A ***remote proxy*** acts as a local representative to a remote object.

***Remote object*** – It’s an object that lives in the heap of a different JVM or more generally a remote object that is running in a different address space.

***Local Representative*** – It’s an object that you can call local methods on and have them forwarded on the remote object.

***RMI:***

Client Heap Server Heap

In RMI, the client helper is a ‘stub’ and the service helper is a ‘skeleton’.

Client Heap RMI Stub RMI Skeleton Server Heap

***Making the Remote service:***

***Step one: Make a Remote Interface***

1. Extend java.rmi.Remote

public interface MyRemote ***extends Remote*** {

1. Declare that all methods must throw a RemoteException

import java.rmi.\*;

public interface MyRemote extends Remote {

public String sayHello() throws ***RemoteException***;

}

1. Be sure arguments and return values are primitives or Serializable

public ***String*** sayHello() throws RemoteException;

***Step two: Make a Remote Implementation***

1. Implement the Remote Interface

public class MyRemoteImpl extends UnicastRemoteObject ***implements MyRemote*** {}

1. Extends the UnicastRemoteObject

public class MyRemoteImpl ***extends UnicastRemoteObject*** implements MyRemote {}

1. Write a no-arg constructor that declares a RemoteException

Public MyRemoteImpl() ***throws RemoteException*** { }

1. Register the service with the RMI registry

try {

MyRemote service = new MyRemoteImpl();

Naming.rebind(“RemoteHello”, service);

} catch (Exception ex) {…}

***Step three: run rmiregistry***

1. Bring up a terminal and start the rmiregistry

***Step four: start the service***

1. Bring up another terminal and start your service

Before Java 5, we had to generate static stubs and skeletons using rmic. Now we don’t have to do this anymore and in fact we shouldn’t do it anymore, because static stubs and skeletons are deprecated. Instead, stubs and skeletons are generated dynamically. This happens automatically when we subclass the UnicastRemoteObject.

***How does the client get the stub object?***

1. Client does a lookup on the RMI registry

***Naming.lookup(“rmi://127.0.0.1/RemoteHello”);***

1. RMI registry returns the stub object
2. Client invokes a method on the stub, as if the stub IS the real service

Note: Complete Server-Side Code and Client-side code is in folder

Client Heap RMI Stub RMI Skeleton Server Heap

3.sayHello()

1. * + 1. 2.stubreturned

1.lookup() RMI registry(on server)

RemoteHello

***Proxy Pattern*** provides a surrogate or placeholder for another object to control access to it. Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.

Here are a few ways proxies control access:

* A ***remote proxy*** controls access to a remote object.
* A ***virtual proxy*** controls access to a resource that is expensive to create.
* A ***protection proxy*** controls access to a resource based on access rights.

<<interface>>

Subject

request()

Proxy

request()

RealSubject

request()

subject

***Remote Proxy:***

With Remote Proxy, the proxy acts as a local representative for an object that lives in a different JVM. A method call on the proxy results in the call being transferred over the wire, invoked remotely and the result being turned back to the proxy and then to the client.

request()

***Virtual Proxy:***

Virtual Proxy acts as a representative for an object that may be expensive to create. The Virtual Proxy often defers the creation of the object until it is needed; the Virtual Proxy also acts as a surrogate for the object before and while it is being created. After that, the proxy delegates requests directly to the RealSubject.

***Designing the CD cover Virtual Proxy***

<<interface>>

Icon

getIconWidth()

getIconHeight()

paintIcon()

ImageProxy

getIconWidth()

getIconHeight()

paintIcon()

ImageIcon

getIconWidth()

getIconHeight()

paintIcon()

Subject

How ImageProxy is going to work:

1. ImageProxy first creates an ImageIcon and starts loading it from a network URL.
2. While the bytes of the image are being retrieved, ImageProxy display “Loading CD cover, please wait…”.
3. When the image is fully loaded, ImageProxy delegates all method calls to the image icon, including paintIcon(), getWidth() and getHeight().
4. If the user requests a new image, we’ll create a new proxy and start the process over.

***Using the Java API’s Proxy to create a protection proxy***

Proxy

request()

RealSubject

request()

InvocationHandler

invoke()

<<interface>>

InvocationHandler

invoke()

<<interface>>

Subject

request()

***Scenario – Match Making in objectville***

***Wild Proxies:***

***Firewall Proxy:***

Controls access to a set of network resources, protecting the subject from “bad” clients.

***Example***: Corporate Firewall sysems

***Smart Reference Proxy:***

Provides additional actions whenever a subject is referenced, such as counting the number of references to an object.

***Caching Proxy:***

Provides temporary storage for results of operations that are expensive. It can also allow multiple clients to share the results to reduce computation or network latency.

***Example***: Web server proxies, content management and publishing systems.

***Synchronization Proxy:***

Provides safe access to a subject from multiple threads.

***Example:*** Seen hanging around JavaSpaces, where it controls synchronized access to an underlying set of objects in a distributed environment.

***Complexity Hiding Proxy:***

Hides the complexity and controls access to a complex set of classes. This is sometimes called the Façade Proxy for obvious reasons. The Complexity Hiding Proxy differs from the Façade Pattern on that the proxy controls access, while the Façade Pattern just provides an alternative interface.

***Copy-On-Write Proxy:***

Controls the copying of an object by deferring the copying of an object until it is required by a client. This is a variant of virtual proxy.

***Example:*** Java’s CopyOnWriteArrayList

* The Proxy Pattern provides a representative for another object in order to control the client’s access to it. There are a number of ways it can manage that access.
* A Remote Proxy manages interaction between a client and a remote object.
* A Virtual Proxy controls access to an object that is expensive to instantiate.
* A Protection Proxy controls access to the methods of an object based on the caller.
* Many other variants of the Proxy Pattern exist including caching proxies, synchronization proxies, firewall proxies, copy-on-write proxies and so on.
* Proxy is structurally similar to Decorator, but the two differ in their purpose.
* The Decorator Pattern adds behavior to an object, while a Proxy controls access.
* Java’s built-in support for Proxy can build a dynamic proxy class on demand and dispatch all calls on it to a handler of your choosing.
* Like any wrapper, proxies will increase the number of classes and objects in your design.