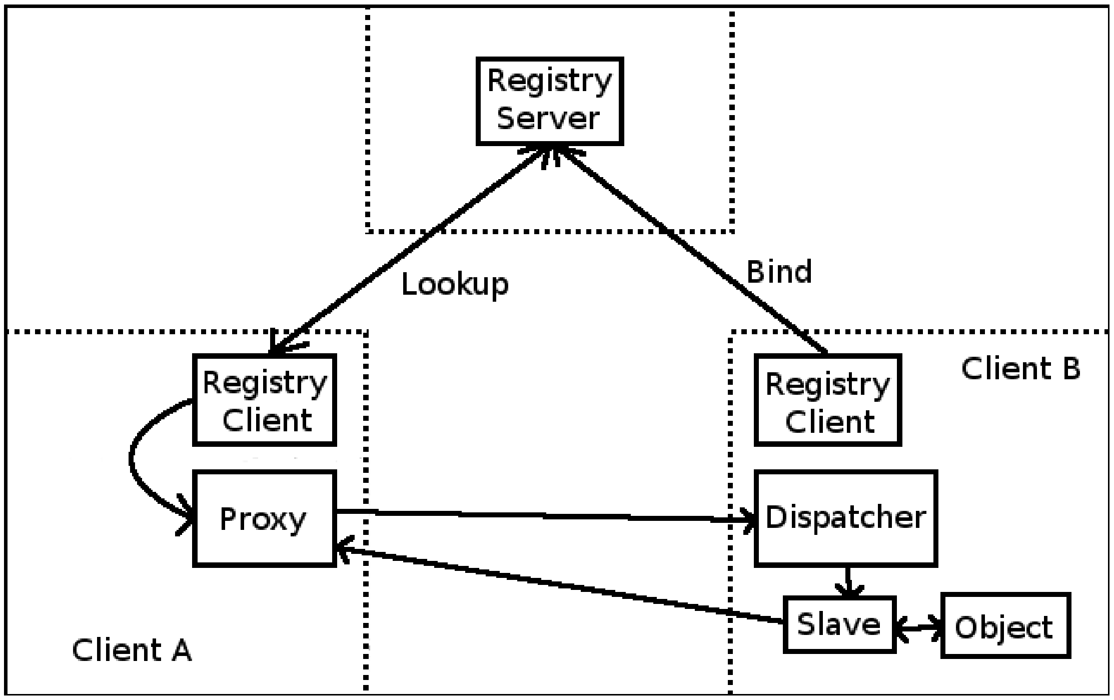
MichaelMichael RMI System

Michael Hansen (mhansen1) & Michael Rosen (mrrosen)
March 5, 2013

RMI Design Overview:

This system utilizes a number of large structures to implement successful remote method invocations. First, the registry, which stores a list of all remote objects bound to it by users. Second, a local server (the RMI class), used to store a local table of local objects which are visible to the outside as well as a dispatcher for listening for and processing any remote object invocations that come in. Finally, remote objects themselves (once dereferenced are referred to as proxies), which sent invocations back to the dispatcher on their JVM of origin. Below is a diagram which illustrates the overall working of the system:



The diagram shows a simplified diagram of how our RMI implementation works. Let's say that Client A wants to run a method using a remote object on Client B. First Client B must have placed the object's information in the registry server using bind, then Client A can do a look up to get that information, using the RegistryMessage class to communicate back and forth. The information returned by the look up is converted by the RMIRegistryClient object into a RemoteObjectRef object, which is then localised into a Proxy with InvocationHandler RMIProxyHandler. This handler marshalls all the method invocations into RMIMessage objects and sends them to Client B's dispatcher (RMIProxy). The dispatcher then unmarshalls the message and spawns a thread to handle the local function call (RMIProxySlave). The slave dereferences any remote object references passed as arguments to the method, finds the target object in the local remote object table, and executes the method. Then it converts the return values to remote object references and marshalls them into a RMIMessage, which it then sends back to Client A's proxy. The proxy then unmarshalls the message, dereferences any remote object references, and either returns the result or throws returned exceptions. Also, note the dotted lines indicate separate threads, not necessarily separate machines, though this could be the case.

User's View of this RMI System:

For an application-level user, this RMI system is designed to be easy to use. Once a registry is set up and running, user applications need only meet a few requirements for this system to function properly, and can use the system with only a few important functions. Usage of this system can be viewed in the provided tests. Below is a documentation-style report of these functions:

RMI:

Description:

This class is used for remote object invocations. In order to use this class, a RMI registry (RegistryServer class) must be running and the host and port of this registry must be known. Once this class is constructed using the host and port of a registry, the object can be used to add and remove objects from the registry. The registry can also be queried for an object reference stored within it or for all objects registered. Any object to be added to the registry must meet the following specifications:

- The object must implement the Remote440 interface, possibly through inheritance
- The object must have an interface that implements the Remote440 interface and contains all methods that can be remotely invoked. Any other methods will not be invokable from remote clients.
- All methods to be remotely invoked must take arguments and have return values that implement either the Remote440 interface or the Serializable interface. Should a method that does not meet this require be invoked remotely, a RemoteException will be thrown.

When invoking a remote method, exceptions resulting from the method call are fed back to the local method invocation, thus the user must handle these exceptions. Should an error occur in the process of performing a remote method invocation, a RemoteException will be fed back to the local method invocation and must too be handled by the user.

Methods:

```
void bind(String name, Remote440 obj) throws Exception
```

Adds a reference to Remote440 `obj` to the registry with key `name`. If an object is already in the registry by the given name, an `AlreadyBoundException` is thrown. If the registry fails to update the registry due to a null object or key, a `NullPointerException` is thrown.

```
void rebind(String name, Remote440 obj) throws Exception
```

Adds a reference to Remote440 `obj` to the registry with key `name`. If an object is already in the registry by the given name, the name is remapped to the new object. If the registry fails to update the registry due to a null object or key, a `NullPointerException` is thrown.

`void unbind(String name) throws Exception`

Removes the reference from the registry with key `name`. If the key is not in the registry, an `NotBoundException` is thrown. If the registry fails to update the registry due to a null key, a `NullPointerException` is thrown.

`Object lookup(String name) throws Exception`

Returns a dereferenced remote object refernece (proxy) from the registry with key `name`. If the key is not in the registry, an `NotBoundException` is thrown. If the registry fails to update the registry due to a null object, a `NullPointerException` is thrown.

`String[] list()`

Returns an array of all names in the registry. These names can be used to look up objects in the registry or ensure a name is currently unused.

Unimplemented Features:

This RMI system does not currently implement any form of garbage collection for remote objects no longer referenced in the local table. This RMI system does not currently implement class retrieval should a client not have the class files needed to instantiate an object. This RMI system does not include an interface generator for creating interfaces for classes that do not have them, thus requiring classes to implement such interfaces.

These would all be possible to implement, but this system does not include them as they are not part of the requirements nor did time allow them to be added. All of them would further benefit the user and we apologize for their absence.