An Introduction to Java RMI

Agenda

- Remote Interface
- RMI Registry
- Exporting a remote object (RMI server)
- Using a remote object (RMI client)
- RMI callbacks

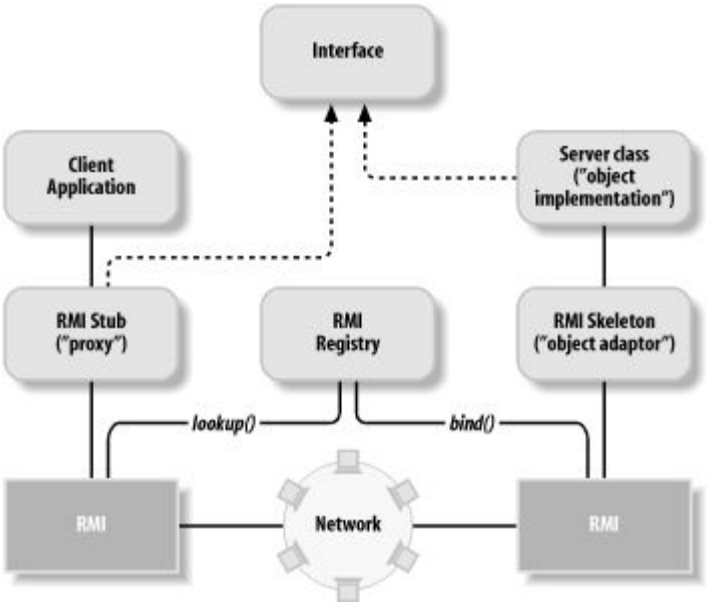
References:

- The Java Remote Method Invocation API
- Package java.rmi Documentation

RMI Architecture

There are 4 main entities:

1. The **interface** that specifies the remote resource
2. The **server** that implements the remote object
3. The **client** that uses the remote object
4. The **registry** that provides access to remote objects



What Is The Registry?

- The registry is a naming service that maps the symbolic names of remote objects to their reference/stub
- A server must publish its remote objects on a registry using a symbolic name, i.e., must perform a “bind operation”
- A client must look the remote object up in the registry using its symbolic name

The Interface

- The interface specifies the contract, i.e., the methods a client can invoke on the remote object
- For any object that we want to make accessible through the network we must define an interface that extends the interface java.rmi.Remote
- The code of the interface we define must be accessible to the client, thus, it can operate on the remote object

The Remote Interface

The java.rmi.Remote is a special interface that has no method, but that only indicates that its instances can be invoked remotely by another JVM

The RMI framework automatically generates the stub and the skeleton for instances of java.rmi.Remote

An Example: Remote Calendar

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RCalendar extends Remote {
        String today() throws RemoteException;
}
```

Remote Methods

- Each method of a remote interface must declare that it can raise a java.rmi.RemoteException in its throws clause
  - It is the superclass of communication-related exceptions that may occur during the execution of a remote method call

- All parameters and return values must be either primitives (int, double, etc.), or implement java.io.Serializable or java.rmi.Remote

Steps For Implementing a RMI Server

1. Implement the remote class
    - A constructor
    - Remote methods
2. Create and install a security manager (not here)
3. Create an instance of the remote object
4. Register the remote object with the RMI remote registry

The Remote Class

- This class implements the remote interface and its methods
- Typically, it extends the java.rmi.server.UnicastRemoteObject
- java.rmi.server.UnicastRemoteObject exports a remote object generating the stub that perform the communication
- Stubs are either generated at runtime using dynamic proxy objects, or they are generated statically at build time

Our Example

```
public class ServerCalendar extends UnicastRemoteObject implements RCalendar {

        protected ServerCalendar() throws RemoteException {
                super();
        }

        @Override
        public String today() throws RemoteException {
                Date now = new Date();
                return now.toString();
        }
}
```

How To Export a Remote Object

We can export and generate the stub for a remote object by

- Subclassing UnicastRemoteObject and calling the UnicastRemoteObject() constructor
- Subclassing UnicastRemoteObject and calling the UnicastRemoteObject(port) constructor
- Calling the exportObject(Remote, port) method

For other approaches, see UnicastRemoteObject documentation

Publishing The Remote Object

- We choice a symbolic name that uniquely identify our remote object (a Java String object)
- The symbolic name must be shared with the client
- We create an instance of our object using a constructor
- We get a reference to the registry and create a mapping between the symbolic name and our object
- In the Java code a registry is an instance of Registry interface

Our Example

```
public class Server {

        public static void main(String[] args) {
                try {
                ServerCalendar cal = new ServerCalendar();
                Naming.rebind("mycalendar", cal);
                }catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Naming class

- This class provides methods for storing and obtaining references to remote objects in a remote object registry
- Each method takes a URL of the form:

//host:port/name

where name is the symbolic name of the remote object; if host is omitted, the default is the local host; if port is omitted, the defaults is 1099

Static Methods in Naming Class

- void bind(String name, Remote obj): Binds the specified name to a remote object
- void rebind(String name, Remote obj): Rebinds the specified name to a new remote object
- void unbind(String name): Destroys the binding for the specified name that is associated with a remote object
- Remote lookup(String url): Returns a reference, a stub, for the remote object associated with the specified name

Get a Reference to a Registry

The class java.rmi.registry.LocateRegistry can be used to

- obtain a reference to a registry on a particular host (including the local host) -- similar to Naming
- create a registry that accepts calls on a specific port -- the registry will be create on the same host of the server

Methods in LocateRegistry

- Registry createRegistry(int port): Creates and exports a Registry instance on the local host that accepts requests on the specified port
- Registry getRegistry(): Returns a reference to the the remote object Registry for the local host on the default registry port of 1099
- Registry getRegistry(String host, int port): Returns a reference to the remote object Registry on the specified host and port

For further methods, see LocateRegistry documentation

Our Example

```
public class Server {

        public static void main(String[] args) {
                try { Registry reg = LocateRegistry.createRegistry(2018);
                ServerCalendar cal = new ServerCalendar();
                reg.rebind("mycalendar", cal);
                }catch(Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Note

- After binding the remote object on the registry the main method terminates
- However the server is still running since a thread is in charge of managing the requests from the clients

rmiregistry command

A registry can be started by using rmiregistry command from a terminal

rmiregistry [port]

Note: rmiregistry needs the interface and class files, thus, pay attention to the paths

Steps For Implementing a RMI Client

1. Locate the registry
2. Look the remote object up by using its symbolic name
3. Use the remote object like standard ones

Our Client (registry on localhost:1099)

```
import java.rmi.Naming;
import common.RCalendar;

public class Client {
        public static void main(String[] args) {
          try {
              RCalendar cal = (RCalendar)Naming.lookup("mycalendar");
              System.out.println("Now: " + cal.today());
          }catch(Exception e) {
              e.printStackTrace();
          }
        }
```

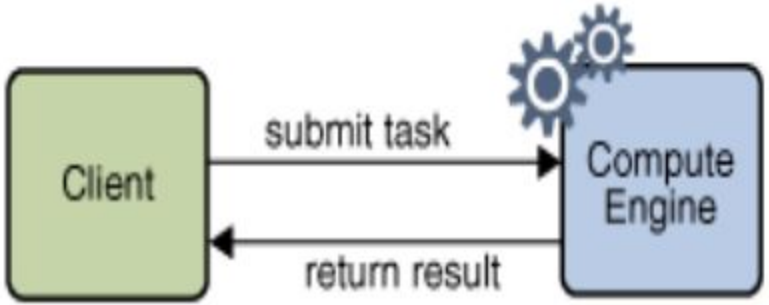
Our Client (registry on a custom port)

```
public class Client {

    public static void main(String[] args) {
        try {
        Registry reg = LocateRegistry.getRegistry("localhost", 2018);
        RCalendar cal = (RCalendar)reg.lookup("mycalendar");
        System.out.println("Now: " + cal.today());
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
```

Example: A Compute Engine

- There is a server with large computational capabilities that are available to remote users
- Clients connect to the server and submit tasks and wait for the results



The Remote Engine

```
public interface RemoteEngine extends Remote {
   public Integer submit(Task task, Integer data) throws RemoteException;
   public final static String ENGINE_NAME = "rengine";
}
```

- It allows us to submit a new task together with its input and to get the result
- The symbolic name of the remote object is a constant in the interface

The Task Interface

```
import java.io.Serializable;

public interface Task extends Serializable {
    public Integer execute(Integer n);
}
```

Exercise: generalize the interface to further types not only Integer

The Remote Service

```
public class TaskEngine extends UnicastRemoteObject implements RemoteEngine {

    public TaskEngine() throws RemoteException { super(); }

    @Override
    public Integer submit(Task task, Integer data) throws RemoteException{
        System.out.println("New task received");
        Integer result = task.execute(data);
        System.out.println("Task completed. Result: " + result);
        return result;
    }
}
```

The Server

```
public class Server
{
   public static void main( String[] args ) throws IOException
   {
      Registry reg = LocateRegistry.createRegistry(2018);
      RemoteEngine engine = new TaskEngine();
      reg.rebind(RemoteEngine.ENGINE_NAME, engine);
      System.out.print("Server started");
   }
}
```

The Client

```
public class Client {

   public static void main(String[] args) throws Exception{
        Registry reg = LocateRegistry.getRegistry(2018);
        RemoteEngine engine =
(RemoteEngine)reg.lookup(RemoteEngine.ENGINE_NAME);
        for(Task t : tasks) {
                int result = engine.submit(t, 5);
                System.out.printf("The task \"%s\" results in %d\n", t, result);
        }
   }
}
```

An Example of Task

```
new Task() {

        public Integer execute(Integer n) {
                int result = 0;
                while(n > 0) {
                        result += n;
                        n--;
                }
                return result;
        }
}
```

RMI Callback

- So far now we used RMI to implement a **synchronous request-reply** communication: the client **waits** until the remote invocation terminates
- We can implement an **asynchronous** communication pattern using the **callback** mechanism:
  - the client invokes a remote method and passes a callback to the server (the method invocation terminates immediately)
  - the server uses the callback to notify the client that some event occured

What Is a Callback?

- A remote object provided by the client (instance of the Remote interface)
- The client and the server must agree on the interface of the callback
- The callback is implemented by the client and it is sent to the server as parameter in an remote invocation
- Actually, Java sends a stub to the remote object in the client

Example

- We want to create a centralized chat application
- The client logs in the system and sends a callback to the server
- The server uses the callback to notify the client when a new user connects and to send messages

The Server Interface

```
public interface ChatServer extends Remote {
        public void login(String name, ChatClient callback) throws RemoteException;
        public void logout(String name) throws RemoteException;
        public void sendMessage(Message msg) throws RemoteException;

        public final static String OBJECT_NAME = "rmichat";
}
```

The Client Callback

```
public interface ChatClient extends Remote {
        public void userLogin(String name) throws RemoteException;
        public void userLogout(String name) throws RemoteException;
        public void receiveMessage(Message msg) throws RemoteException;
}
```

Registering Clients (Server)

```
public class RmiServer extends UnicastRemoteObject implements ChatServer {
    private Map<String, ChatClient> clients;

    @Override
    public synchronized void login(String name, ChatClient callback) throws
RemoteException {
        for(String n : clients.keySet())
            clients.get(n).userLogin(name);
        clients.put(name, callback);
    }
```

Receiving messages (Client)

```
public class RmiClient extends UnicastRemoteObject implements ChatClient {

    @Override
    public synchronized void receiveMessage(Message msg) throws
RemoteException {
        System.out.printf("[%s] says \"%s\"\n", msg.sender, msg.text);
    }
```

Further RMI Related Topics

- Security & security manager

https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/rmi_security_recommendations.html

- Customized sockets

https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/socketfactory/index.html

Conclusion

- Remote Interface
- RMI Registry
- Exporting a remote object (RMI server)
- Using a remote object (RMI client)
- RMI callbacks

References:

- The Java Remote Method Invocation API
- Package java.rmi Documentation