.*;  
  
public interface SampleServer extends Remote  
{  
    public int sum(int a,int b) throws RemoteException;  
}
```



Bước 2: Viết lớp cho các Remote Object

- kế thừa `java.rmi.server.UnicastRemoteObject`.

```
/* SampleServerImpl.java */
```

```
import java.rmi.*;
```

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

```
public class SampleServerImpl extends
```

```
    UnicastRemoteObject
```

```
    implements SampleServer
```

```
{
```

```
    SampleServerImpl() throws RemoteException {
```

```
        super();
```

```
    }
```

```
    public int sum(int a,int b) throws RemoteException {
```

```
        return a + b;
```

```
    }
```

```
}
```



Bước 3: Viết Server

```
import java.rmi.* ;

public class SumServer {
    public static void main(String args[]) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            SampleServerImpl Server = new SampleServerImpl();
            Naming.rebind("SAMPLE-SERVER" , Server);
            System.out.println("Server waiting.....");
        }
        catch (java.net.MalformedURLException me) {
            System.out.println("Malformed URL: " + me.toString());
        }
        catch (RemoteException re) {
            System.out.println("Remote exception: " + re.toString());
        }
    }
}
```



Bước 3: Viết client

- Tìm kiếm tên của server trong registry. Sử dụng lớp `java.rmi.Naming`.
- Tên server ở dạng URL (`rmi://host:port/name`)
- Cổng RMI mặc định: 1099.
- Tên trong URL phải khớp chính xác tên mà server đăng ký trong registry.
 - "SAMPLE-SERVER"



Bước 3: Viết client

```
import java.rmi.*;
import java.rmi.server.*;
public class SampleClient {
    public static void main(String[] args) {
        // set the security manager for the client
        System.setSecurityManager(new RMISecurityManager());
        try {
            System.out.println("Security Manager loaded");
            String url = "://localhost/SAMPLE-SERVER";
            SampleServer remoteObject = (SampleServer)Naming.lookup(url);
            System.out.println("Got remote object");
            System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );
        }
        catch (RemoteException exc) {
            System.out.println("Error in lookup: " + exc.toString()); }
        catch (java.net.MalformedURLException exc) {
            System.out.println("Malformed URL: " + exc.toString()); }
        catch (java.rmi.NotBoundException exc) {
            System.out.println("NotBound: " + exc.toString());
        }
    }
}
```




Bước 4 và 5: Biên dịch - Sinh stubs và skeletons

```
javac SampleServer.java
```

```
javac SampleServerImpl.java
```

```
javac SumServer.java
```

```
rmic SampleServerImpl
```

```
javac SampleClient.java
```



Bước 6: Chạy RMI registry

- Chạy `rmiregistry`. Hoặc `start rmiregistry`
- Chỉ định cổng khác mặc định:
`rmiregistry <new port>`



Bước 7 và 8: Chạy Server và Client

- Khi registry đã chạy, có thể chạy server.
- Thiết lập security (tùy phiên bản Java)

```
java -D java.security.policy=policy.all  
SampleServerImpl
```

```
java -Djava.security.policy=policy.all SampleClient
```



3. Ví dụ 2

Tính phí thể chấp



3.1. Kịch bản sử dụng

- Betty mới nhận việc trong 1 công ty cho vay.
 - Quản lý Joy yêu cầu cô public phương thức tính thể chấp.
 - Tức là phương thức tính thể chấp sẽ được cung cấp trên intranet hoặc internet.
 - Tính toán thể chấp là 1 phần mềm đã có sẵn, được cài đặt trên 1 máy đơn lẻ.
 - Nếu được cài đặt trên server, sẽ dễ dàng bảo trì và cập nhật hơn.
 - Đây là nhiệm vụ đầu của Betty. Rất háo hức, cô chọn công nghệ Java RMI.



3.2. Phương thức `calculatePayment()`

- Betty bắt đầu xem lại lớp tính toán Thẻ chấp, trong đó quan trọng là phương thức *`calculatePayment()`*:

```
public double calculatePayment(double principal, double
    annualRate, int years){
    double monthlyInterest = annualRate / 12;
    double monthlyPayment = (principal *
    monthlyInterest)
        / (1 - Math.pow(1/ (1 + monthlyInterest),
    years * 12));
    return monthlyPayment;
}
```



3.2. Phương thức calculatePayment()

- Để tính phí cần trả hàng tháng cho 1 khoản vay, cần 3 dữ liệu:
 - Tiền gốc-principal, lãi suất-annual rate và thời hạn-term.
 - Khi người dùng cung cấp đủ dữ liệu, sẽ trả về số tiền phải trả hàng tháng.



3.2. Phương thức calculatePayment()

```
import java.rmi.Remote;  
import java.rmi.RemoteException;  
  
public interface Payment extends Remote {  
    public double calculatePayment(double  
        principal, double annualRate, int terms)  
        throws RemoteException;  
}
```




3.3. Thiết kế Remote Object

- RMI: *remote object là đối tượng mà phương thức của nó có thể được gọi từ JVM khác JVM quản lý nó*



3.3. Thiết kế Remote Object

```
import java.rmi.RemoteException;

public class PaymentImpl implements Payment {

    public double calculatePayment(double principal, double annRate,
int years)
                                throws RemoteException {
        double monthlyInt = annRate / 12;
        double monthlyPayment = (principal * monthlyInt)
                                / (1 - Math.pow(1/ (1 + monthlyInt), years *
12));
        return format(monthlyPayment, 2);
    }
    public double format(double amount, int places) {
        double temp = amount;
        temp = temp * Math.pow(10, places);
        temp = Math.round(temp);
        temp = temp/Math.pow(10, places);
        return temp;
    }
}
```



Java RMI registry

```
java.rmi.registry.Registry registry =  
    java.rmi.registry.LocateRegistry.getRegistry();  
registry.bind("Mortgage", stub);  
hoặc:  
registry.rebind("Mortgage", stub);
```



Server Class

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server {

    public Server() {}

    public static void main(String args[]) {

        try {
            PaymentImpl robj = new PaymentImpl();
            Payment stub = (Payment) UnicastRemoteObject.exportObject(robj, 0);

            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Mortgage", stub);
            System.out.println("Mortgage Server is ready to listen...");

        } catch (Exception e) {
            System.err.println("Server exception thrown: " + e.toString());
            e.printStackTrace();
        }
    }
}
```



3.5. RMI Client

- RMI client: 1 ứng dụng:
 - Lấy ra 1 tham chiếu từ xa tới 1 hoặc nhiều remote objects trên server
 - Gọi 1 phương thức của các remote objects đó.
- RMI client: Java Class
 1. Xác định host, dùng cơ chế registry.
 2. Tra cứu remote object theo tên của nó
- Khi Client lấy được stub Class từ RMI Server, sẽ gọi remote method từ stub đó



3.5. RMI Client

- Xác định host: sử dụng lớp LocateRegistry.

```
Registry reg =
```

```
    LocateRegistry.getRegistry(hostName) ;
```

- Tra cứu remote object:

- Phải biết tên của remote object
- Ví dụ để lấy về 1 stub Class của object Mortgage:

```
Payment stub =
```

```
    (Payment) reg.lookup("Mortgage") ;
```



Client Class

- Sử dụng stub lấy về được từ phương thức *lookup()* để gọi *calculatePayment()* và truyền vào 3 tham số.
- Sử dụng try/catch xử lý lỗi.



Client class

```
try {  
    Registry reg = LocateRegistry.getRegistry("localhost");  
    stub = (Payment) reg.lookup("Mortgage");  
  
} catch (Exception e) {  
    System.err.println("Client exception thrown: " + e.toString());  
    e.printStackTrace();  
}  
.....  
  
public static void print(double pr, double annRate, int years){  
    double mpayment = 0;  
    try {  
        mpayment = stub.calculatePayment(pr, annRate, years);  
    } catch (Exception e) {  
        System.out.println("Remote method exception thrown: " + e.getMessage());  
    }  
    System.out.println("The principal is $" + (int)pr);  
    System.out.println("The annual interest rate is " + annRate*100 + "%");  
    System.out.println("The term is " + years + " years");  
    System.out.println("Your monthly payment is $" + mpayment);  
}
```




3.6. Test ứng dụng

- Server Stub Class:
 - Từ jdk 1.5.0:
 - Không cần sinh stub Class trước khi start server
 - Đã được sinh tự động
 - Bản cũ:
 - Sử dụng rmic (của jdk) để sinh 1 stub class
 - Được file Server_stub.class
- Run server:
 - Chạy rmiregistry (của jdk)
 - Chạy lớp trên server



Chạy Server

- Chạy lớp Server:

`C:\myrmi\start java Server`

- Nếu OK, thông báo nhận được:

`Mortgage server is ready to listen...`

- Chú ý RMI server có thể nhận đồng thời nhiều kết nối từ client



Chạy Client

- Khi server đã chạy, có thể start client:

- `C:\myrmi\java Client`

- Khi client start, lệnh

`LocateRegistry.getRegistry("localhost") :`

- Gửi tới RMI registry trên server (localhost), tìm đối tượng tên là "Mortgage".
 - Server trả về instance của stub class tương ứng
 - Client gọi `calculatePayment()` của tham chiếu tới remote object như gọi từ cục bộ



Tổng hợp lệnh

- Tóm tắt các lệnh đã thực hiện trong hệ thống client/server RMI:

```
C:\myrmi>javac -d . Payment.java  
PaymentImpl.java Server.java Client.java
```

```
C:\myrmi>set classpath=
```

```
C:\myrmi>start rmiregistry
```

```
C:\myrmi>start java Server
```



Tổng hợp lệnh

```
C:\myrmi>java Client
```

```
Usage: java Client principal annualInterest years
```

```
For example: java Client 80000 .065 15
```

You will get the output like the following:

```
The principal is $80000
```

```
The annual interest rate is 6.5%
```

```
The term is 15 years
```

```
Your monthly payment is $696.89
```



Tổng hợp lệnh

```
C:\myrmi>java Client 150000 .060 15
```

```
The principal is $150000
```

```
The annual interest rate is 6.0%
```

```
The term is 15 years
```

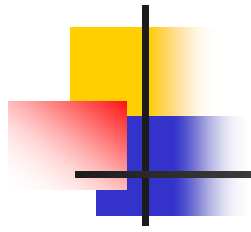
```
Your monthly payment is $1265.79
```

```
C:\myrmi>
```



4. Tổng kết

- Các bước thiết kế RMI client/server
 1. Design a remote interface.
 2. Design a remote object.
 3. Design an RMI server.
 4. Design an RMI client.



4. Tải lớp động



Mã lệnh Byte Code cho Stubs?

- Trước khi sử dụng đối tượng từ xa RMI, client phải nhận được **serialized stub object**.
- Đối tượng này chứa **remote reference**. Các trường dữ liệu của tham chiếu có thể gồm:
 - Tên host của server,
 - Các cổng
 - Các thông tin cần thiết để xác định duy nhất một remote object trên server.
- Các đối tượng này không chứa **các lệnh máy ảo** (mã byte codes), cài đặt các phương thức của đối tượng cục bộ.



Sao chép các lớp Stub

- Hai cách mã class Stub → client.
- Sao chép thẳng các lớp **stub** tới máy client: Đặt trong thư mục chạy ứng dụng client hoặc trong **CLASSPATH**.
 - Hạn chế : Client code không nên hạn chế vào implementation của giao diện
- Cơ chế Tải lớp động (Dynamic class loading)

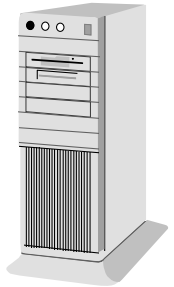
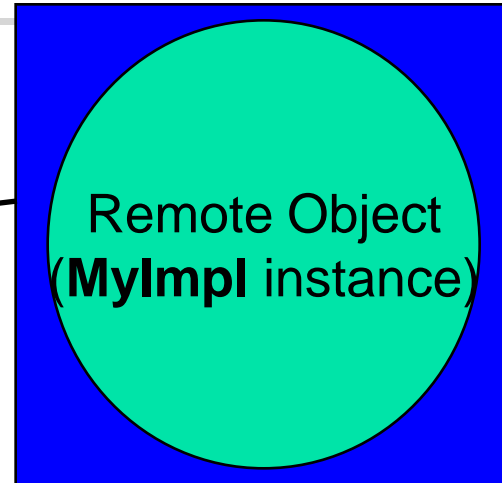
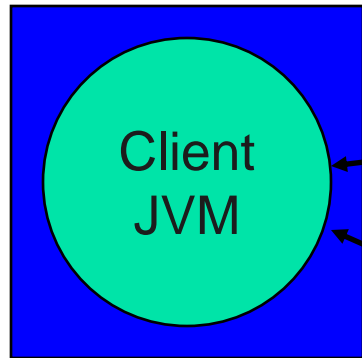


Dynamic Class Loading

- Đăng tải mã thực thi cần thiết cho client trên một Web Server.
- Thông tin về vị trí các mã này có thể được ghi chú (*annotated* với URL) và truyền cho client.
- JVM của client sẽ tự download các mã thực thi từ Web Server chỉ định trong ghi chú.

Dynamic Class Loading

Mã stub được Serialized,
ghi chú với code-base:
<http://myWWW/download/>

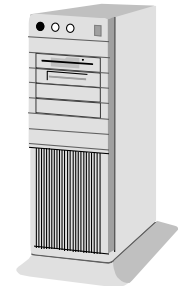
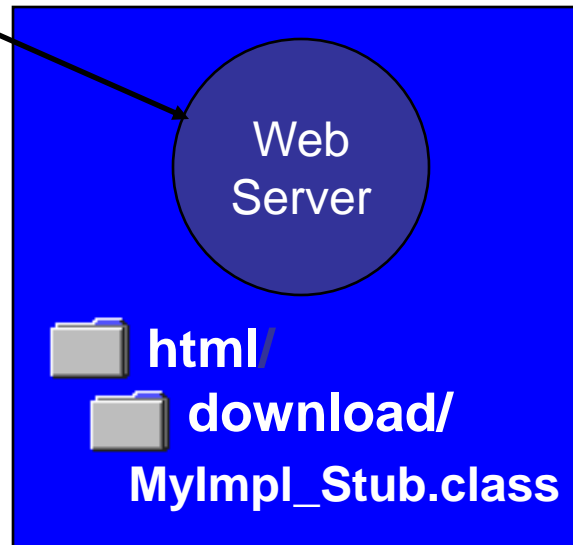


Server



Client

Yêu cầu các
file class stub



Server
(myWWW)



Thuộc tính `java.rmi.server.codebase`

- Chú thích đối tượng nhị phân hóa với các URL.
- Thiết lập thuộc tính `java.rmi.server.codebase` tại máy ảo nơi sinh ra stub.
- Giá trị của thuộc tính: `code-base URL`.



Thiết lập Code-base

- Từ phía server **HelloServer** :

```
java -Djava.rmi.server.codebase=http://soict.hut.edu.vn/dungct/dsd1/ HelloServer
```

- Khi một đối tượng được tạo lại sau đó ở một máy ảo Java khác (mà không thể tìm thấy bản sao của các lớp cần thiết) nó sẽ tự động tìm kiếm ở thư mục **dungct/dsd1/** của host chỉ định.



Quản lý an ninh

- Không an toàn nếu client khi sử dụng mã nguồn của ai đó trên server ngẫu nhiên.
- Nếu cơ chế quản lý an ninh bị tắt, các lớp stubs được nạp từ **CLASSPATH** cục bộ.
- Thiết lập an ninh qua SecurityManager
 - `System.setSecurityManager(new RMISecurityManager());`
 - Chính sách an ninh của `RMISecurityManager` giống `SecurityManager`
 - Phía client cần thiết lập chi tiết hơn.



Thiết lập Security Manager

- Thuộc tính **java.security.policy** : phía client
 - **java.rmi.server.codebase**: phía server
- Ví dụ:

```
grant { permission java.security.AllPermission "", "" ; } ;
```




Sử dụng thuộc tính

- Giả sử tập tin ghi chính sách an ninh là **policy.all**, chạy client (ví dụ **HelloClient**) như sau:

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

- Hoặc sử dụng **System.setProperty()**.