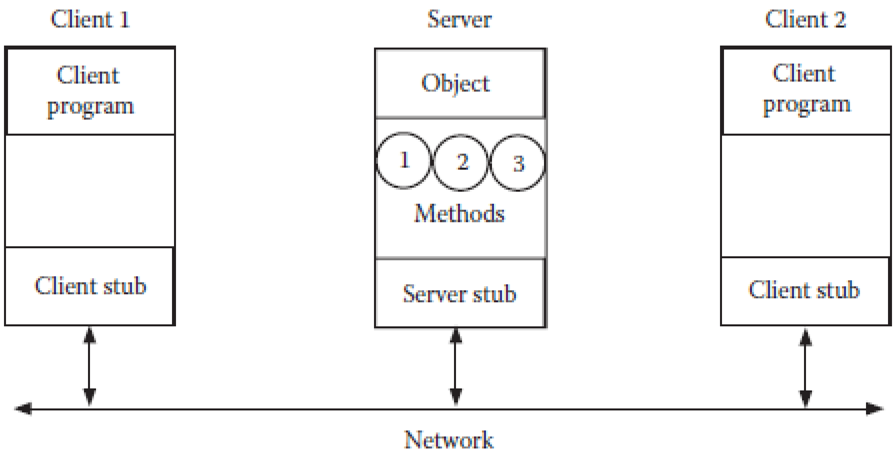
RMI

En este documento se describe el proyecto a ser ejecutado en la versión de Java "1.7.0_01" (o posterior) o en cualquier otro lenguaje.
Java(TM) 2 Runtime Environment, Standard Edition (build 1.7.0_01-b08)
Java HotSpot(TM) Client VM (build 1.7.0_01-b08, mixed mode)

Como herramienta de trabajo se describe la implementación con JCreator 5

El proyecto se ha dividido en dos partes:

1) Implementación del ejercicio propuesto en el tutorial de RMI para la versión descrita de Java.
2) La implementación de un modelo de Chat para ser sofisticado a un sistema Publish/Subscribe más complejo (ver el documento Publish subscribe) y del cual debe entregarse un reporte escrito.



Ejemplo 1. Hello World! usando RMI

Tomado de la página de Oracle:
http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html

Este tutorial se refiere al armado de un servidor que inicialmente proporciona una cadena de texto como respuesta a la solicitud de ejecución de un método. El código se compone de tres partes:

```
//Interfaz (remota):
//==================================================================
package example.hello;

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

public interface Hello extends Remote {

      String sayHello() throws RemoteException;
}
```

```
//==================================================================
Código del Servidor (remoto):
//==================================================================
package example.hello;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server implements Hello {

    public static int serviciosPrestados;

    public Server() {}

    public String sayHello() {
            return "Hello, world!";
    }

    public static void main(String args[ ]) {
      String downloadLocation = new
String("file:C:/Users/mmejiao/Documents/ JCreator%20LE/MyProjects/rmi/");
      try {
            System.setProperty("java.rmi.server.codebase", downloadLocation);
            Server obj = new Server();
            Hello stub = (Hello) UnicastRemoteObject.exportObject(obj,0);
            Registry registry = LocateRegistry.getRegistry();
            registry.rebind("Hello", stub);
            System.err.println("Server ready");
      } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
      }
    }
}
```

```
//============================================================
// Código del Cliente (local):
//============================================================
package example.hello;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class Client {

      private Client() {}

      public static void main(String[] args) {
       String host = (args.length < 1) ? null : args[0];
       try {
             Registry registry = LocateRegistry.getRegistry(host);
             Hello stub = (Hello) registry.lookup("Hello");
             String response = stub.sayHello();
             System.out.println("response: " + response);
       }
       catch (Exception e) { // RemoteException
             System.err.println("Client exception: " + e.toString());
             e.printStackTrace();
       }
    }
}

//============================================================
```

Pasos para JCreator 5

JC1) En JCreator, asegúrese que la versión de la distribución de Java que está ejecutando es la apropiada (> 1.7.0_01). Para ello, revise la versión referenciada en la opción:
Configure - Options - JDK Profiles.
En caso de requerirlo, modifique la distribución de Java referenciada.

JC3) En un nuevo espacio de trabajo, cree un nuevo proyecto con el nombre "rmi" (Java acepta / como separador en la trayectoria independientemente del sistema operativo "Usuarios" = "Users" y "Mis Documentos" = "Documents").

Programe la interfaz, el servidor y el cliente.
Construya el proyecto. El espacio de trabajo tiene terminación .jcw

JC4) Ubique el rmiregistry:
Asegúrese que el *PATH* pasa por el directorio bin de la distribución de Java. Por ejemplo C:\Archivos de programa\Java\jdk1.7.0_04\bin

En caso contrario, cambie el valor de la variable de entorno *PATH*: Equipo – Propiedades – Configuración avanzada del sistema – Variables de entorno.
http://docs.oracle.com/javase/tutorial/essential/environment/paths.html

JC4) Inicie el registry en el servidor:
Abra una ventana de DOS (ejecutar cmd)
Ejecute start rmiregistry

Esto inicializará otra ventana de DOS donde se estará ejecutando el registro rmi (por default escucha en el puerto 1099).

JC5) Ejecute el servidor (desde JCreator), se debe quedar "corriendo" en su ventana.

JC6) Ejecute el cliente, desde otra instancia de JCreator o desde otra máquina (pasando el hostname del servidor, ej. 113DAC27A.itam.mx o 148.205.199.118). Ejecuta y termina, solamente escribe el letrero: Hello, world!.

JC7) Detenga el servidor y agréguele un contador de "servicios prestados". El valor del contador debe mandarse en el letrero de respuesta.

Ejemplo 2. Chat usando Publish-subscribe

Este tutorial define una situación simétrica entre el servidor del chat y los clientes. Los clientes mandan un objeto remoto al servidor del chat para que se les avise cuando hay mensajes. El sistema trabaja como “subscripción – publicación”.

En este caso tendremos dos interfaces, la del servidor y la del cliente. Como clases concretas se tiene el servidor, el cliente y una clase para el mensaje a ser transferido.

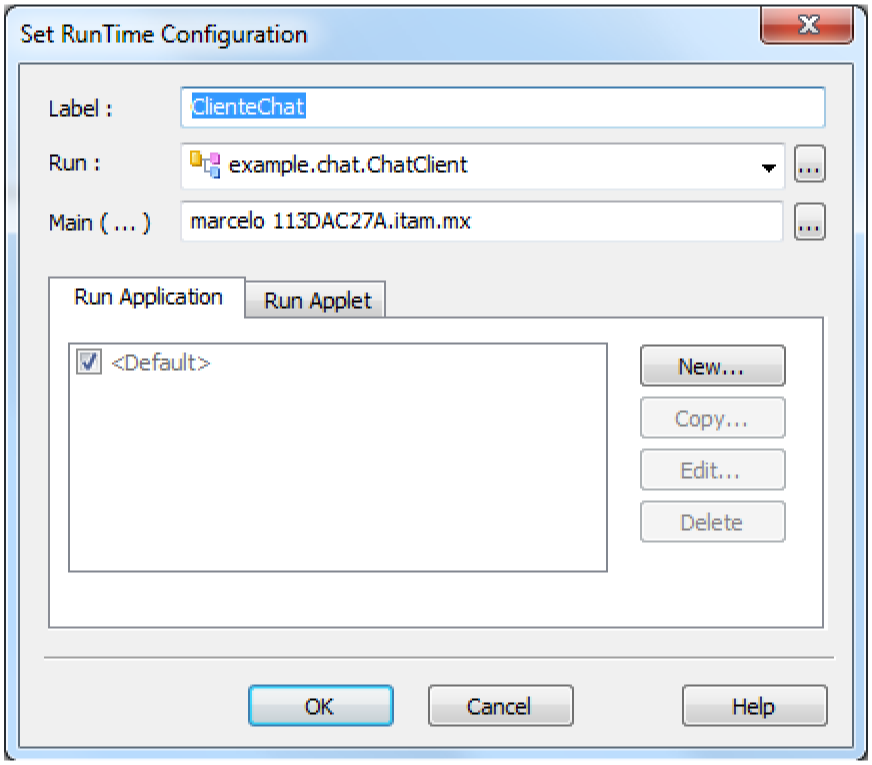
```
//================================================================
// Mensaje
//================================================================
package example.chat

import java.io.*;

public class Message implements Serializable {
   public String name;
   public String text;

  public Message(String name, String text) {
      this.name = name;
      this.text = text;
    }

}

//================================================================
// Interfaz Cliente Chat  - para callback
//================================================================
package example.chat

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

public interface IChatClient extends java.rmi.Remote {
  void receiveEnter(String name) throws RemoteException;
  void receiveExit(String name)throws RemoteException;
  void receiveMessage(Message message) throws RemoteException;
}

//================================================================
// Interfaz Servidor Chat
//================================================================
package example.chat

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

public interface IChatServer extends java.rmi.Remote {
  void login(String name, IChatClient newClient) throws RemoteException;
  void logout(String name)throws RemoteException;
  void send(Message message) throws RemoteException;
}
```

```
//===============================================================
// Servidor Chat
//===============================================================
package example.chat;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class ChatServer extends UnicastRemoteObject implements IChatServer {

	Hashtable<String, IChatClient> chatters = new Hashtable<String, IChatClient>();

	public ChatServer() throws RemoteException  {}

	public synchronized void login(String name, IChatClient nc) throws RemoteException {
		chatters.put(name,nc);
		Enumeration entChater = chatters.elements();
		while( entChater.hasMoreElements() ){
			((IChatClient) entChater.nextElement()).receiveEnter(name);
		}
		System.out.println("Client " + name + " has logged in");
	}

	public synchronized void logout(String name) throws RemoteException {
		System.out.println("Client " + name + " has logged out");
		Enumeration entChater = chatters.elements();
		while( entChater.hasMoreElements()) {
			((IChatClient) entChater.nextElement()).receiveExit(name);
		}
		chatters.remove(name);
  }

	public synchronized void send(Message message) throws RemoteException {
		Enumeration entChater = chatters.elements();
		while( entChater.hasMoreElements() ) {
			((IChatClient) entChater.nextElement()).receiveMessage(message);
		}
		System.out.println("Message from client "+message.name+":\n"+message.text);
	}

	public static void main( String[] args ){
		String downloadLocation = new
String("file:C:/Users/mmejiao/Documents/ JCreator%20LE/MyProjects/rmi/");
		String serverURL = new String("///ChatServer");
		try {
			System.setProperty("java.rmi.server.codebase", downloadLocation);
			ChatServer server = new ChatServer();
			Naming.rebind(serverURL, server);
			System.out.println("Chat server ready");
		}
		catch(Exception ex){
			ex.printStackTrace();
		}
	}

}
```

```
//=================================================================
// Cliente Chat
//=================================================================
package example.chat;

import java.rmi.*;
import java.rmi.server.*;
import java.io.*;

public class ChatClient extends UnicastRemoteObject implements IChatClient {

	public String name;
	IChatServer server;
	String serverURL;

	public ChatClient( String name, String url ) throws RemoteException {
		this.name = name;
		serverURL = url;
		connect();
	}

	private void connect() {
		try {
			server=(IChatServer) Naming.lookup("rmi://"+serverURL+"/ChatServer");
			server.login(name, this); // callback object
		}
		catch( Exception e ) {
			e.printStackTrace();
		}
	}

	private void disconnect() {
		try {
			server.logout(name);
		}
		catch( Exception e ) {
			e.printStackTrace();
		}
	}

	private void sendTextToChat(String text) {
		try {
			server.send(new Message(name,text));
		}
		catch( RemoteException e ) {
			e.printStackTrace();
		}
	}

	public void receiveEnter( String name ) {
		System.out.println("\nLog in "+name+"\n"+this.name+" -- Cadena a enviar: ");
	}

	public void receiveExit( String name ) {
		System.out.println("\nLog out " + name + "\n");
		if ( name.equals(this.name) )
			System.exit(0);
```

```
            else
                  System.out.println(this.name + " -- Cadena a enviar: " );
      }

      public void receiveMessage( Message message ) {
                  System.out.println
('\n'+message.name+":\n"+message.text+"\n"+name+" -- Cadena a enviar: ");
      }

      public static String pideCadena(String letrero) {
            StringBuffer strDato = new StringBuffer();
            String strCadena = "";
            try {
                  System.out.print(letrero);
                  BufferedInputStream bin = new BufferedInputStream(System.in);
                  byte bArray[] = new byte[256];
                  int numCaracteres = bin.read(bArray);
                  while (numCaracteres==1 && bArray[0]<32)
                        numCaracteres = bin.read(bArray);
                  for(int i=0;bArray[i]!=13 && bArray[i]!=10 && bArray[i]!=0; i++) {
                        strDato.append((char) bArray[i]);
                  }
                  strCadena = new String( strDato );
            }
            catch( IOException ioe ) {
                  System.out.println(ioe.toString());
            }
            return strCadena;
      }

      public static void main( String[] args ) {
            String strCad;
            try {
                  ChatClient clte = new ChatClient( args[0],args[1] );
                  strCad = pideCadena(args[0] + " -- Cadena a enviar: ");
                  while( !strCad.equals("quit") ) {
                        clte.sendTextToChat(strCad);
                        strCad = pideCadena(args[0]+" -- Cadena a enviar: ");
                  }
                  System.out.println("Local console "+clte.name+", going down");
                  clte.disconnect();
            }
            catch( RemoteException e ) {
                  e.printStackTrace();
            }
      }

}
//=================================================================
```

El procedimiento para hacer funcionar ese ejemplo es similar al del primer ejemplo. La única diferencia está en que deben pasarse argumentos forzosamente al ClienteChat (nombreCliente serverURL). Y serverURL es 113DAC27A.itam.mx


Set RunTime Configuration

Label : ClienteChat

Run : example.chat.ChatClient

Main (...) marcelo 113DAC27A.itam.mx

Interfaces y clases de Java

public interface **Remote**
The Remote interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine.
Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends java.rmi.Remote are available remotely.
Implementation classes can implement any number of remote interfaces and can extend other remote implementation classes.
RMI provides some convenience classes that remote object implementations can extend which facilitate remote object creation. These classes are java.rmi.server.UnicastRemoteObject and java.rmi.activation.Activatable.

```
java.lang.Object
    java.rmi.server.RemoteObject
        java.rmi.server.RemoteServer
            java.rmi.server.UnicastRemoteObject
```

All Implemented Interfaces:
   Serializable, Remote
public class **UnicastRemoteObject**
extends RemoteServer
Used for exporting a remote object with JRMP and obtaining a stub that communicates to the remote object.
The remote object is exported using the UnicastRemoteObject.exportObject(Remote) method
static Remote exportObject(Remote obj, int port)
Exports the remote object to make it available to receive incoming calls, using the particular supplied port.

The RemoteServer (abstract) class is the common superclass to server implementations and provides the framework to support a wide range of remote reference semantics. Specifically, the functions needed to create and export remote objects (i.e. to make them remotely available) are provided abstractly by RemoteServer and concretely by its subclass(es).

The RemoteObject (abstract) class implements the java.lang.Object behavior for remote objects. RemoteObject provides the remote semantics of Object by implementing methods for hashCode, equals, and toString.

Registry is a remote interface to a simple remote object registry that provides methods for storing and retrieving remote object references bound with arbitrary string names. The `bind`, `unbind`, and `rebind` methods are used to alter the name bindings in the registry, and the lookup and list methods are used to query the current name bindings. In its typical usage, a Registry enables RMI client bootstrapping: it provides a simple means for a client to obtain an initial reference to a remote object. Therefore, a registry's remote object implementation is typically exported with a well-known address, such as with a well-known ObjID and TCP port number (default is 1099).

The **LocateRegistry** class is used to obtain a reference to a bootstrap remote object registry on a particular host (including the local host), or to create a remote object registry that accepts calls on a specific port. Note that a `getRegistry` call does not actually make a connection to the remote host. It simply creates a local reference to the remote registry and will succeed even if no registry is running on the remote host. Therefore, a subsequent method invocation to a remote registry returned as a result of this method may fail.

The **Naming** class provides methods for storing and obtaining references to remote objects in a remote object registry. Each method of the Naming class takes as one of its arguments a name that is a String in URL format (without the scheme component) of the form:

```
    //host:port/name
```

Where `host` is the host (remote or local) where the registry is located, `port` is the port number on which the registry accepts calls, and where `name` is a simple string uninterpreted by the registry. Both `host` and `port` are optional. If `host` is omitted, the host defaults to the local host. If `port` is omitted, then the port defaults to 1099, the "well-known" port that RMI's registry, `rmiregistry`, uses. *Binding* a name for a remote object is associating or registering a name for a remote object that can be used at a later time to look up that remote object. A remote object can be associated with a name using the Naming class's `bind` or `rebind` methods. Once a remote object is registered (bound) with the RMI registry on the local host, callers on a remote (or local) host can lookup the remote object by name, obtain its reference, and then invoke remote methods on the object. A registry may be shared by all servers running on a host or an individual server process may create and use its own registry if desired.

java.rmi.server.codebase
This property specifies the locations from which classes that are published by this VM (for example: stub classes, custom classes that implement the declared return type of a remote method call, or interfaces used by a proxy or stub class) may be downloaded. The value of this property is a string in URL format or a space-separated list of URLs that will be the codebase annotation for all classes loaded from the CLASSPATH of (and subsequently marshalled by) this VM. This property must be set correctly in order to dynamically download classes and interfaces using Java RMI. If this property is not set correctly, you will likely encounter exceptions when attempting to run your server or client.

public interface **Serializable**
Serializability of a class is enabled by the class implementing the Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable.
The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through `synchronized` methods.