The Supercomputing People

Corporate Profile

Session 1:

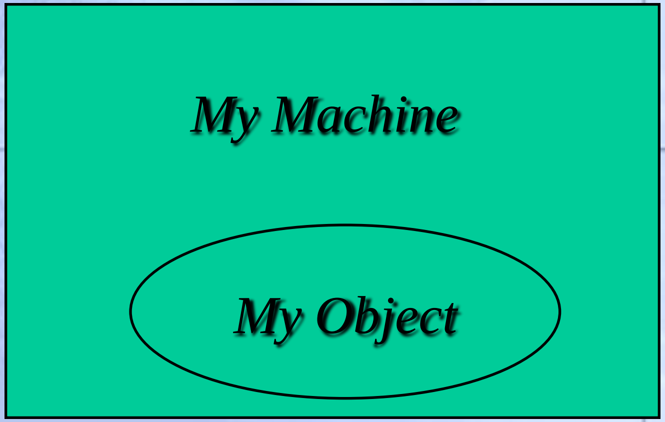
Remote Method Invocation (RMI)

Contents

- Overview of RMI
- Network Programming in Java using RMI
- Steps to build an RMI application
- Compiling and Running an RMI program

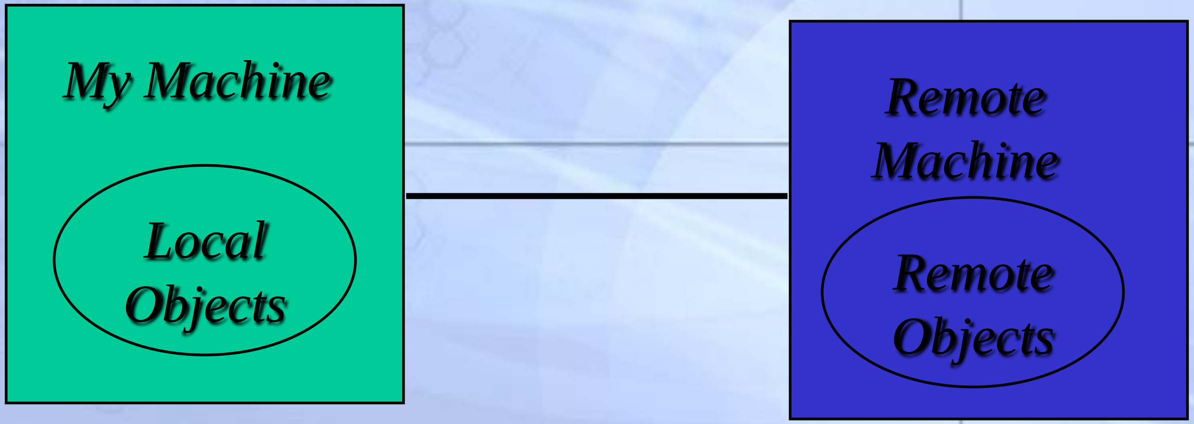
In the Good Old Days...

Only local objects existed



Today's World...

Network and Distributed Objects



Different Approaches to Distributed Computation

- High-performance, parallel scientific apps
- Connecting via sockets
  - custom protocols for each application
- RPC/DCOM/CORBA/RMI
  - make what looks like a normal function call
  - function is actually invoked on another machine
  - Arguments are *marshalled for transport*
  - Value is *unmarshalled on return*

What is RMI?

- RMI enables the programmer to create distributed Java applications, in which the methods of remote Java objects can be invoked from other Java Virtual Machines, possibly on different hosts.

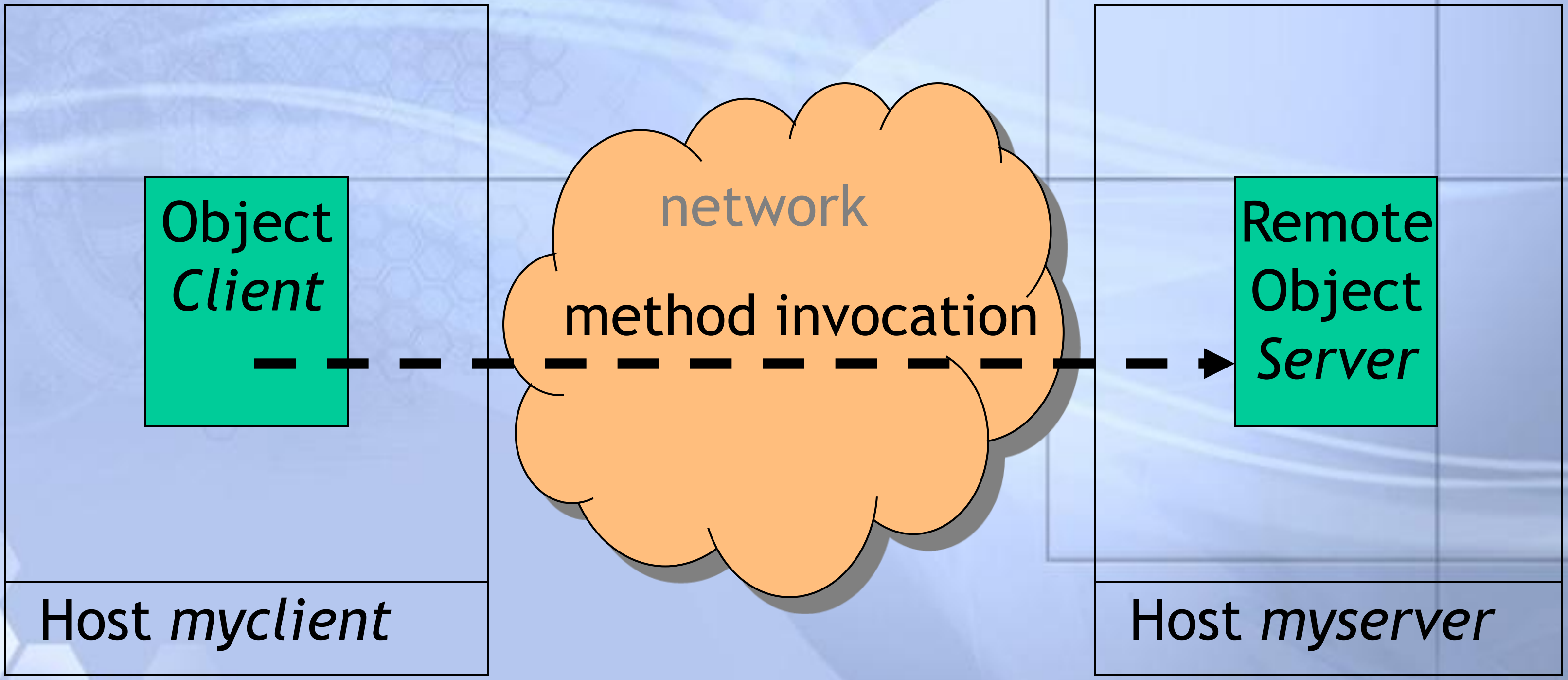
- A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the naming service provided by RMI or by receiving the reference as an argument or a return value.

- A client can call a remote object in a server, and that server can also be a client of other remote objects.

- RMI uses object serialization to marshal and unmarshal parameters.

What is RMI?

- Remote Method Invocation

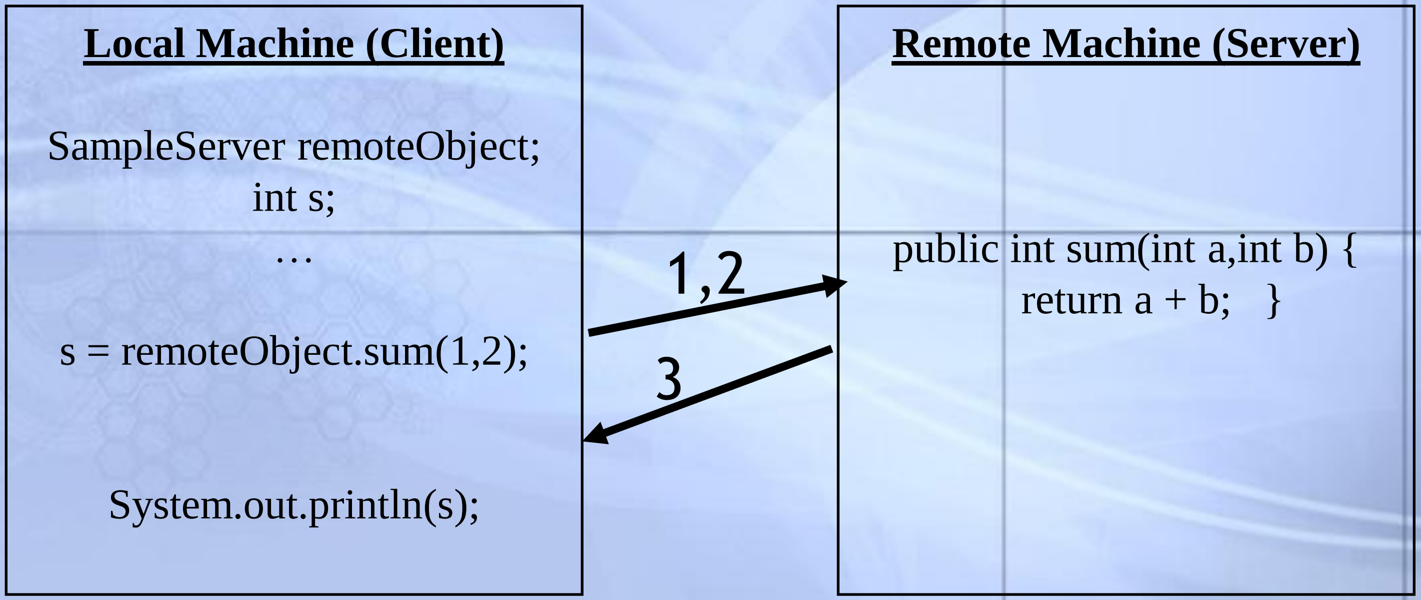


Clear Definition of RMI

- Mechanism for performing method calls between different Java virtual machines (JVM)
- Attempts to hide all the low level details such as serialization, communication protocol, etc.
- Object-oriented analogue of Remote Procedure Call

Overview

- Java RMI allowed programmer to execute remote function class using the same semantics as local functions calls.



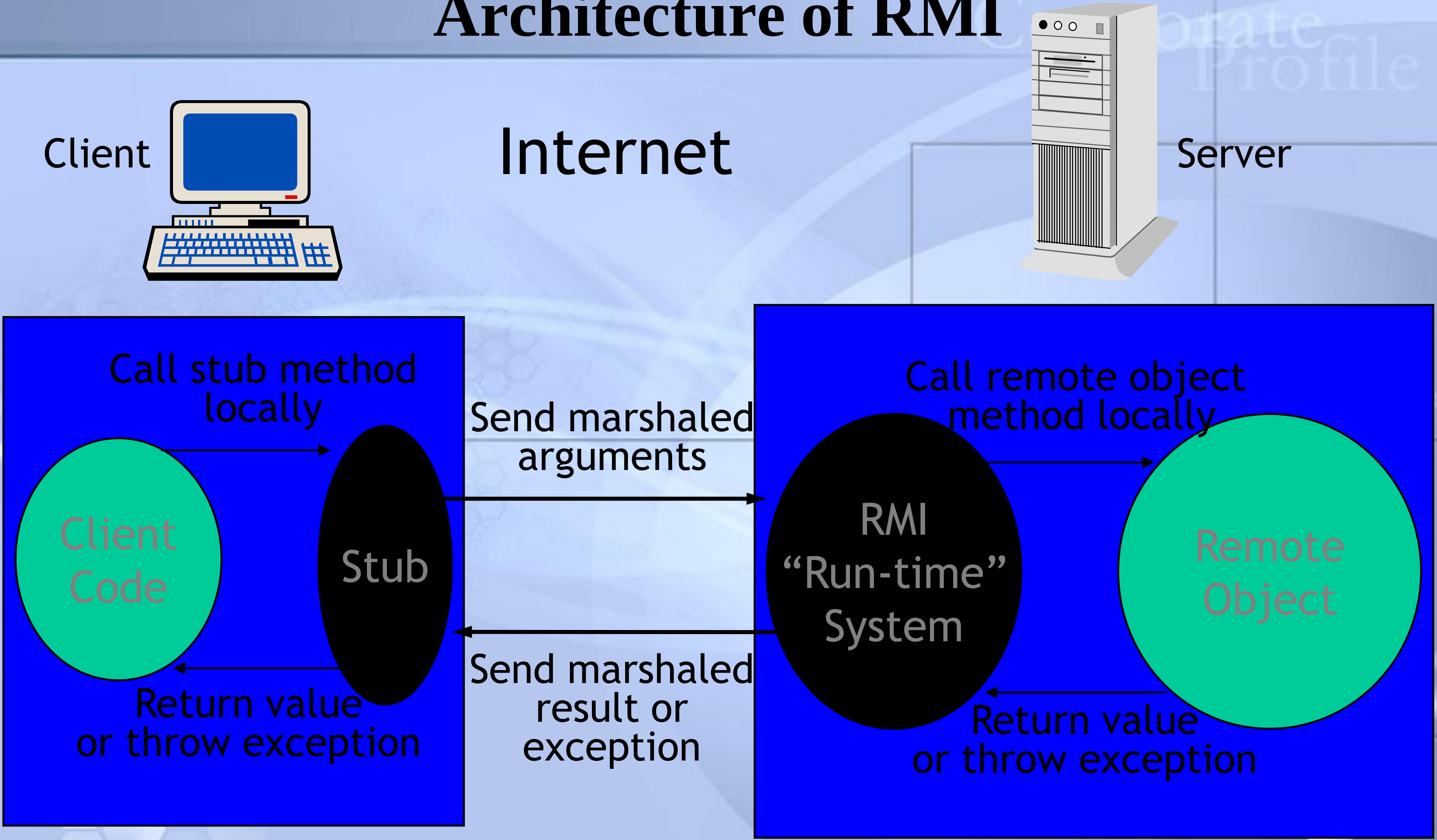
Local Machine (Client)

```
SampleServer remoteObject;
int s;
…

s = remoteObject.sum(1,2);

System.out.println(s);
```

Remote Machine (Server)

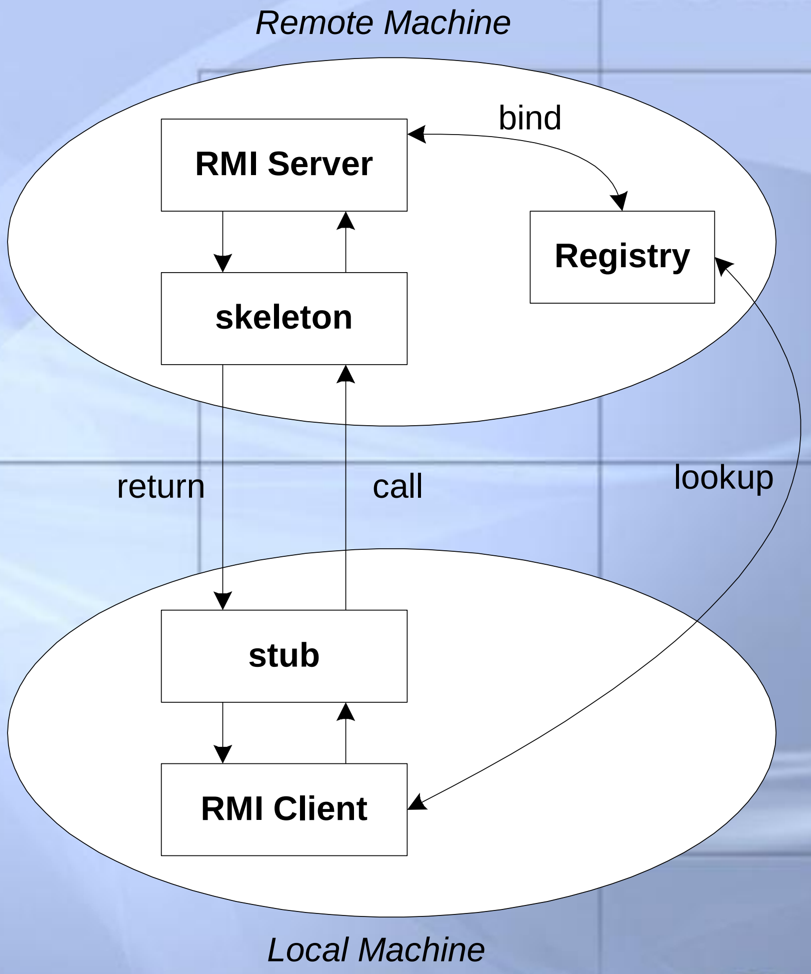
```
public int sum(int a,int b) {
    return a + b;   }
```

Architecture of RMI


Client
Internet
Server
Call stub method locally
Client Code
Stub
Return value or throw exception
Send marshaled arguments
Send marshaled result or exception
Call remote object method locally
RMI "Run-time" System
Remote Object
Return value or throw exception

The General RMI Architecture

- The server must first bind its remote object's name to the registry
- The client lookup these names in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.



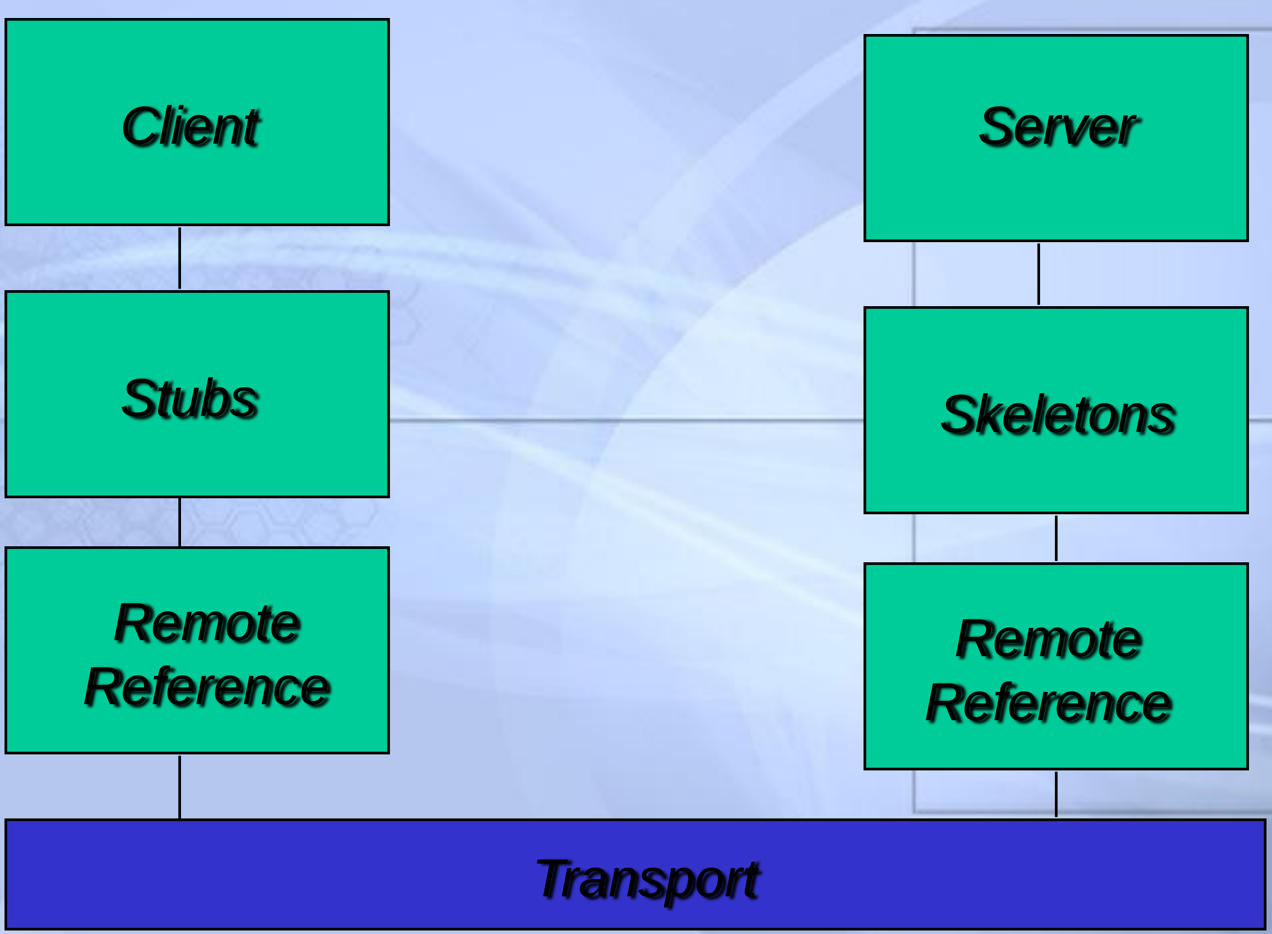
Java RMI allows...

- **provide user with a “thin client ”**
  - allows good performance on lowend workstations
- **run server on high end hardware**
  - maximize $ investment over many clients
  - server remote from client
- **Distributed network object**

What RMI is...and isn’t

- Java only solution to the problem of distributed object computing
- Unlike CORBA it is not language independent
- Isn’t meant to replace CORBA
- Good for java only solutions, not easy to integrate with legacy code

Layered view of RMI



The parts

- **Client - user interface**
- **Server - data source**
- **Stubs**
  - marshals argument data (serialization)
  - unmarshals results data (deserialization)
- **Skeletons (not need in Java 2, done also in stubs)**
  - unmarshals argument data
  - marshals results data

The parts (cont.)

- **Remote Reference Layer**
  - provides a *RemoteRef* object that represents the link to the remote service implementation object.
  - encodes and decodes the on-wire protocol
  - implements the remote object protocols

- **Transport layer**
  - The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.
  - handles the underlying socket handling required for communications
    - sets up and maintains connections
    - communications related error handling

Terms and Terminology in Java for RMI

java.rmi.Remote

- The interface **java.rmi.Remote** is a *marker interface*.
- It declares no methods or fields;
- Extending it tells the RMI system to treat the interface concerned as a remote interface.

java.rmi.UnicastRemoteObject

- Objects that require remote behavior should extend **RemoteObject**, typically via **UnicastRemoteObject**.

- defines a non-replicated remote object whose references are valid only while the server process is alive.

- provides support for point-to-point active object references (invocations, parameters, and results) using TCP streams.

java.rmi.RemoteException

- Require all remote methods be declared to throw **RemoteException.**

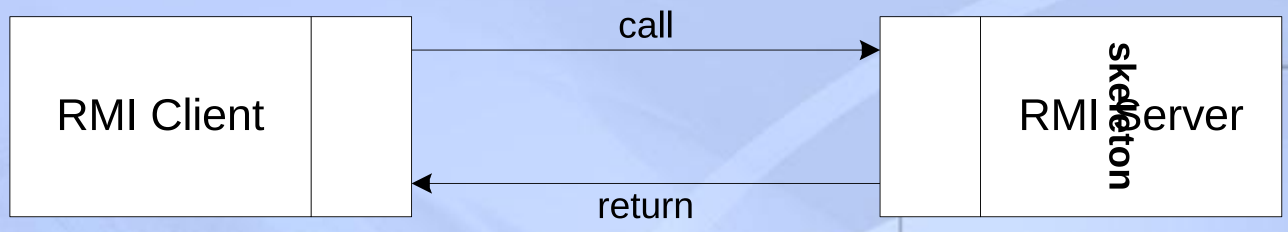
- To handle
  - unexpected failure of the network or remote machine.

Remote Object and Interfaces

- **Remote Objects** are those that can be referenced remotely
  - extends **java.rmi.UnicastRemoteObject**
  - constructor/methods throws **java.rmi.RemoteException**

- **Remote interfaces** describe services that can be provided remotely
  - extends **java.rmi.Remote** interface
  - all methods throw **java.rmi.RemoteException**

The Stub and Skeleton



- stub
  - responsible for sending the remote call over to the server-side skeleton
  - opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton.
- skeleton
  - contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

rmic Compiler

- The only "compiler" technology peculiar to RMI => the **rmic** *stub generator*.

  (compiled in the normal way with **javac**)

- input => remote implementation class
- output => a new class
  - contain code to send arguments to, and receive results from, a remote object, whose Internet address is stored in the stub instance.

Marshalling of Arguments

- Objects passed as arguments must be marshaled for transmission over the network.
- Java has a general framework for converting objects to an external representation that can later be read back into an arbitrary JVM.
- This framework is Object Serialization.

Developing an RMI Application

Steps for Developing an RMI System

Implementation

1. Define the remote interface
2. Develop the remote object by implementing the remote interface
3. Develop the client program.

Compilation

4. Compile the Java source files.
5. Generate the client stubs and server skeletons.

Deployment

6. Start the Java RMI registry
7. Start the server
8. Run the client

Step 1: Defining the Remote Interface

- To create an RMI application, the first step is the defining of a remote interface between the client and server objects.

```
/* SumInterface.java */
import java.rmi.*;

public interface SumInterface extends Remote
{
  public int sum(int a,int b) throws RemoteException;
}
```

Step 2: Develop the remote object and its interface

- The server is a simple unicast remote server.
- Create server by extending **java.rmi.server.UnicastRemoteObject**.
- The server uses the **RMISecurityManager** to protect its resources while engaging in remote communication.

Step 2 (Cont.)

```
/* SumImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class SumImpl extends UnicastRemoteObject
                              implements SumInterface
{
  SumImpl() throws RemoteException
  {
     super();
  }
  public int sum(int a,int b) throws RemoteException
  {
     return a + b;
  }
}
```

Parameters and Return Values

- Primitive data type
- Remote objects – by reference
- Serializable objects - by copy

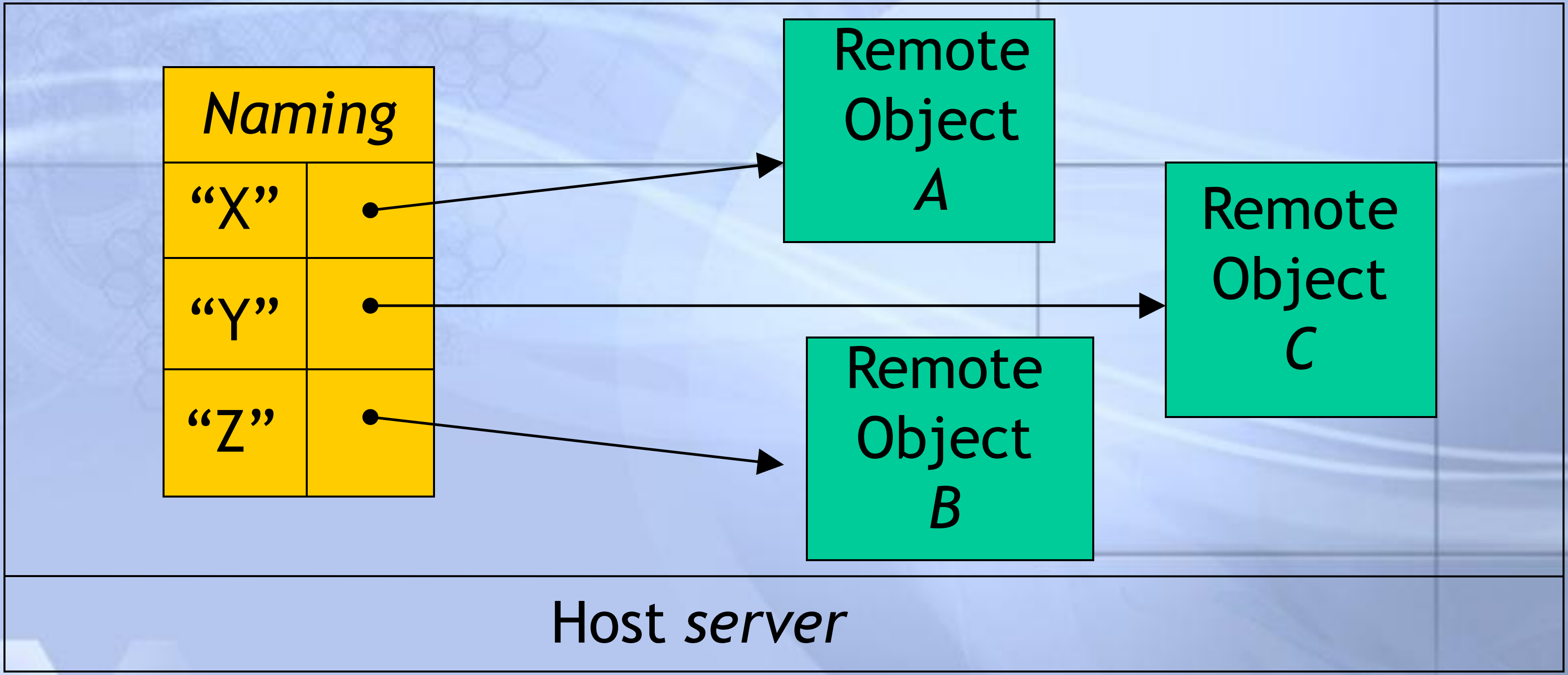
❖ Many of the core classes, including the classes in the packages java.lang and java.util, implement the Serializable interface

Step 2 : Naming

- The server must bind its object's name to the registry, the client will look up the remote object's name.
- Use **java.rmi.Naming** class to bind the server name to registry.
- In the **main** method of your server object, the RMI security manager can be created and installed.

Naming Service

- Directory that associates names to remote objects (*bind*)



Step 2: Naming

```
/* SumImpl.java*/
  public static void main(String args[])
  {  try
      {
        //create a local instance of the object
        SumImpl obj= new SumImpl();

        //put the local instance in the registry
        Naming.rebind(“rmi://localhost:1099/sumObj" , obj);

        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());   }
  }
```

Step 3: Develop the client program

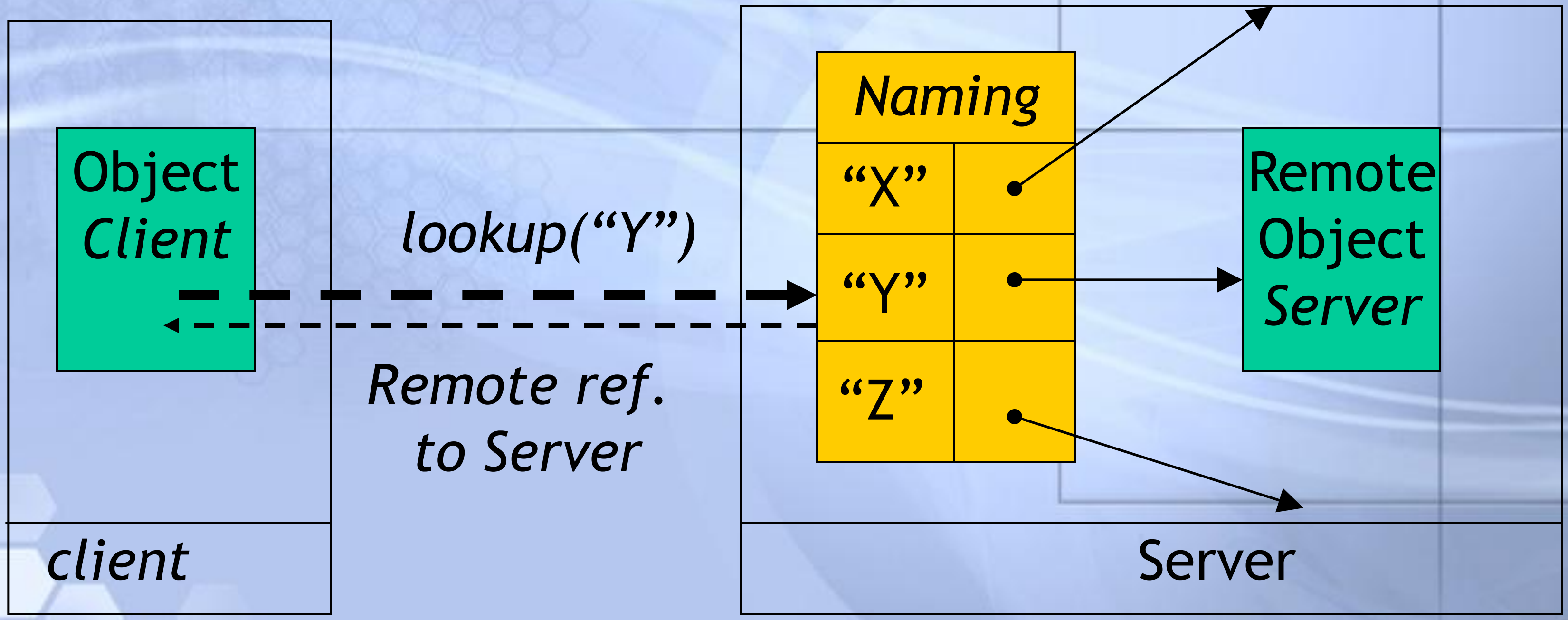
- Create and install a security manager
- Locate the RMI registry
- Get the remote object stub from the RMI registry
- Method call is blocked until the method returns

Step 3: Develop the client program

```
import java.rmi.*; import java.rmi.server.*; import java.net.*;
public class SampleClient  {
   public static void main(String[]  args)    {
      try
        {
          String url = “rmi://localhost:1099/sumObj";
          SumInterface remoteObject = (SumInterface)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());
        }
    }
}
```

Naming Service : lookup

- Client use Naming Service to find a particular Server object (lookup)



Step 4 & 5: Compilation & Generate the stubs and skeletons

- Assume the program compile and executing at elpis on ~/rmi
- Once the interface is completed, you need to generate stubs and skeleton code. The RMI system provides an RMI compiler (rmic) that takes your generated interface class and procedures stub code on its self.

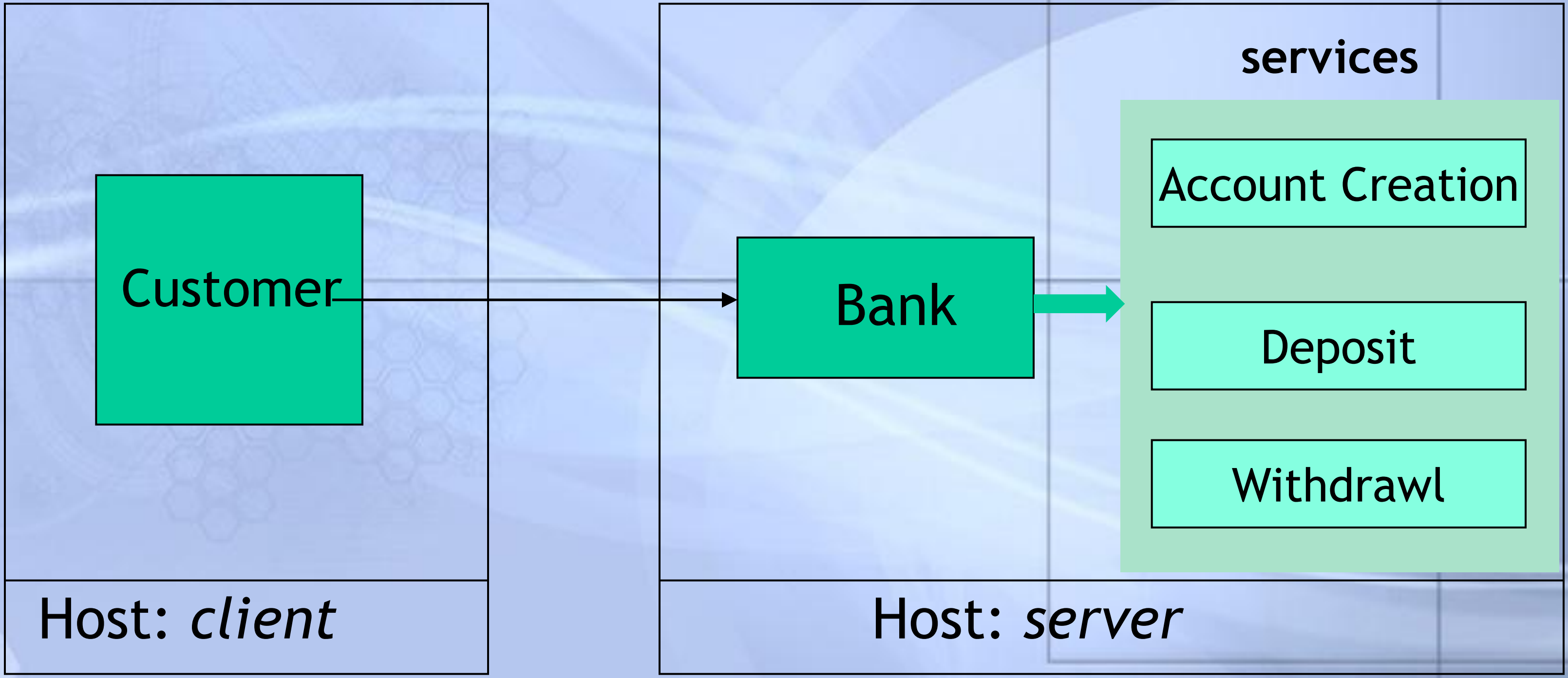
```
elpis:~/rmi> set CLASSPATH="~/rmi"


elpis:~/rmi> javac SumInterface.java
elpis:~/rmi> javac SumImpl.java
elpis:~/rmi> rmic SumImpl


elpis:~/rmi> javac SampleClient.java
```

To run

- Copy **stub** file to client machine
- Start the RMI registry
  - rmiregistry is in the JSDK bin directory
- Start the RMI Server
- Start the RMI Client

Lets make Exercise !

NetBank example



Thank You!