Java RMI

Sistemas Distribuídos 2013/2014

Programação orientada a objectos

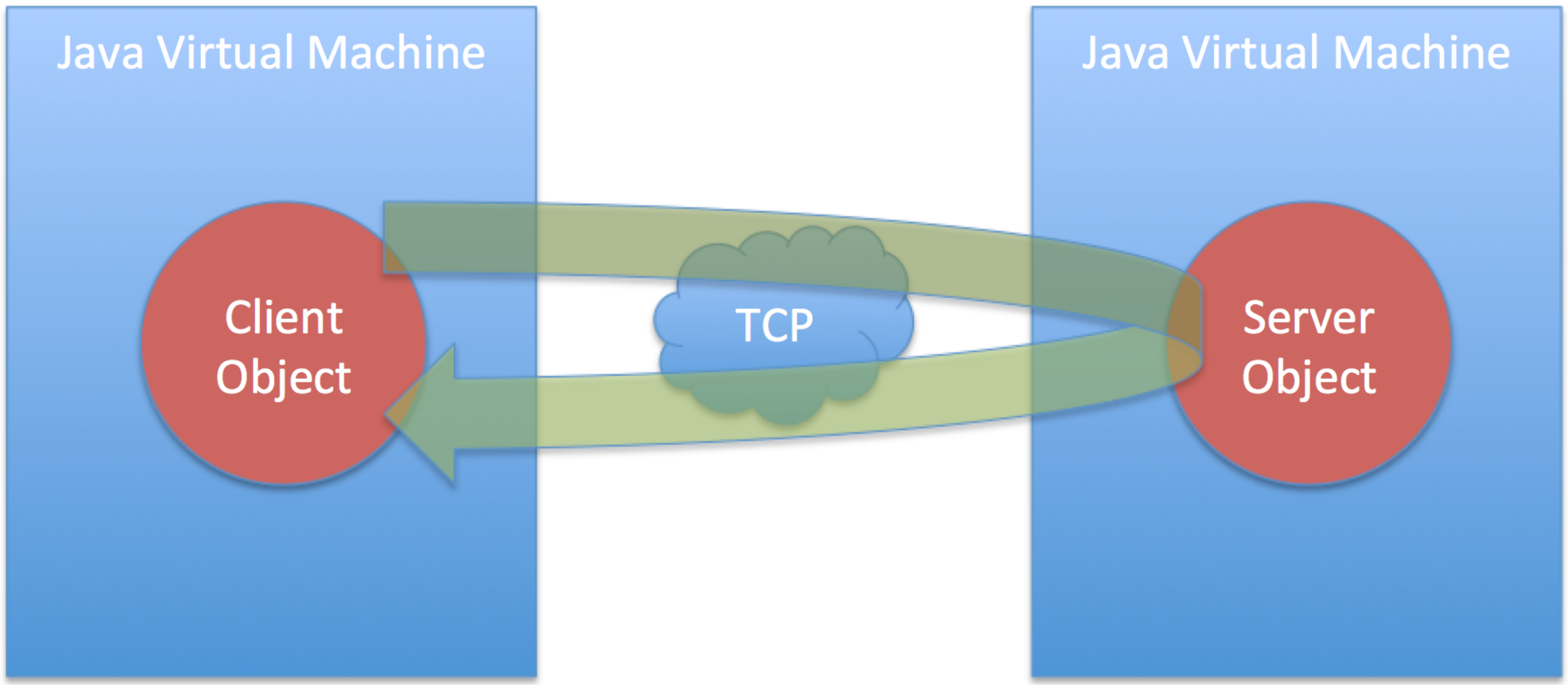
- Um programa orientado a objectos (*e.g.*, Java, C++) consiste numa colecção de objectos que interagem entre si.
- Cada objecto "comunica" com outros objectos, chamando os seus métodos, passando argumentos e recebendo resultados.
- Num sistema de objectos distribuídos, é possível chamar métodos de objectos remotos.

Programação orientada a objectos

```
Date latada = eventos.getDataDeInicio("Latada 2013");
```

- O objecto `latada` existe na máquina cliente.
- O objecto `eventos` reside num servidor.
- Ao chamar `getDataDeInicio()`, o sistema encarrega-se de obter os dados no servidor.

O que pretendemos ter



RMI & RPC

- Remote Procedure Call (RPC).
- Invocação de métodos (RMI).
- Both RMI and RPC provide a programming model similar to centralized programs.
- RMI is similar to RPC but extended into the world of distributed objects.

Object model

- **Object references** → **Remote object references**. Objects can be accessed via object references. In Java, a variable that appears to hold an object actually holds a reference to that object (which is now remote).
- **Interfaces** → **Remote interfaces**. An interface provides a definition of the signatures of a set of methods without implementing them.
- **Methods** → **Remote methods**. The receiver executes the appropriate method and then returns control to the invoking object, sometimes supplying a result.
- **Exceptions** → **Remote exceptions**.
- **Garbage collection** → **Distributed garbage collection**.

RMI Definitions

- Remote object:
  - An object whose methods can be invoked from another Java virtual machine, potentially on a different host.
- Remote interfaces:
  - Interfaces written in Java that declare the methods of the remote object

Remote references:

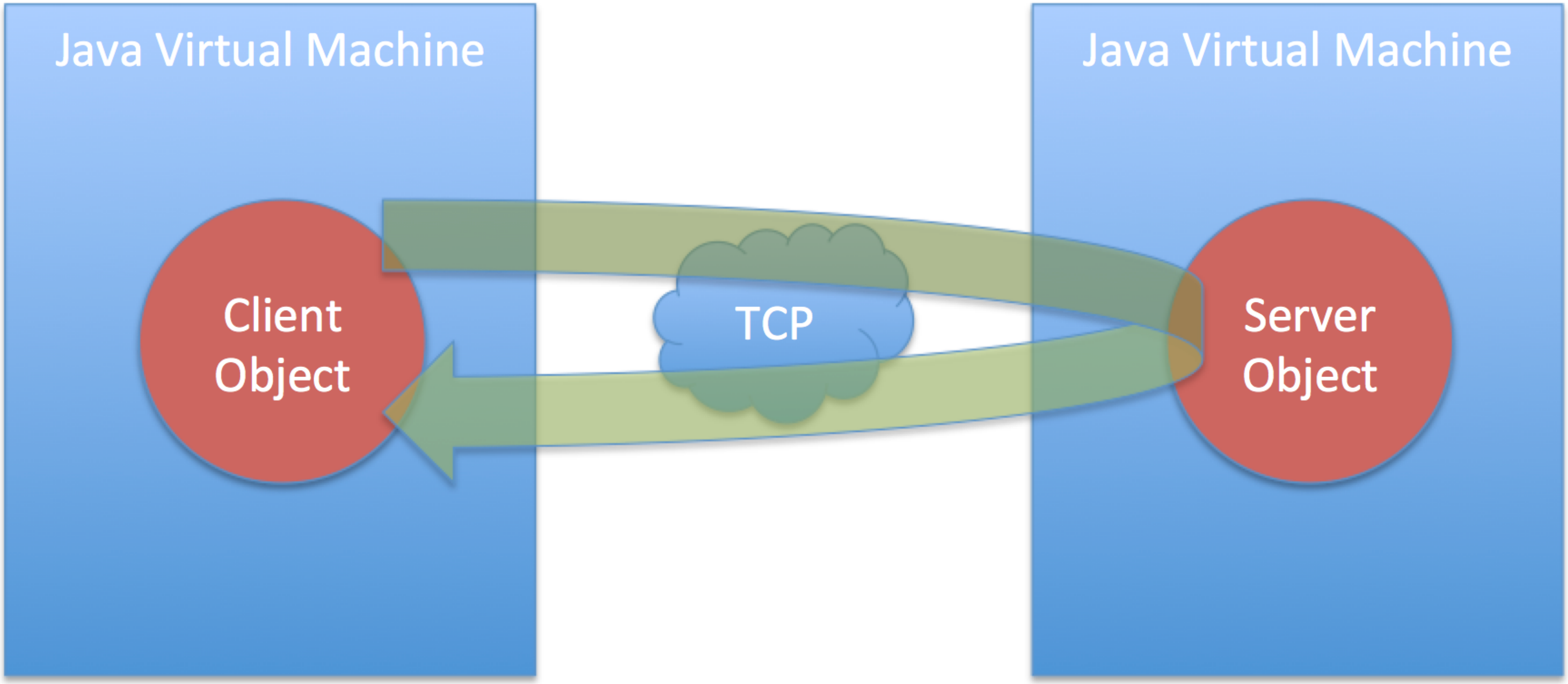
- Refer to remote objects.
- Invoked in the client exactly like local object references.

Invocation semantics

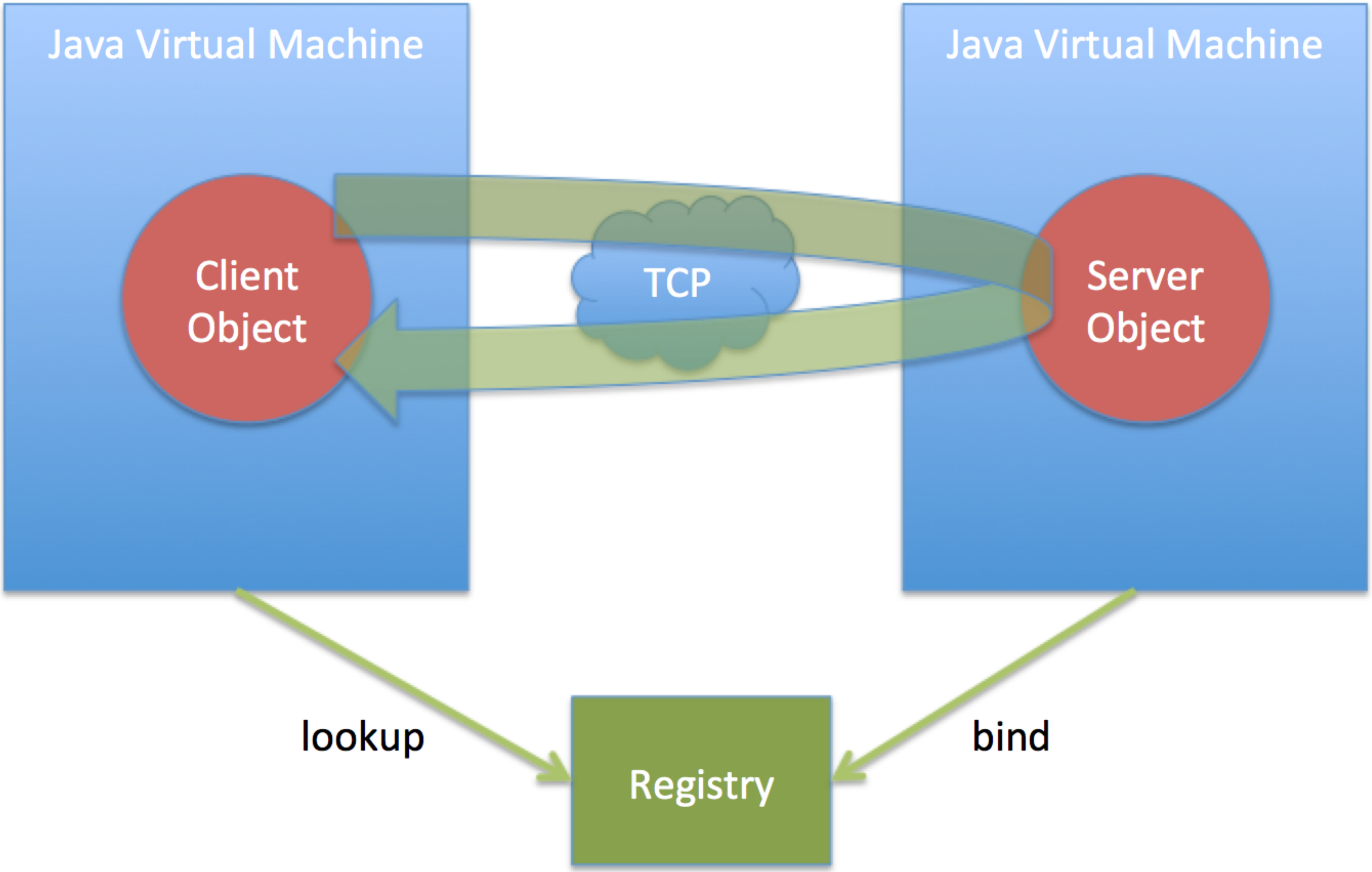
Fault tolerance measures			Invocation semantics
Retransmit request	Filter duplicates	Re-execute procedure or retransmit reply	
No	Not applicable	Not applicable	Maybe
Yes	No	Re-execute procedure	At-least-once
Yes	Yes	Retransmit reply	At-most-once

- Java RMI and CORBA provide “at-most-once” semantics.
- CORBA also allows “maybe” semantics.
- Sun RPC (ONC RPC) provides “at-least-once”.

How does the client locate the remote object?



How does the client locate the remote object?



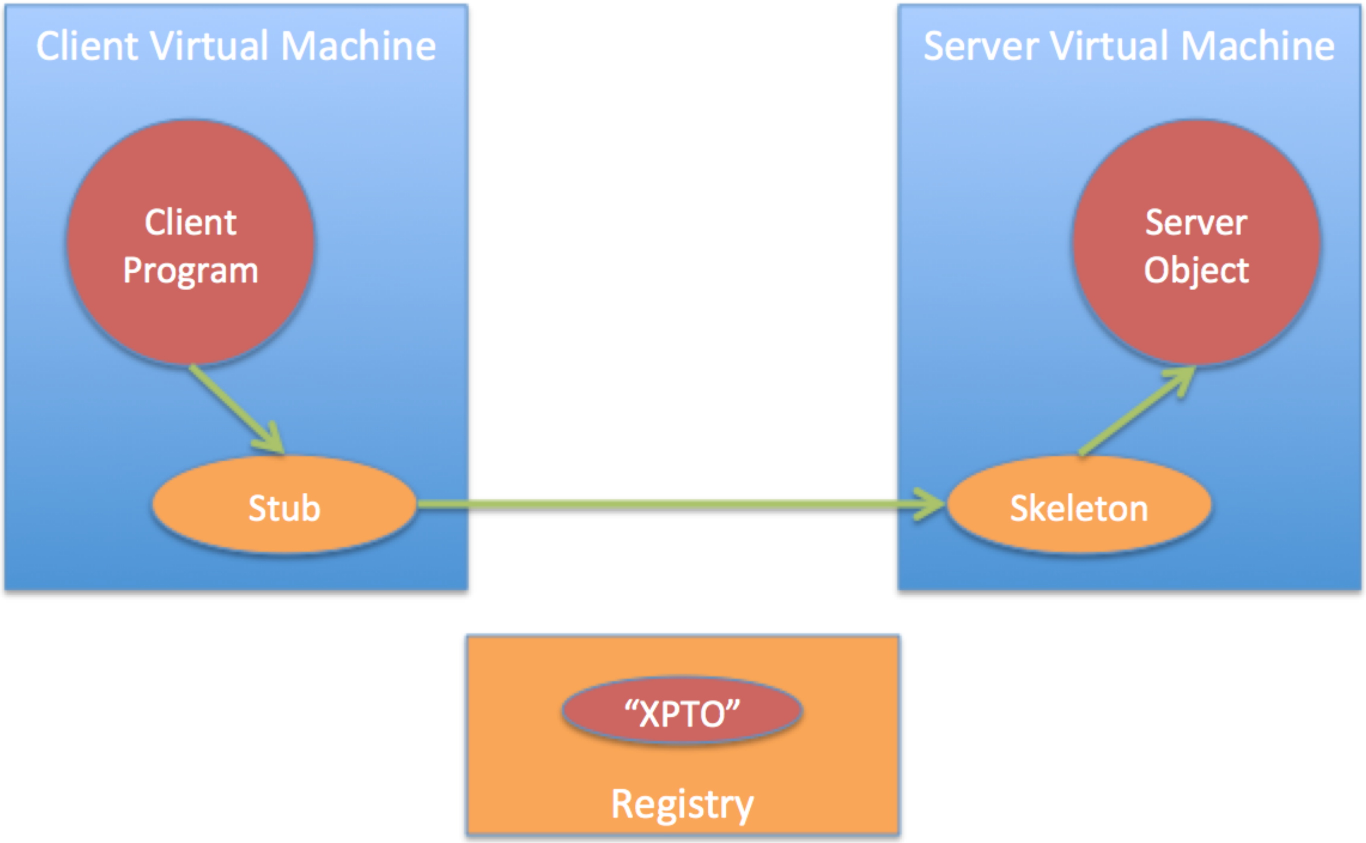
The registry

- Register and lookup remote objects
- Servers can register their objects
- Clients can find server objects and obtain a remote reference
- A registry is a process running on a host machine
- Java RMI – RMI Registry
- Sun RPC – Portmapper
- CORBA – Naming service

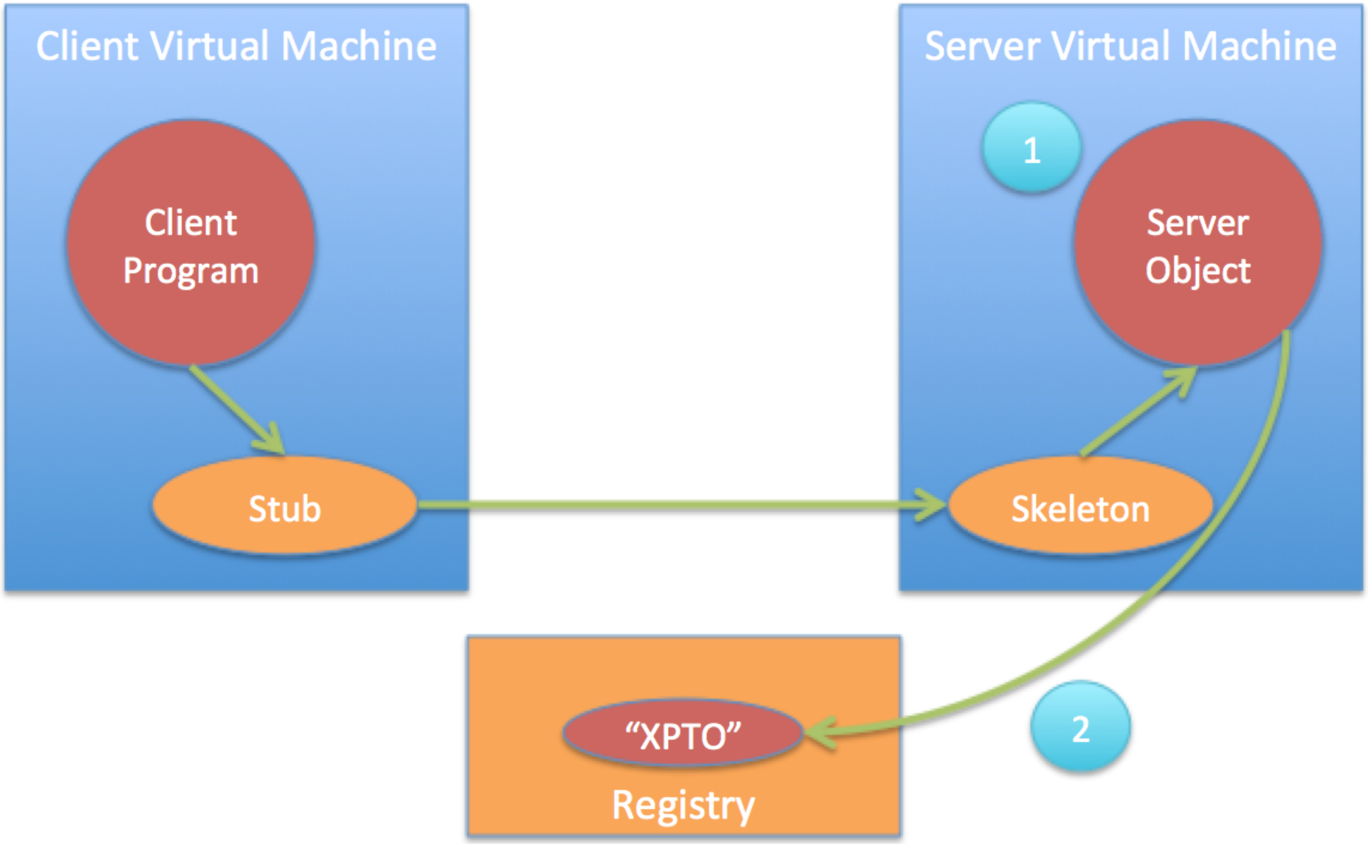
Components of the RMI architecture

- *Server object interface.* An interface of `java.rmi.Remote` which specifies the methods of the server.
- *Server class.* A class that implements the remote interface.
- *Server object.* A server class instance.
- *RMI registry.* A naming service that registers remote objects and allows remote objects to be located by name.
- *Client program.* A program that wants to invoke remote methods on the server object.
- *Server stub.* An object on the client host that serves as a stub for the remote object.
- *Server skeleton.* An object on the server host that interacts with the server stub and with the server object.

RMI System Architecture

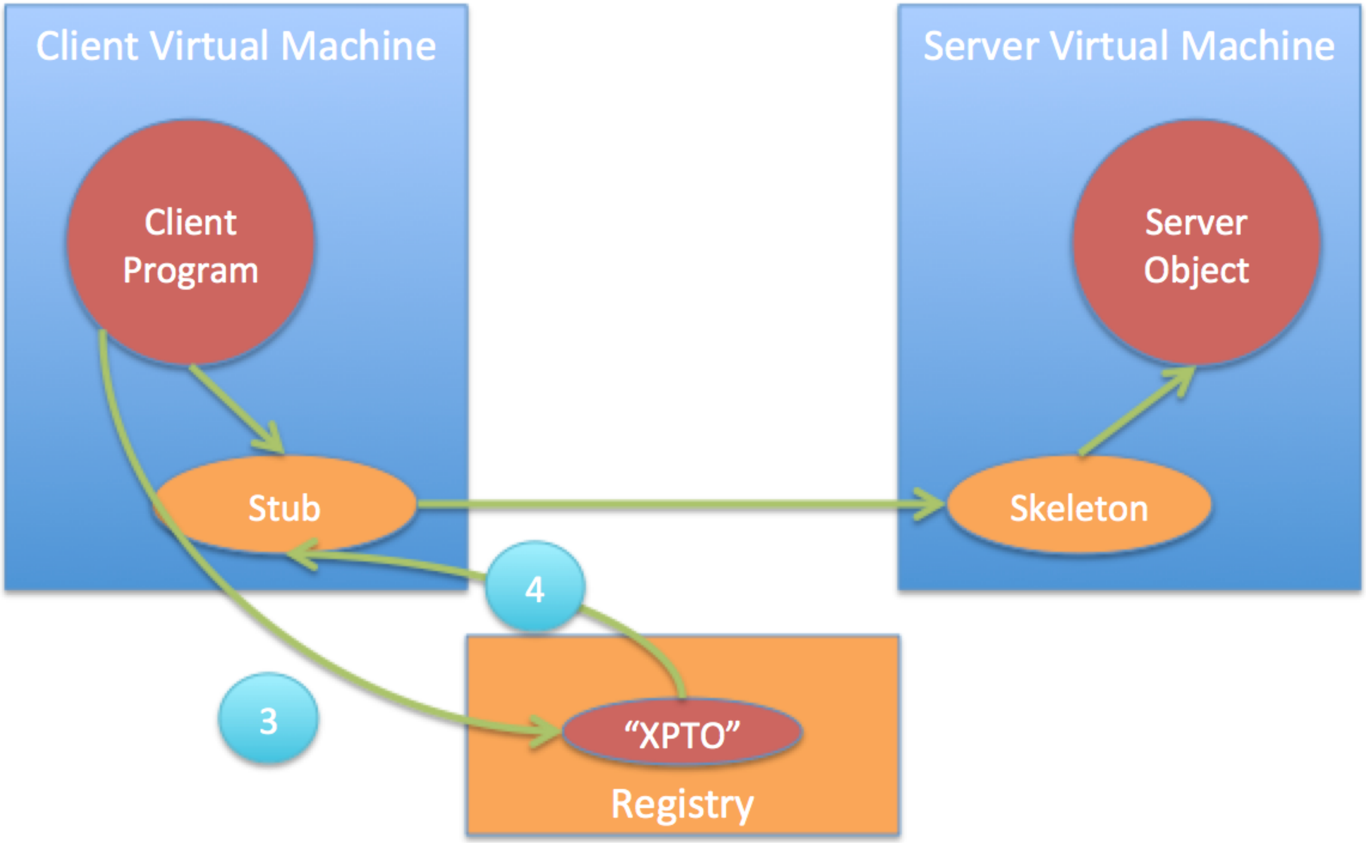


RMI Flow



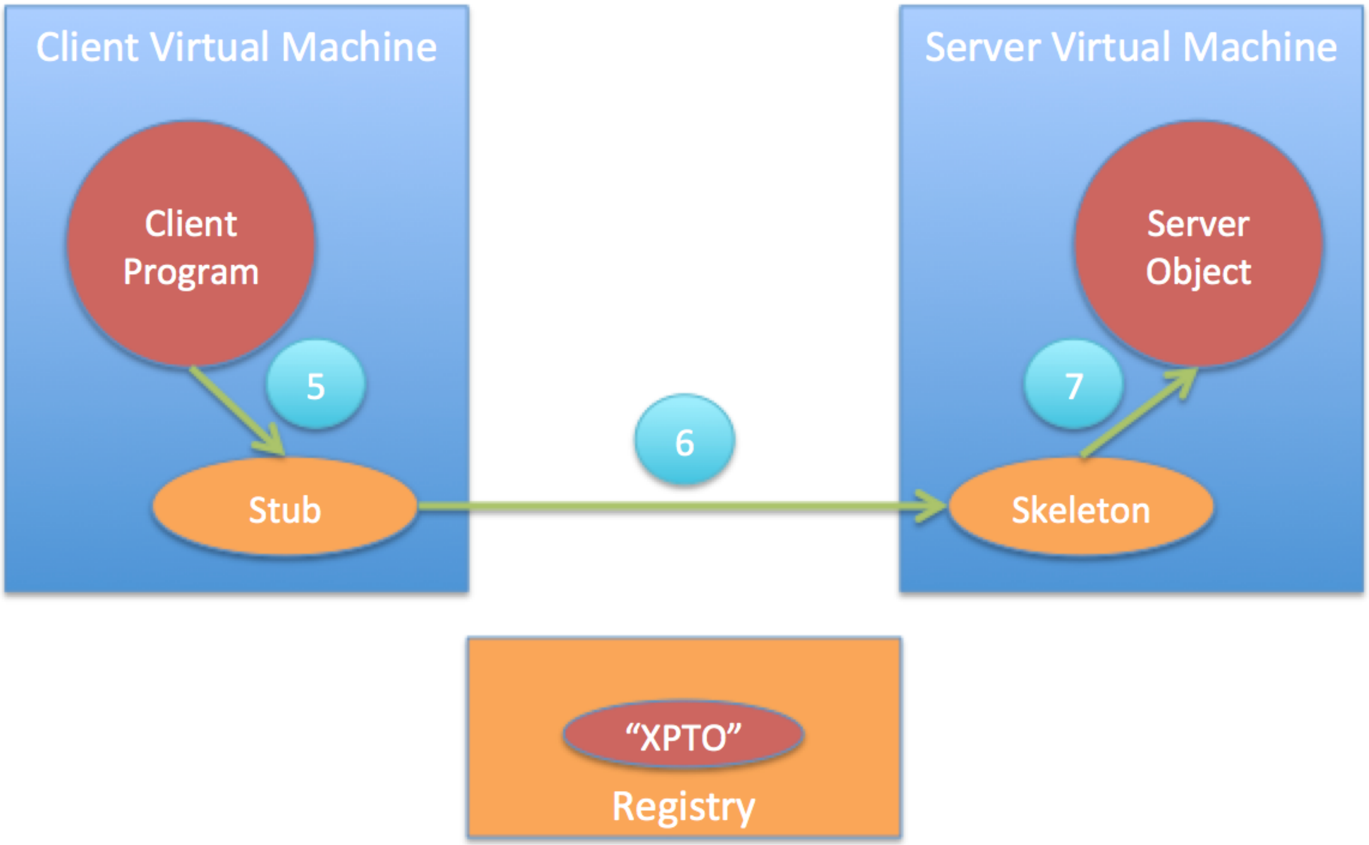
1. Server creates Remote Object
2. Server registers Remote Object

RMI Flow



3. Client requests object from Registry
4. Registry returns remote reference (and stub gets created)

RMI Flow



5. Client invokes stub method
6. Stub talks to skeleton
7. Skeleton invokes remote object method

RMI Advantages

- RMI provides a very clean API
    - Access to remote objects
    - Java-to-Java only
    - Client-server protocol
    - High-level API
    - Transparent
    - Lightweight
- Neither client nor server handle anything explicitly with input streams, output streams, or sockets.
- Complex Java objects can be sent back and forth, but no parsing is required at either end (serialization).

Java RMI Programming

1- Build a Java RMI object

1. You define your remote object interface in a normal Java interface. The interface must extend `java.rmi.Remote`
   - All the methods must throw `java.rmi.RemoteException`

2. Your real remote object implementation must extend from `java.rmi.server.UnicastRemoteObject` and implement the interface specified in 1.
   - Note: everything that travels through the network must be *serializable*, i.e. implement `java.io.Serializable`. This includes any classes that are used as parameters.

3. Create an object and bind it to the **RMI Registry**.

First Example: Math Server

```
public interface MathServer  extends java.rmi.Remote
{
  public int add(int a, int b) throws java.rmi.RemoteException;
  public int mult(int a, int b) throws java.rmi.RemoteException;
}
```

Math Server

```
public class MathServerImpl
  extends java.rmi.server.UnicastRemoteObject
  implements MathServer
{
  public MathServerImpl() throws java.rmi.RemoteException {
    // Must have a constructor and throw RemoteException
  }

  public int add(int a, int b) throws java.rmi.RemoteException {
    return a+b;
  }

  public int mult(int a, int b) throws java.rmi.RemoteException {
    return a*b;
  }
}
```

Math Server

```
public class Server
{
  public static void main(String[] args)
  {
    System.getProperties().put("java.security.policy", "security.policy");
    System.setSecurityManager(new RMISecurityManager());

    try {
      MathServerImpl myServer = new MathServerImpl();
      Naming.rebind(“calculadora”, myServer);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Using a remote object

```
public class Client
{
 public static void main(String[] args)
 {
  System.getProperties().put("java.security.policy", "security.policy");
  System.setSecurityManager(new RMISecurityManager());

  try {
   MathServer myServer =
    (MathServer) Naming.lookup(“rmi://localhost/calculadora”);

   int result = myServer.add(2, 3);
   System.out.println(result);
  }
  catch (RemoteException e) {
   e.printStackTrace();
  }
 }
}
```

Compiling and running

1. Compile it

```
javac *.java
```

2. Generate the stubs and skeletons for your remote objects

```
rmic MathServerImpl
// NOT NECESSARY FOR JAVA 1.5
// JAVAC does RMIC for you
```

3. Setup a policy file (`security.policy`)

```
grant codeBase "file:/./-"
{
  permission java.security.AllPermission;
};
```

RMIC (older versions of Java) Generating Stubs + Skeletons

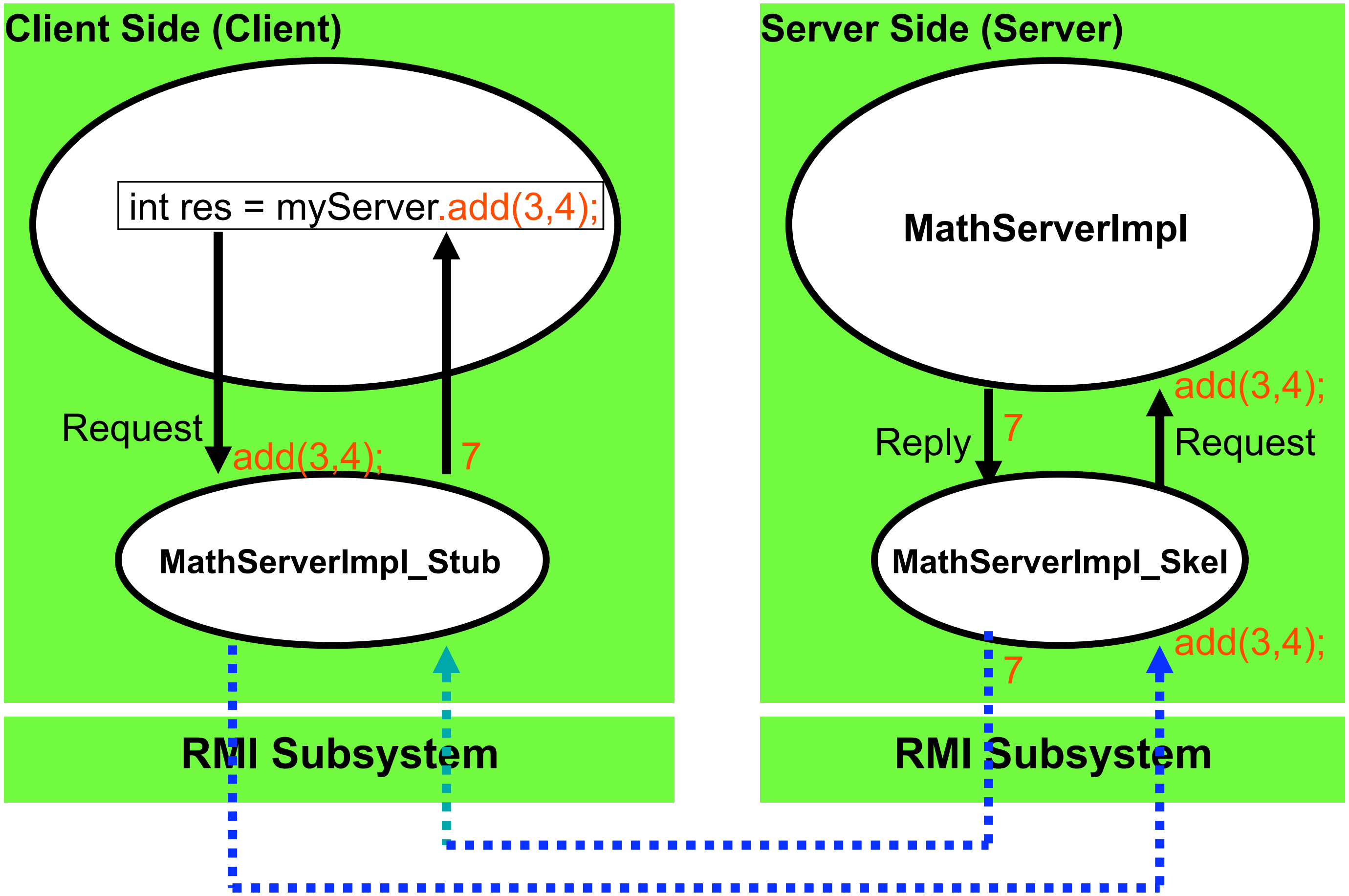
- Stubs and skeletons are generated by calling the RMI compiler **rmic** on the server implementation class.

  - `C:\> rmic MathServerImpl`

- This generates two class files:
  - `MathServerImpl_Skel.class`
    - server skeleton class
  - `MathServerImpl_Stub.class`
    - client stub class

Stubs and skeletons

```
MathServer myServer = (MathServer) Naming.lookup(“rmi://localhost/mathServer);
```



Client Side (Client)

int res = myServer.add(3,4);

Request

add(3,4);

7

MathServerImpl_Stub

RMI Subsystem

Server Side (Server)

MathServerImpl

add(3,4);

Reply

7

Request

MathServerImpl_Skel

7

add(3,4);

RMI Subsystem

Executing the Application

- For running the server
  - Initiate Java’s Registry service
    - start rmiregistry
  - Run the server
    - java Server
- For running the client
    - java Client

Note: Make sure that the client has access to the MathServerImpl_Stub.class file!

Bootstrap: how to identify the remote object?

- The name of a remote object includes the following information
  - The Internet address of the machine that is running the RMI Registry, where the remote object is being registered
  - The port to which the RMI Registry is listening (the default port is 1099)
  - The local name of the remote object.

```
rmi://myserver.com/calculator
```

The Registry (2)

- How to start rmiregistry at a given port:
  - `LocateRegistry.createRegistry(PORT);`

- Important methods of Registry
  - // Returns an array of the names bound in this registry
    `String[] list();`
  - // Returns the reference bound to the specified name
    `Remote lookup(String objectName);`
  - // Binds the name to a remote object
    `void bind(String objectName, Remote object);`
  - // Replaces the binding for the specified name
    `void rebind(String objectName, Remote object);`
  - // Removes a reference from the registry
    `void unbind(String objectName);`

Remote References

- **Naming.rebind();**
  - Estamos a passar uma referência do objecto para a classe Naming.
  - A classe Naming constroi um objecto do stub e faz o bind deste stub no objecto remoto do REGISTRY.

- **Naming.lookup();**
  - O REGISTRY devolve o stub ao cliente.
  - O stub sabe qual é o hostname e porto onde o servidor está à escuta de um socket.
  - O cliente pode invocar o método do stub para executar o método do objecto remoto.

Security in RMI

- If no SecurityManager is specified, no dynamic code downloading can take place.

- Typically, the RMISecurityManager is used:

```
System.getProperties().put("java.security.policy","security.policy");
System.setSecurityManager(new RMISecurityManager());
```

- You must specify a policy file
  - `security.policy`

Examples of policy files

```
// Grants all the code, even if it is downloaded,
  permissions for connect,
// accept and resolve sockets…
grant {
   permission java.net.SocketPermission "*:1024-65535",
  "connect,accept,resolve";
   permission java.net.SocketPermission "*:80", "connect";
};
```

```
// Grants all the code, in the current directory,
// permissions for doing everything
grant codeBase "file:/./-"
{
  permission java.security.AllPermission;
};
```

Parameter Passing

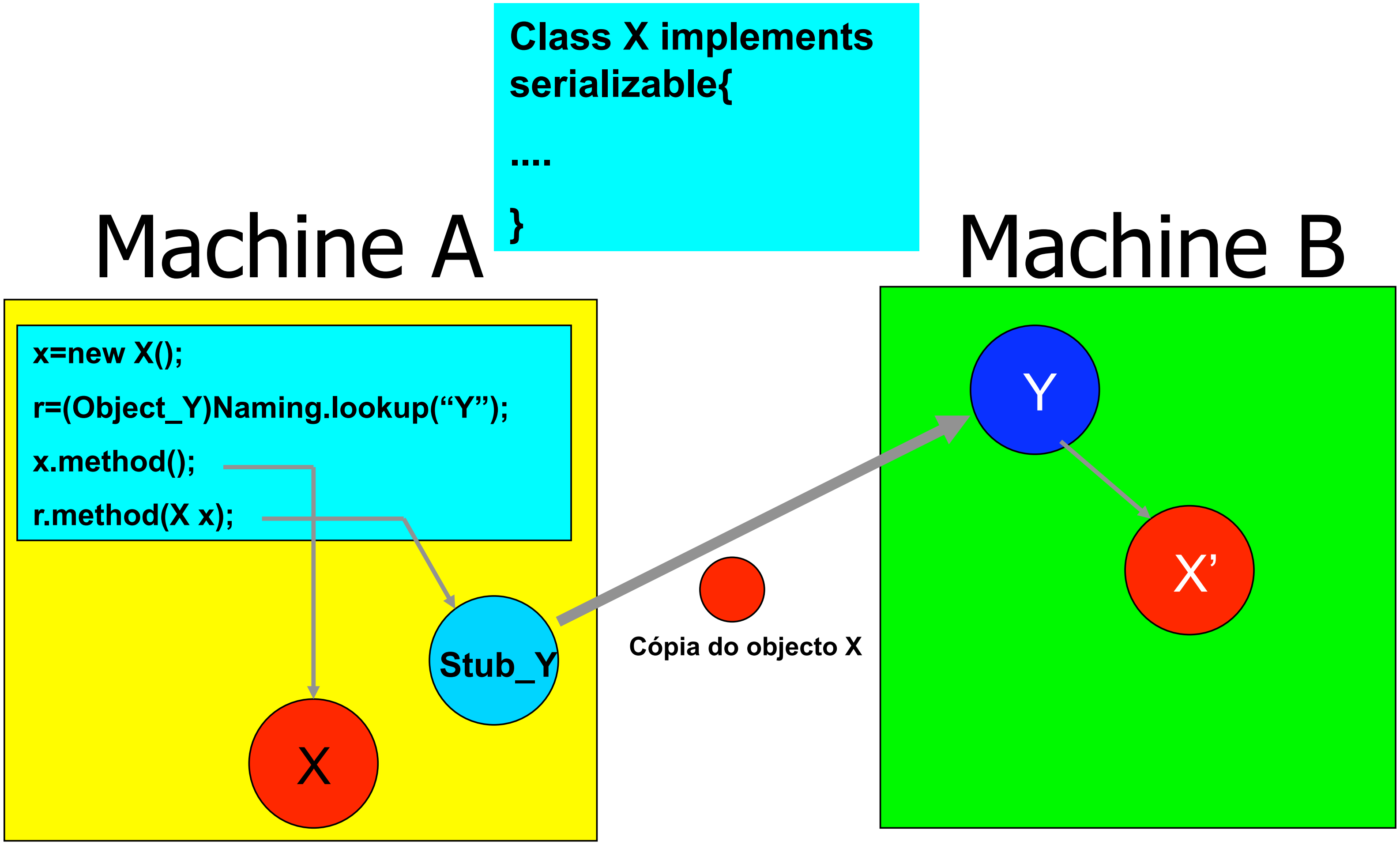
- **Primitive types**
  - passed by value
- **Remote objects**
  - passed by reference
- **Non-remote objects**
  - passed by value
  - uses Java Object Serialization

Parameter Passing

Parameter	Atomic types (int etc.)	Non-remote object	Remote object
Local	by-value	by-reference	by-reference
Remote	by-value	by-value	by-reference

- Non-remote objects passed to a remote object must implement `java.io.Serializable`.
- Any changes made to a non-remote object passed to a remote object occur only on the passed copy, not on the original.
- Any changes made to remote objects passed to a remote method are visible in the source objects.

Remote and Non-Remote Objects



```
Class X implements
serializable{

....

}
```

```
x=new X();
r=(Object_Y)Naming.lookup(“Y”);
x.method();
r.method(X x);
```

X: serializable object

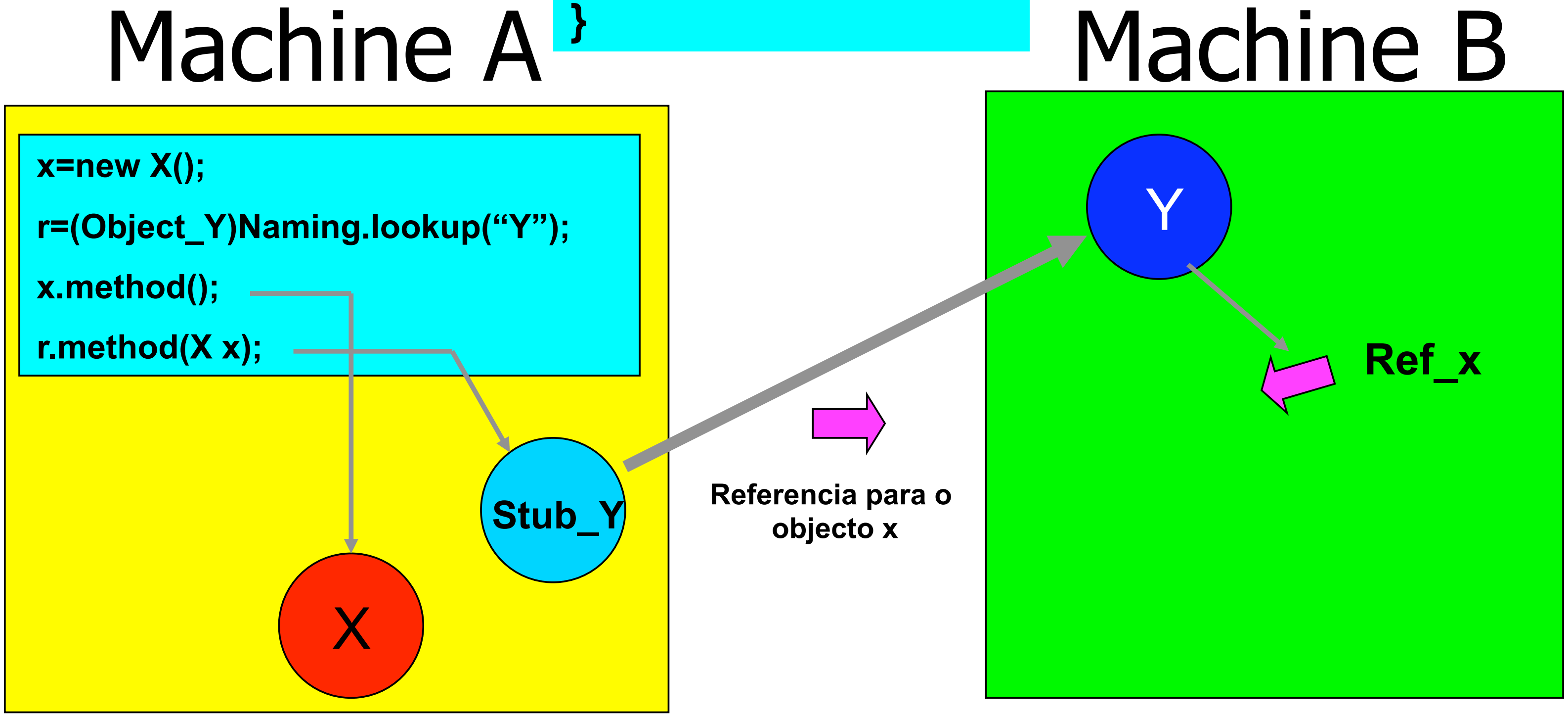
Y: remote object

Passagem por Referencia

```
Class X extends
Remote{

....

}
```



Machine A

```
x=new X();
r=(Object_Y)Naming.lookup("Y");
x.method();
r.method(X x);
```

Referencia para o objecto x

CODIGO DO Y:

```
ref_x.method();
```

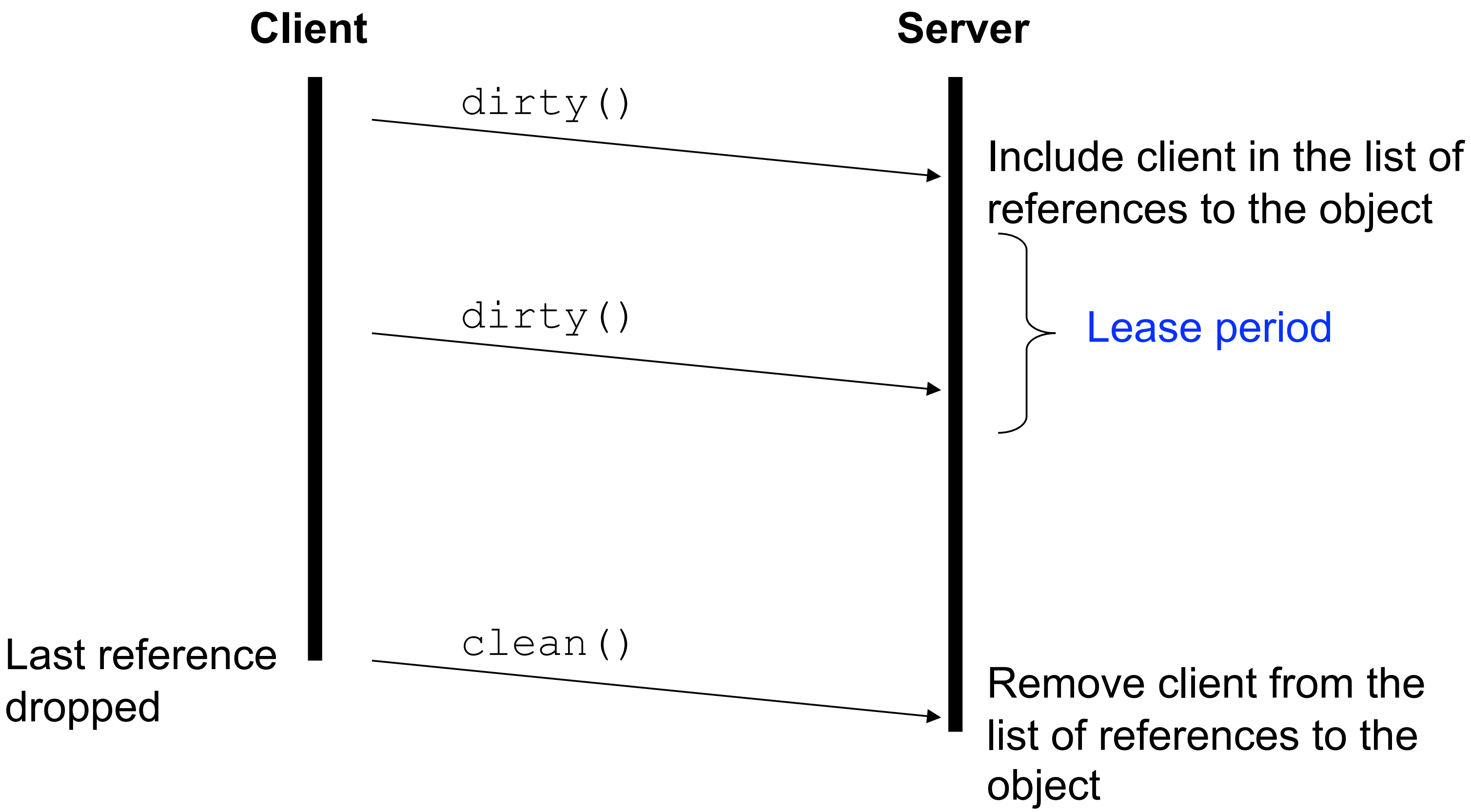
X: remote object

Y: remote object

Distributed Garbage Collection (I)

- RMI uses an algorithm similar to Modula-3 “Network Objects” to court references
- When a reference enters in a JVM, the client must call a `dirty()` method in the server
- After the `dirty()` call is received, the client can hold (and renew) the reference for some time:
  - LEASE PERIOD
- If a remote reference expires that lease period the remote object is available for garbage collection.

Distributed Garbage Collection (II)



Client

Server

`dirty()`

Include client in the list of references to the object

`dirty()`

Lease period

`clean()`

Last reference dropped

Remove client from the list of references to the object

Distributed Garbage Collection (III)

- The local JVM maintains reference counters of its “live” remote objects
- When the client JVM drops a remote reference object it must send a `clean()` call

Distributed Garbage Collection (IV)

- This protocol includes a number of subtleties:
  - It needs to ensure that clean() and dirty() calls arrive in correct order to avoid premature collection of a remote object
- When an RMI object is not referenced by any client, RMI uses a weak reference
  - To allow the local garbage collector to remove the object

Distributed Garbage Collection (V)

- To receive “unreferenced” notifications, a remote object must implement the interface `java.rmi.server.Unreferenced`
  - Method `unreferenced()` is invoked
- A partition in the network may cause a premature collection of an object
- If the client attempts to use an expired reference it will get a `RemoteException`

HTTP Tunneling

- RMI opens dynamic socket connections.
- Does not work if there is a firewall.

- Solution: **HTTP Tunneling**
  - Encapsulate the RMI call within an HTTP POST

- HTTP Tunneling: does not allow the use of RMI callback.

Java RMI Examples

Hello Interface

```
import java.rmi.*;

public interface Hello extends Remote {

  public String sayHello() throws java.rmi.RemoteException;

}
```

HelloImpl (Server)

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

public class HelloImpl extends UnicastRemoteObject implements Hello {

  public HelloImpl() throws RemoteException {
    super();
  }

  public String sayHello() throws RemoteException {
     System.out.println("print do lado do servidor...!");

    return "Hello, World!";
  }
  //==========================================================
  public static void main(String args[]) {

    try {
      HelloImpl h = new HelloImpl();
      Naming.rebind("rmi://localhost/hello", h);
      System.out.println("Hello Server ready.");
    }
     catch (RemoteException re) {
      System.out.println("Exception in HelloImpl.main: " + re);
    }
    catch (MalformedURLException e) {
      System.out.println("MalformedURLException in HelloImpl.main: " + e);
    }   }   }
```

HelloClient

```
import java.rmi.*;

public class HelloClient {

  public static void main(String args[]) {

    System.getProperties().put("java.security.policy","policy.all") ;
    System.setSecurityManager(new RMISecurityManager());

    try {

      Hello h = (Hello) Naming.lookup("rmi://localhost/hello");

      String message = h.sayHello();

      System.out.println("HelloClient: " + message);
    }
    catch (Exception e) {
      System.out.println("Exception in main: " + e);
    }

  }

}
```

RMI Callbacks

Callbacks

- Used on complex 2-way interactions
- Servers may wish to make calls back to the client
  - Error or problem reporting
  - Periodic updating & progress reports
  - In OO programs the role of clients and servers are not always rigid. They often operate in a peer-to-peer manner.

- Some problems…
  - Robustness
  - Servers with state
  - Garbage collection

Callback – How-To

- How do you create a callback?
  - Make your client into a server!

- Make your client implement a **Remote interface**.
  - Define a client remote interface

- Make it available as a Server (export your client interface as a remote object)
  - **`extend UnicastRemoteObject`**

- Pass a client remote reference to the server. The server can then use this reference to make calls on the client.

Interfaces

Server:

```
import java.rmi.*;
public interface Hello_S_I extends Remote {
  public void print_on_server(String s) throws java.rmi.RemoteException;
  public void subscribe(String name, Hello_C_I client) throws
    RemoteException;
}
```

Client:

```
import java.rmi.*;
public interface Hello_C_I extends Remote{
    public void print_on_client(String s) throws java.rmi.RemoteException;
}
```

Server

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

public class HelloServer extends UnicastRemoteObject implements Hello_S_I {

  static Hello_C_I client;

  public HelloServer() throws RemoteException {
    super();
  }

  public void print_on_server(String s) throws RemoteException {
    System.out.println("> "+s);
  }

  public void subscribe(String name, Hello_C_I c) throws RemoteException {
   System.out.println("Subscribing "+name);
   System.out.print("> ");
   client = c;
  }

```

Server (cont.)

```
public static void main(String args[]) {
    String a;
    System.getProperties().put("java.security.policy",
   "policy.all") ;
    System.setSecurityManager(new RMISecurityManager());
    try {
      HelloServer h = new HelloServer();
      Naming.rebind("hello", h);
      System.out.println("Hello Server ready.");
      while(true){
        System.out.print("> ");
        a=User.readString();
        client.print_on_client(a);
      }
    }
     catch (RemoteException re) {
      System.out.println("Exception in HelloImpl.main: " + re);
    }
    catch (MalformedURLException e) {
      System.out.println("MalformedURLException in HelloImpl.main:
   " + e);
    }
}
}
```

Client

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
public class HelloClient extends UnicastRemoteObject   implements Hello_C_I
   {
  HelloClient() throws RemoteException{
        super();
    }
  public void print_on_client(String s)throws RemoteException{
   System.out.println("> "+s);
   }
  public static void main(String args[]) {
    // usage: java HelloClient username
    System.getProperties().put("java.security.policy","policy.all") ;
    System.setSecurityManager(new RMISecurityManager());
    try {
      Hello_S_I h = (Hello_S_I) Naming.lookup("hello");
      HelloClient c= new HelloClient();
      h.subscribe(args[0], (Hello_C_I) c);
      System.out.println("Client sent subscription to server");
    }
    catch (Exception e) {
      System.out.println("Exception in main: " + e);
    }  } }
```