Remote Method Invocation (RMI)

Introduction to RMI

- Overall goals
  - Distribute tasks across the network
  - Greater scalability and availability
  - Transparent programming model
  - Interoperability with existing technologies
- Java is great fit for distributed computing
  - Platform neutral
  - Flexible security model
  - Designed with network in mind

Overview

- We want to make method calls to an object on a remote system. To do this:
  - Create an interface with the needed methods
  - Write a class implementing the interface
  - Perform a few steps to set up glue
  - Make a connection
  - Interact with the remote object as if it were local

What Do We Need?

- Objects accessible across the network
  - Encapsulation - achieved using interfaces
  - Marshaling of parameters across the network
    - Simple types
    - Complex objects
    - Remote interfaces
  - Way to obtain object's interface
  - Ability to advertise object's interfaces
  - Runtime management

Passing Parameters in RMI

- Simple types are passed as is
- Objects are serialized across the network and reference on client side has nothing to do with server side
- For remote objects stubs are created on client side
- Non-serializable and non-remote objects can't be passed as parameters in RMI

RMI Basic Elements

- Remote object
  - Implementation of remote methods
  - Dealing with threads (not thread safe)
- Registry
  - Naming
    - Urls are used for remote objects (`rmi://`)
  - Locating
    - Must dispatch URL to a correct server
  - Registering
    - Server objects advertise their availability

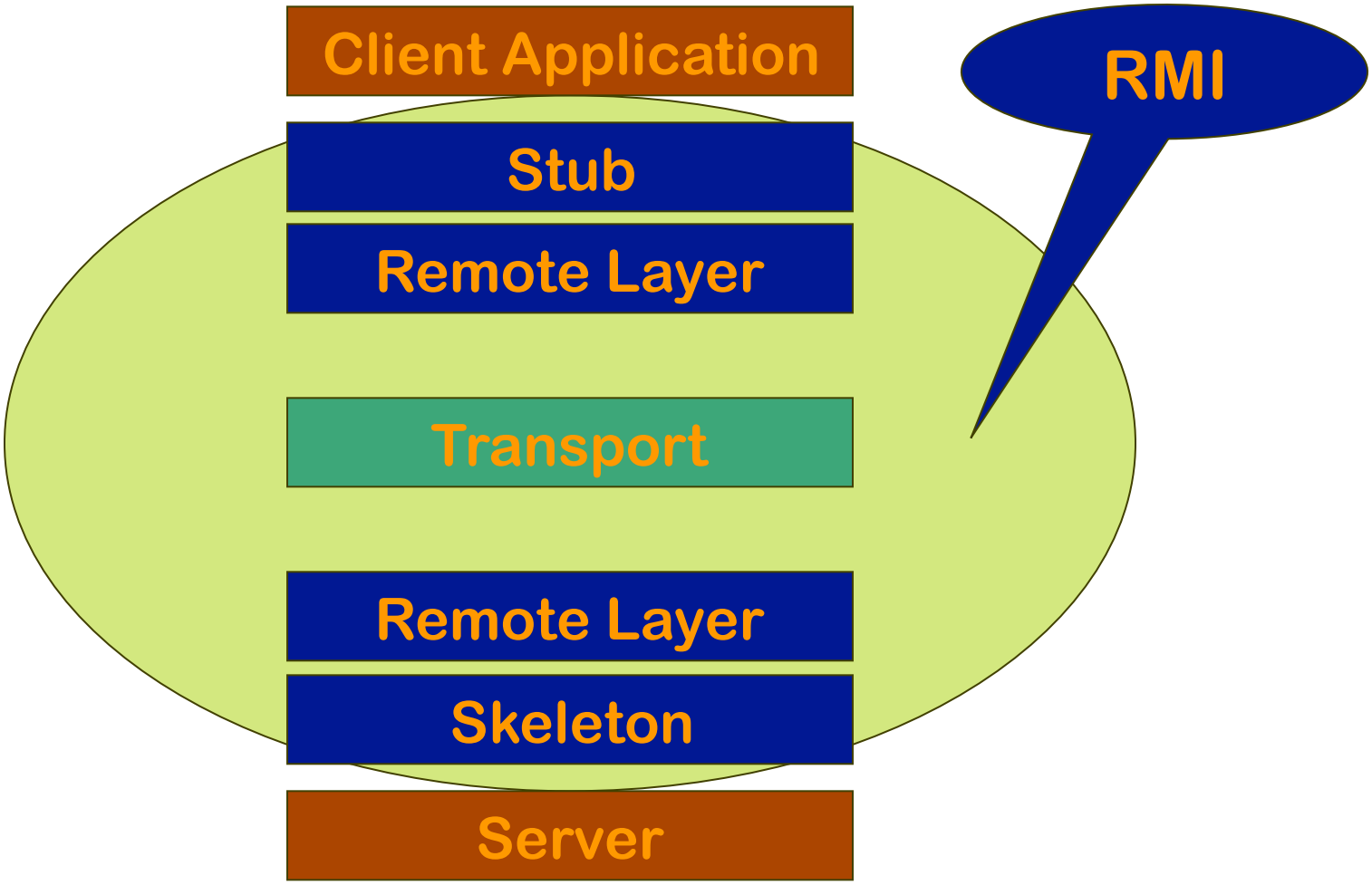
RMI Server

- Lifetime management
- Garbage collection
- Resource pooling
  - Reuse connections
- Thread management
  - All requests are free threaded

RMI Server

- Transport
  - Good abstraction - socket factory
- Marshaling
  - Serialization

RMI Layers


Client Application
RMI
Stub
Remote Layer
Transport
Remote Layer
Skeleton
Server

Remote Method Invocation

- Remote Object
  - Implements `java.rmi.Remote` interface
  - Only remote methods can be accessed by a client
  - Can implement more than one remote interface
  - Never accessed directly by a client, stub is used instead
  - Made available (exported) by server
  - Advertised using registry

Basic Algorithm

- Server
  - Create `RMISecurityManager` (optional)
  - Initialize remote object
  - Advertize this object
    - Prepare to accept connections
    - Associate URL with this object
- Client
  - Locate the object
  - Get a reference for this object's stub
  - Call remote method

Extending Remote Interface

- **Interface and implementation object**

```
public interface Basket extends Remote
{
   int add(String sFruit, int c)
   throws RemoteException;

   void remove(String sFruit, int c)
   throws RemoteException;

   void hangOnTree(String sTree);
}
```

Each remote method must be declared to throw RemoteException

Implementing the Remote Object

```
public interface Basket extends Remote
{
   int add(String fruit, int c)
   throws RemoteException, BasketException;

   void remove(String fruit, int c)
   throws RemoteException;
}
```

- add() **and** remove() **are remote methods**

Implementing Remote Object

```
public class BasketImpl extends UnicastRemoteObject
                        implements Basket {
   public BasketImpl()
   throws RemoteException
   { ... }

   public int add( String fruit, int c )
   throws RemoteException, BasketException
   { ... }

   public int remove( String fruit, int c )
   throws RemoteException
   { ... }

   public void hangOnTree(String tree) {
      // local methods are legal
   }
```

Obtaining a Registry

- Server locates an existing registry, or creates one in process
  - Default registry port is 1099

- Locating a registry

```
Registry reg = LocateRegistry.getRegistry(host,1099);
```

- Creating a registry

```
Registry reg = LocateRegistry.createRegistry(1099);
```

Binding the Remote Object

```
    public static void main( String args[] ) {
        String SERVER_NAME = "BasketServer";
        System.setSecurityManager(new RMISecurityManager());
        try {
            BasketImpl server = new BasketImpl();
            Registry reg = LocateRegistry.createRegistry(1099);
            reg.rebind(SERVER_NAME, server );
        }
        catch( Exception ex ) {
            System.out.println( ex.getMessage() );
            ex.printStackTrace();
        }
    }
}
```

RemoteException

- **Extends from** `IOException`
- **Wraps other exceptions that happen inside your method (even your subclass)**
- **Base class for all remote exceptions**

Getting a Remote Object

- Client looks up remote object

```
String host = "localhost";
String url = "//" + host + "/" +   SERVER_NAME;
Basket server = (Basket)Naming.lookup(url);
```

- Once connected other Remote objects can be passed as arguments

exportObject

- UnicastRemoteObject.exportObject(…)
  - **When a remote object cannot subclass** UnicastRemoteObject
  - **Implement a Remote interface**
  - **Call** UnicastRemoteObject.exportObject(…)
  - **Allows passing as a remote object**

Let's take a break!

RMI Cookbook

- **Write a** `Remote` **interface**
  - **Derive from** `java.rmi.Remote`
  - **All methods throw** `java.rmi.RemoteException`
- **Write a remote server**
  - **Derive from** `UnicastRemoteObject`
  - **Implement** `Remote` **interface**

RMI Cookbook II

- Write Code to Register the Server
- Write the client
  - Lookup server
  - Cast as remote interface
- Start the server
- Start the client

RMI Security

- Client may load classes from server
- Server may load classes from client
- Start command.

```
java –Djava.rmi.server.codebase=http://host/classes/
     -Djava.security.policy=net.policy ClassToRun
```

  - codebase - where this client/server loads its code from
  - policy - specifies the policy file containing the permissions granted to code bases

Policy Files

- Allow socket connections
- Allow access to codebase

```
grant {
   permission java.net.SocketPermission
               "*:1025-65535", "connect,accept";
   permission java.net.SocketPermission
              "*:80", "connect";
   permission java.io.FilePermission
              "d:\\classes\\", "read";
};
```

RMI Callback

- `Remote` objects can be passed as parameters
- Allows the server to make calls on the client
- Objects which are passed do not have to bind to a name server

RMI Magic

- **RMI server creates an instance object**
- `UnicastRemoteObject`**'s constructor exports the object**
  - **Makes object available to service incoming RMI calls**
  - **Binds to an arbitrary TCP socket**
  - **Creates thread which listens for incoming requests**

RMI Magic

- Server registers object with the registry
  - Gives registry the stub for that object
  - Contains information needed to locate object on server
    - *Hostname*
    - *Port*
- A client obtains stub with a registry lookup
- Registry finds stubs using codebase

RemoteRef.invoke()

- **Client issues a remote method invocation**
- **Stub class provides a** `RemoteRef`
- **Marshals the arguments over the connection**
- **Uses a subclass of** `ObjectOutputStream`
  - **Knows how to deal with objects implementing** `java.rmi.Remote`
  - **Other objects serialized normally**

RMI Magic

- Client connects to the server socket
- Server creates a new thread to deal with the request
- Original thread continues listening
- Server reads the header information and unmarshals the RMI argument

RMI Magic

- Server calls skeleton class's "*dispatch*" method
- Skeleton calls appropriate method on the object
- Sends result back across the wire using the same `RemoteRef`
- Any exceptions are caught and sent instead of the return value

RMI Magic

- On client side return value is un-marshaled
- Returned from the stub back to client code
- Exception are un-marshaled and re-thrown from the stub

Synchronization

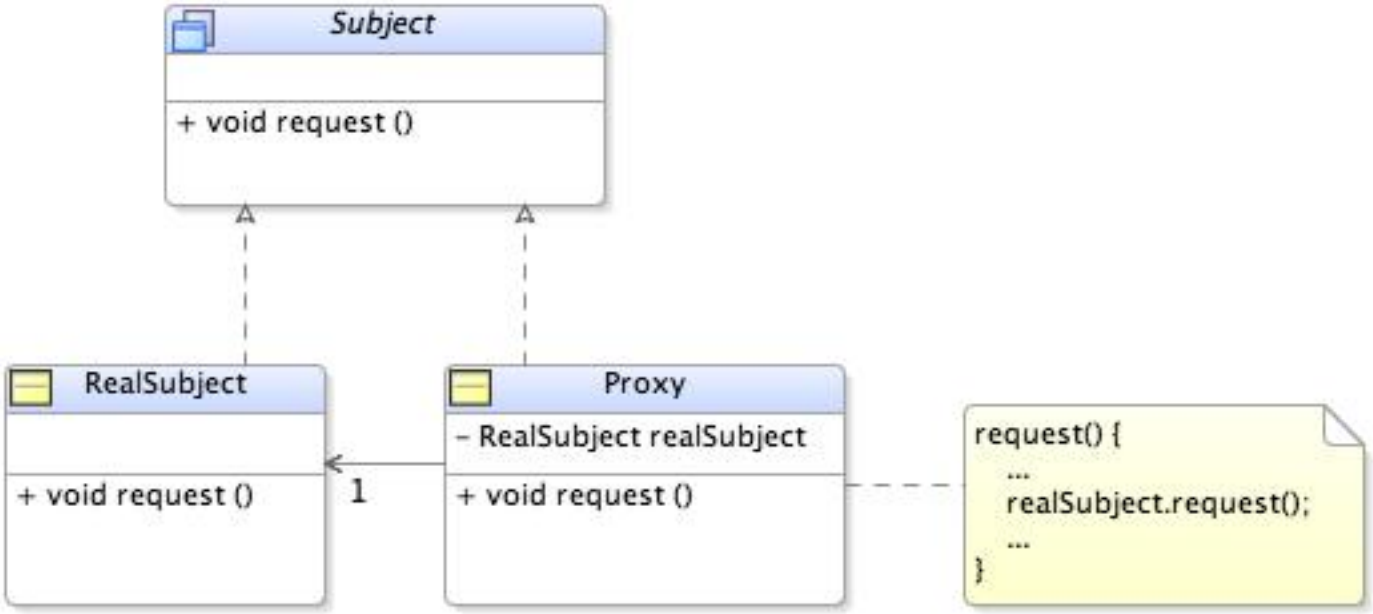
- No synchronization provided by the server side components
- Responsible for your own thread-safety

Socket Connections

- Server
  - Single `UnicastRemoteObject`
  - One thread listening for connections
  - Multiple threads dealing with method calls
- Client
  - One socket to the server per client (minimum)
  - Multiple stubs pointing to the same server object may multiplex multiple virtual connections over a single concrete connection

Proxy Pattern

- Use a proxy wherever a "smart" reference is required



Using IIOP

- Internet Inter-Orb Protocol (IIOP)
  - Protocol used by CORBA objects
  - Allows some level of interoperability between CORBA and RMI

javax.rmi.PortableRemoteObject

- **Use as base class for server rather than** `java.rmi.RemoteObject`
- **Use** `narrow()` **rather than simple cast**

```
Object narrow( Object, Class )
```

- **Use use** `rmic` **comiler to compile stubs/ skeletons using** `-iiop` **option**

Using JNDI

- Java Naming and Directory Service
  - Part of and used extensively by J2EE
  - `javax.naming`
- Standard interface
- RMI Registry service provider available from Sun
  - `http://java.sun.com/products/jndi`