Institute of Computer Science
Department of Mathematics and Computer Science

Freie Universität Berlin

Algorithms and Programming IV

Java RMI (19-3)

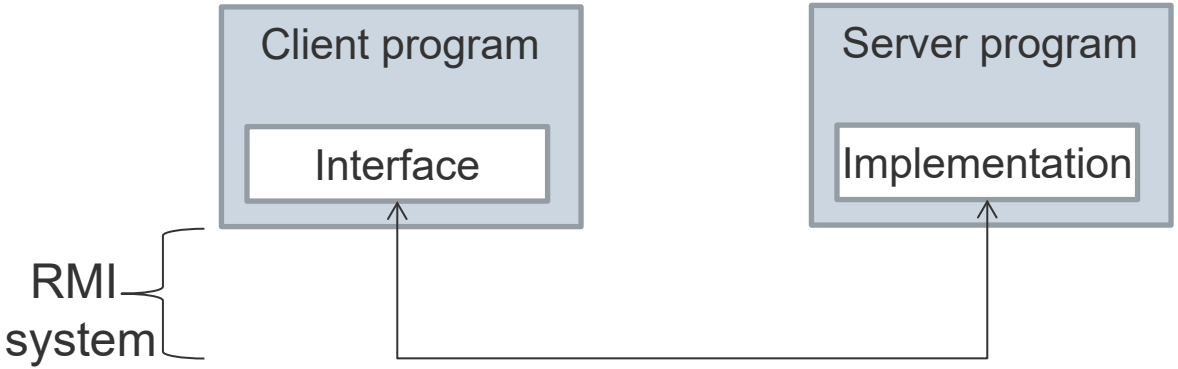
Summer Term 2021 | 18.06.2021

Barry Linnert

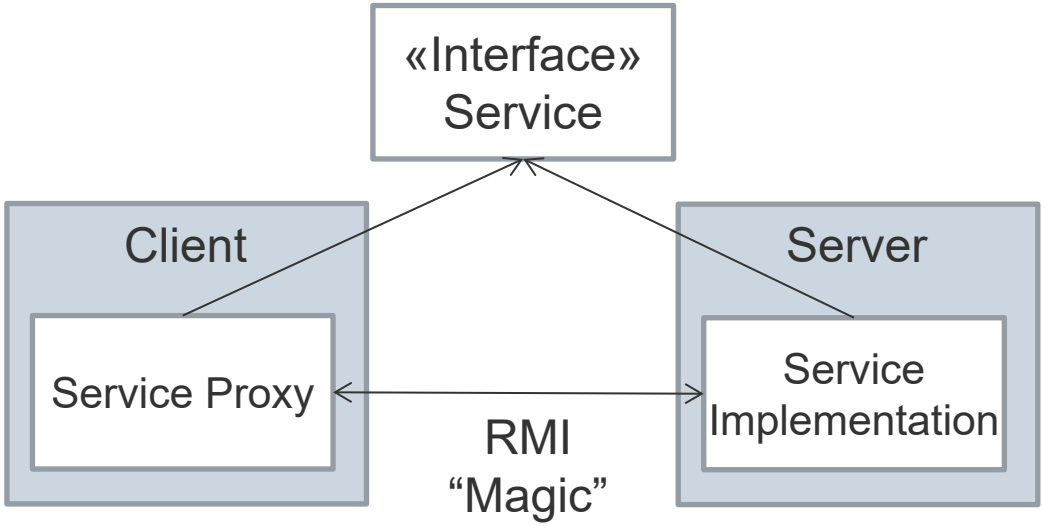
Separation of Concerns

RMI architecture is based on one important principle: *the **definition of behavior** and the **implementation of that behavior** are separate concepts*.

RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.



Implementation of the Interface



Abstraction Layers in the RMI Implementation

1. Stub and Skeleton Layer

Intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service

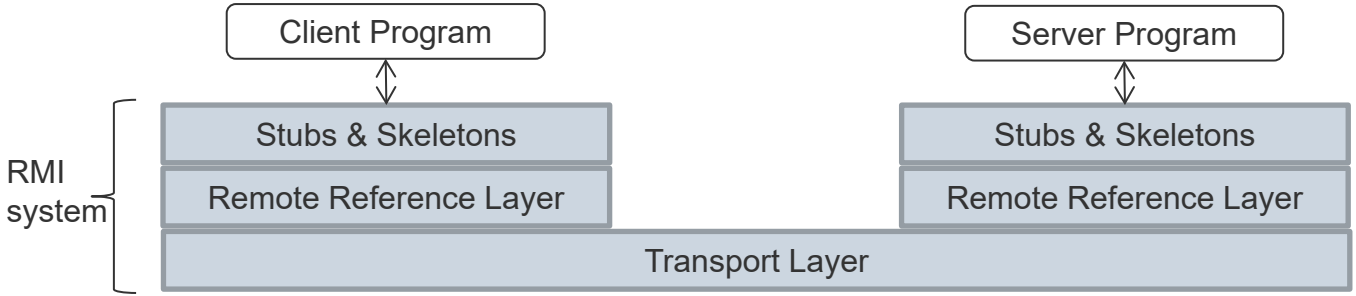
2. Remote Reference Layer

Interpret and manage references made from clients to the remote service objects

3. Transport Layer

Is based on TCP/IP connections between machines in a network

Provides basic connectivity, as well as some firewall penetration strategies



Server and Client Programs

Server program

- Contains classes for the dispatcher and skeletons, together with the implementations of the classes of all of the servants
- Contains a *initialization* section (responsible for creating and initializing at least one of the servants to be hosted by the server)
- Generally allocates a separate thread for the execution of each remote invocation -> designer of the remote object implementation must allow concurrent executions

Client program

- Contain the classes of the proxies for all of the remote objects that it will invoke
- Require a means of obtaining a remote object reference for at least one of the remote objects held by the server -> binder

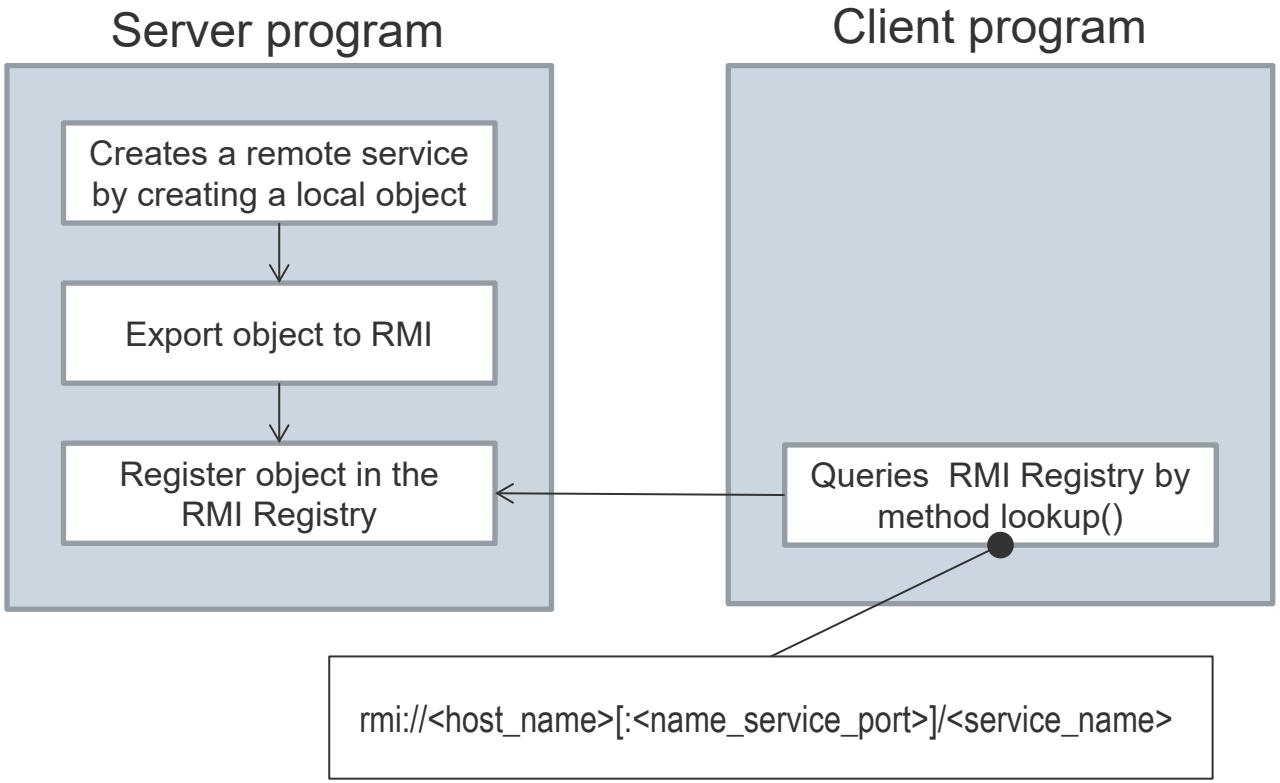
Naming Remote Objects

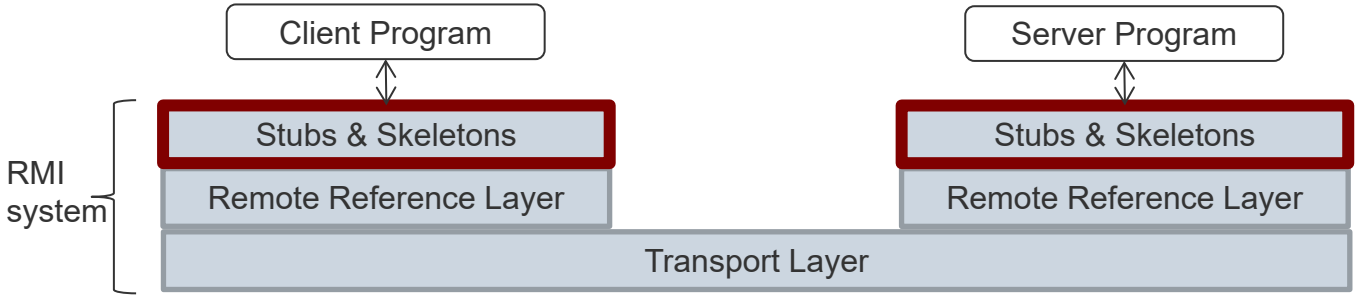
How does a client find a RMI remote service?

RMI includes a simple service called the RMI Registry, *rmiregistry*.

The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.

Naming Remote Objects (*cont.*)





RMI Architecture Layers

STUB AND SKELETON LAYER

Stub and Skeleton Layer

RMI uses the **Proxy design pattern**

- The stub class is the *proxy*
- The remote service implementation class is the *RealSubject*

The Skeleton is a helper class.

- Carries on a conversation with the stub
- Reads the parameters for the method call → makes the call to the remote service implementation object → accepts the return value → writes the return value back to the stub

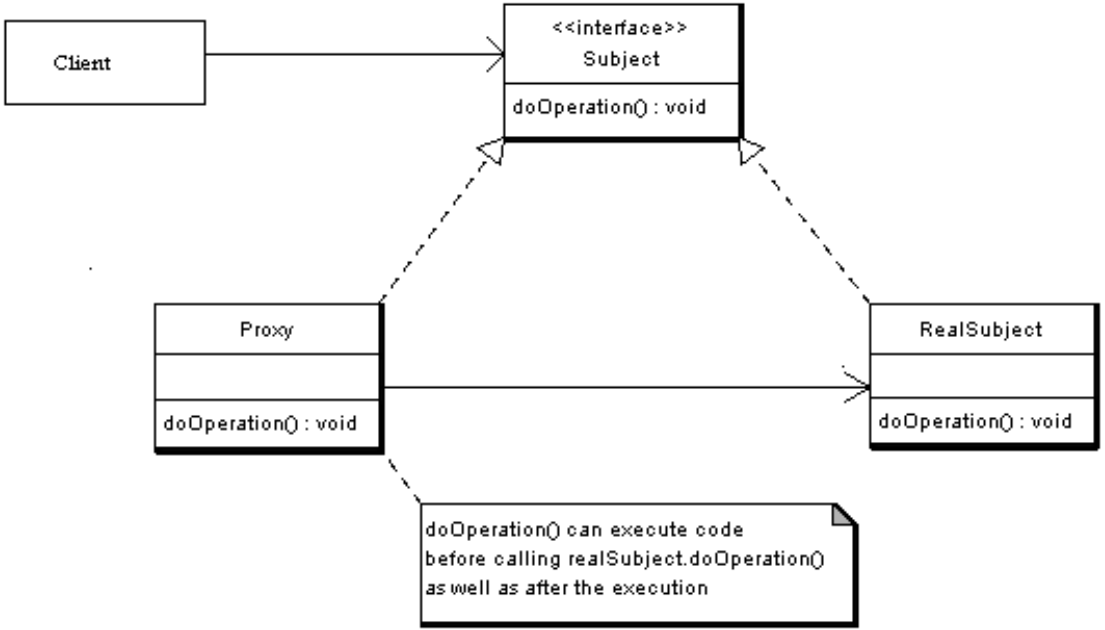
Please note: In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses ***reflection*** to make the connection to the remote service object.

Proxy Design Pattern

Motivation

- Provide a surrogate or placeholder for another object to control access to it

Implementation



Proxy Design Pattern: Applications

Remote Proxies: Providing a local representation for an object that is in a different address space. A common example is Java RMI stub objects.

Virtual Proxies: Delaying the creation and initialization of expensive objects until needed, where the objects are created on demand.

Protection Proxies: A proxy controls access to RealSubject methods, by giving access to some objects while denying access to others.

Smart References: Providing a sophisticated access to certain objects such as tracking the number of references to an object and denying access if a certain number is reached, as well as loading an object from database into memory on demand.

RMI with Reflections

Reflection enables Java code

- to dynamically discover information about the fields, methods and constructors of loaded classes (at runtime), and
- to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects (within security restrictions).

More information:

http://download.oracle.com/javase/tutorial/reflect/

Using *Reflection* in RMI

1. Proxy has to marshal information about a method and its arguments into a request.
2. For a method it marshals an object of class *Method* into the request. It then adds an array of *objects* for the method's arguments.
3. The dispatcher unmarshals the *Method* object and its arguments from request message.
4. The remote object reference is obtained from remote reference module.
5. The dispatcher then calls the *Method* object's "invoke" method, supplying the target object reference and the array of argument values.
6. After the method execution, the dispatcher marshals the result or any exceptions into the reply message.

dispatcher is generic and can be used for all classes of remote objects.

Java *Reflection* API

java.lang.reflect.Method class

- provides methods for obtaining the type information for parameters and return value.
- can be used to invoke methods on a given object: **java.lang.reflect.Method.invoke()**
  - First argument is the object instance on which this particular method is to be invoked
  - Second argument are the method parameters packed in an array •Return value is the return object of the method

More information:
https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Method.html

Example Factory Design Pattern

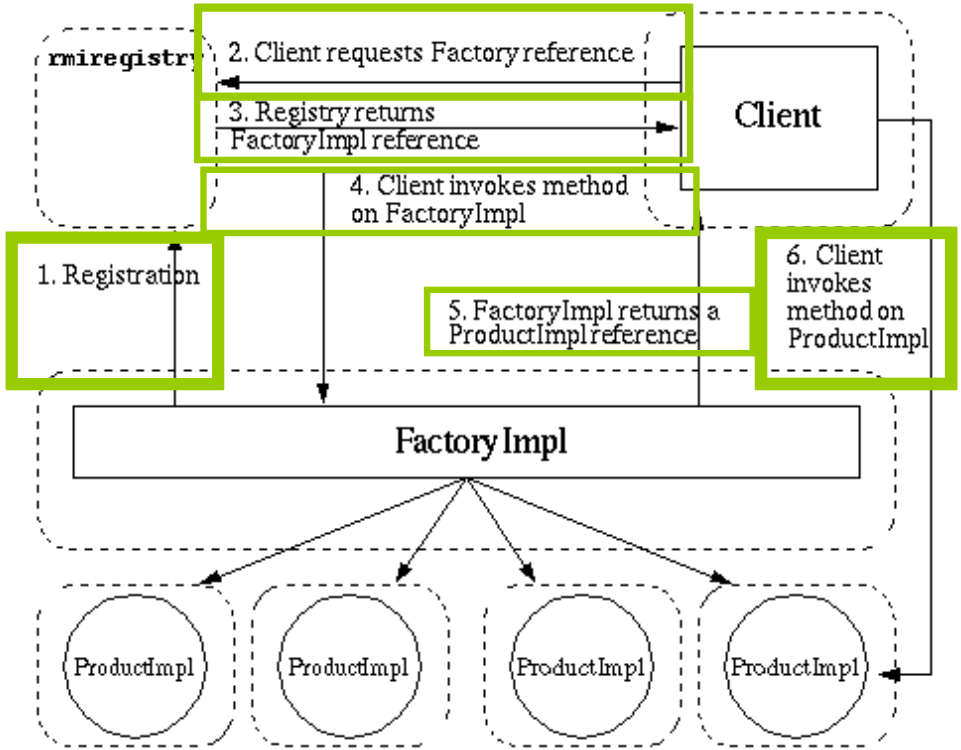
1

http://www.flickr.com/photos/23065375@N05/

2

http://www.flickr.com/photos/dunedinpubliclibraries/

Factory Design Pattern



How could the examples be implemented in Java RMI?

1

```
public interface AccountManager extends Remote {
    public Account createAccount() throws RemoteException;
}

public interface Account extends Serializable {
    public depositMoney();
    ...
}
```

2

```
public interface Librarian extends Remote {
    public LibraryCard createLibraryCard() throws RemoteException;
    public Loanable getLoanable() throws RemoteException;
}

public interface LibraryCard extends Serializable {...}
public interface Loanable extends Serializable {...}
```

Dynamic Code Loading with RMI

The Java VM interprets bytecode or compiles it on the fly and can load and run code dynamically.

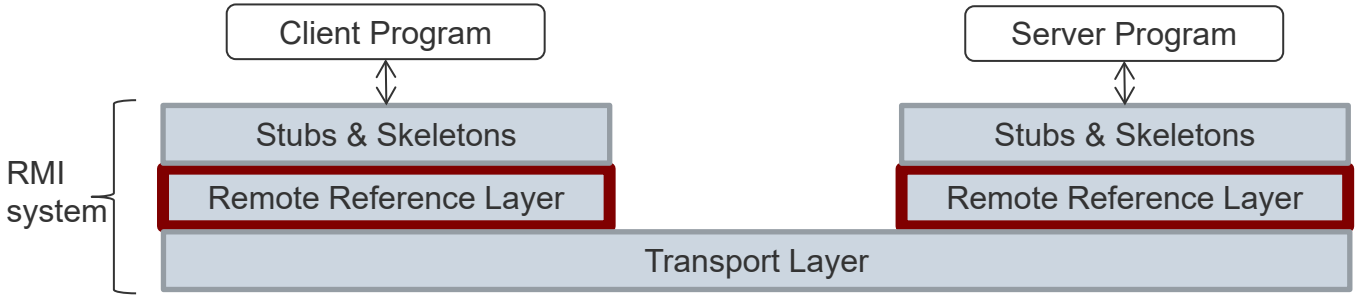
Bytecode loading is encapsulated in a ClassLoader which can be customized by developers.

Can load bytecode from anywhere (specifically from the network).

For example the URLClassLoader (“out of the box”) loads from a Uniform Resource Locator (URL) (per file://, ftp://, http://).

Further information: http://download.oracle.com/javase/1.4.2/docs/guide/rmi/codebase.html

Slide adapted from Volker Roth



RMI Architecture Layers

REMOTE REFERENCE LAYER

Remote Reference Layer

Defines and supports the invocation semantics of the RMI connection.

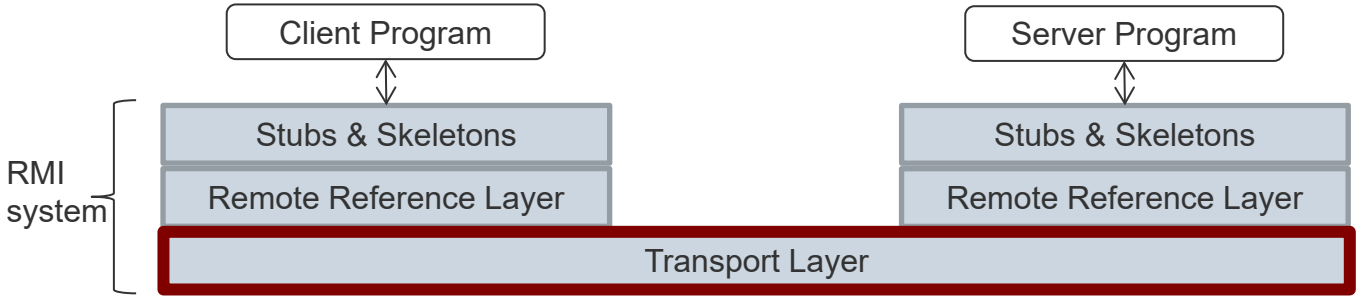
Provides a **RemoteRef** object that represents the link to the remote service implementation object.

JDK 1.1 implementation of RMI

- Provides a unicast, point-to-point connection
- Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system

Java 2 SDK implementation of RMI

- When a method call is made to the proxy for an activatable object, RMI determines if the remote service implementation object is dormant
- If yes, RMI will instantiate the object and restore its state from a disk file

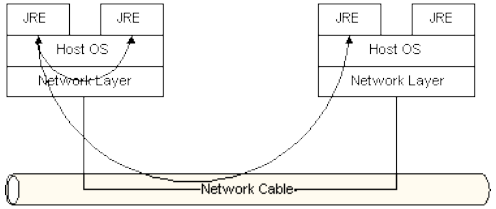


RMI Architecture Layers

TRANSPORT LAYER

Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.



On top of TCP/IP, RMI uses a wire level protocol called **Java Remote Method Protocol (JRMP)**. JRMP is a proprietary, stream-based protocol that is only partially specified in two versions:

- First version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server.
- Second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.

Abstraction Layers in the RMI Implementation