

# Sistema de Gestión de Comercio Remoto usando Java RMI - Memoria de la Práctica (2012-2013)

January 5, 2013

Octavio Martinez Garcia  
71280002-N  
omartinez185@alumno.uned.es  
CA Burgos

## Contents

<b>1 Implementación:</b>	<b>3</b>
1.1 Package Regulador:	3
1.1.1 Regulador:	3
1.1.2 ServicioAutenticacionInterface:	6
1.1.3 ServicioMercanciasInterface:	6
1.1.4 ServicioAutenticacionImpl:	7
1.1.5 ServicioMercanciasImpl:	9
1.2 Package Distribuidor:	11
1.2.1 Distribuidor:	11
1.2.2 ServicioVentaInterface:	13
1.2.3 ServicioVentaImpl:	13
1.3 Package Cliente:	15
1.3.1 Cliente:	15
1.4 Package Common:	17
1.4.1 Oferta:	17
1.4.2 TProducto:	18
1.4.3 TextUI:	18
1.4.4 Utils:	20
<b>2 Diagrama de Clases:</b>	<b>21</b>
<b>3 Operativa del Sistema:</b>	<b>22</b>
<b>4 Conclusiones, Notas, etc.:</b>	<b>30</b>
4.1 Recopilacion de decisiones de diseño:	30
<b>5 Bibliografía:</b>	<b>30</b>

# 1 Implementación:

Para establecer el marco en que se desarrolla la práctica, inicialmente se han creado 4 paquetes, cliente, distribuidor, regulador y common, que contienen el código de las diferentes clases java relacionados directamente con ellos, y clases de utilidad empleadas por el resto de clases, dentro del common.

A continuación se detallan las clases de cada uno de ellos junto con sus funcionalidades:

## 1.1 Package Regulador:

### 1.1.1 Regulador:

Es la Clase que representa al regulador de la actividad comercial y la cual posee y gestiona los dos servicios de que hacen uso los clientes y distribuidores, es decir, el Servicio de Autenticación y el Servicio de Mercancías.

Posee dos atributos que son precisamente los dos servicios que gestiona y de los cuales hacen uso el resto de actores del sistema:

```
private ServicioAutenticacionImpl autenticador;  
private ServicioMercanciasImpl servMercancias;
```

Asimismo, mediante el constructor de la clase se crea un objeto regulador y se levantan sus dos servicios, autenticación y mercancías, mediante los métodos privados [1.1.1] `levantarAut()` y [1.1.1] `levantarMerchs()`

```
public Regulador() throws RemoteException {  
    levantarAut();  
    levantarMerchs();  
}
```

- `levantarAut()`:

1. se obtiene la ruta de la interfaz remota que se va a exportar al servicio de RMI para que esté disponible el acceso a sus métodos remotos, esto se hace mediante la clase `Utils` del package `common`, y su método `setCodebase()` [1.4.4], el cual se explicará mas adelante
2. se crea una nueva instancia del `servAutenticaciónImpl` y se asigna al atributo correspondiente del `Regulador`
3. se exporta dicho objeto en el puerto por defecto mediante `UnicastRemoteObject.exportObject()`, obteniéndose un objeto `remote` que casteamos a un `ServicioAutenticacionInterface`
4. esta interfaz remota se vincula al registro con el nombre "Autenticador", que es el nombre por el cual deberán buscarla en el registro posteriormente los clientes y/o distribuidores que quieran hacer uso de sus métodos.

5. todo este código se encierra en un bloque try/catch para tratar con las posibles excepciones que puedan surgir, imprimiendo un mensaje de error informativo y la traza del error.

```
private void levantarAut() {
    System.out.println("intentando establecer el codebase del autenticador");

    Utils.setCodebase(ServicioAutenticacionInterface.class);
    System.out.println("codebase del autenticador establecido");

    try {
        autenticador = new ServicioAutenticacionImpl();
        ServicioAutenticacionInterface remoteAut =
            (ServicioAutenticacionInterface) UnicastRemoteObject.exportObject(autenticador, 0);
        System.out.println("Regulador exportado");

        Registry registro = LocateRegistry.getRegistry();
        System.out.println("Registro obtenido");

        registro.rebind("Autenticador", remoteAut);
        System.out.println("Registro bindeado");

    } catch (Exception e) {
        System.err.println("error levantando el serv aut");
        e.printStackTrace();
    }
}
```

- levantarMercs():

- en este caso se procede igual que con el anterior servicio, pero con el Servicio de Mercancias del regulador

```
private void levantarMercs() {
    System.out.println("intentando establecer el codebase del serv Mercs");

    Utils.setCodebase(ServicioMercanciasInterface.class);
    System.out.println("codebase del serv Mercs establecido");

    try {
        servMercancias = new ServicioMercanciasImpl();
        ServicioMercanciasInterface remoteMercs =
            (ServicioMercanciasInterface) UnicastRemoteObject.exportObject(servMercancias, 0);
        System.out.println("Serv mercs exportado");

        Registry registro = LocateRegistry.getRegistry();
        System.out.println("Registro obtenido");

        registro.rebind("ServMercs", remoteMercs);
        System.out.println("Serv Mercs bindeado");

    } catch (Exception e) {
        System.err.println("error levantando el serv mercs");
        e.printStackTrace();
    }
}
```

El regulador tiene 5 métodos públicos que se corresponden con las funciones disponibles en su interfaz de usuario. Básicamente hacen uso de la clase de utilidad TextUI [1.4.3] que representa la interfaz de usuario basada en texto para mostrar mensajes significativos al usuario, y de los métodos (no remotos, ya que solo hace uso de ellos el regulador, el cual contiene estas dos clases como

atributos) definidos en las clases ServicioAutenticacionImpl y ServicioMercanciasImpl. Para que no resulte demasiado prolijo, incluyo solamente el código de uno de ellos, ya que el resto son similares.

**NOTA1:** son métodos para ser usados por los usuarios del programa para visualizar listas de clientes, distribuidores, ofertas y demandas, y no para ser usados por los otros actores del sistema de comercio mediante llamadas a métodos remotos (ya que el regulador propiamente dicho carece de una interfaz remota que sea puesta a disposición de clientes/distribuidores)

```
public void listarDemandas() throws RemoteException {
    String op = TextUI.input("Listar Demandas", "Enter para continuar");
    if (op != null) {
        servMercancias.listarDemandas();
    } else {
        System.out.println("Operacion cancelada");
    }
}
```

El método salir es algo diferente y merece un comentario aparte. Bajo este método publico se recoge la acción de finalizar el programa servidor, previamente a lo cual, se desligan del registro los nombres de las interfaces remotas vinculadas anteriormente al levantar los servicios, y se desexportan los objetos remotos exportados al servicio RMI de modo que no estén disponibles para el acceso remoto una vez finalice el regulador del cual dependen

```
public void salir() throws RemoteException, NotBoundException {
    String op = TextUI.input("Salir", "Enter para continuar");
    if (op != null) {
        Registry registro = LocateRegistry.getRegistry();
        registro.unbind("Autenticador");
        UnicastRemoteObject.unexportObject(autenticador, true);
        registro.unbind("ServMercs");
        UnicastRemoteObject.unexportObject(servMercancias, true);
        System.exit(0);
    } else {
        System.out.println("Operacion cancelada");
    }
}
```

Finalmente está el método main de la clase, el cual básicamente crea un objeto regulador (el cual a su vez levanta sus 2 servicios, aut y mercs), y crea un menu de opciones de texto, mediante la clase de la interfaz de texto, dentro de un bucle infinito while, el cual finaliza cuando se selecciona la opcion correspondiente a salir del programa. Asimismo dentro del main se encarga de crear el registro de java RMI en el puerto por defecto 1099.

```
public static void main (String[] args) throws Exception
{
    try{
        //levantamos el registro de rmi
        LocateRegistry.createRegistry(1099);

        // creamos el regulador que levanta sus dos servicios aut y mercs
        Regulador regulador = new Regulador();

        String[] opciones = {"Listar Ofertas", "Listar Demandas", "Listar Clientes", "Listar Distribuidores", "Salir"};

        while(true) {
            int op = TextUI.menu("Regulador", opciones);
            switch (op) {
```

```

        case 0:
            regulador.listarOfertas();
            break;
        case 1:
            regulador.listarDemandas();
            break;
        case 2:
            regulador.listarClientes();
            break;
        case 3:
            regulador.listarDistribuidores();
            break;
        case 4:
            regulador.salir();
            break;
        default:
            System.out.println("Opcion no disponible");
            break;
    }
} catch (Exception e){
    e.printStackTrace();
}
}

```

### 1.1.2 ServicioAutenticacionInterface:

Corresponde a la interfaz remota del servicio de autenticación que será exportada al servicio de java RMI para que los clientes y distribuidores hagan uso de sus métodos remotos:

- `getAutenticacion()` : les permite autenticarse en el sistema para interactuar con el resto de servicios que proporciona.
- `getDistriName()` : les permite obtener el nombre de un distribuidor para ponerse en contacto con el a la hora de comprarle una oferta de un producto

```

public interface ServicioAutenticacionInterface extends Remote{

    public int getAutenticacion(String name, int tipo) throws RemoteException;

    public String getDistriName(int aut, int id) throws RemoteException;

}

```

### 1.1.3 ServicioMercanciasInterface:

Corresponde a la interfaz remota del servicio de mercancías que será exportada al servicio de java RMI para que los clientes y distribuidores hagan uso de sus métodos remotos:

- `registrarOf()` : permite a los distribuidores registrar una oferta de un producto en el servicio de mercancías del regulador:
  - NOTA2: en mi caso, he establecido que un regulador puede realizar varias ofertas de un mismo producto (las cuales se diferencian por sus atributos (tipo de producto, cantidad (kg), precio (€)) ) pero los clientes deben comprar la oferta completa y no solamente una parte de ella.

- registrarDem() : permite a los clientes registrarse como demandantes de un determinado producto en el serv. de mercancías
- getOfertas() : imprime las ofertas en las que estaria interesado un cliente según sus demandas, en cuyos tipos de producto se ha debido inscribir como demandante previamente
- borrarOferta() : borra la entrada del registro, dentro del servicio de mercancías, de una oferta concreta de un distribuidor. Se utiliza a la hora de efectuar compras de mercancía a un distribuidor, de forma que el cliente informe al serv. de mercancías de que se ha completado la compra y se puede borrar esa entrada del registro de ofertas vinculado a ese distribuidor.

```
public interface ServicioMercanciasInterface extends Remote{
    public void registrarOf(int id, Oferta of) throws RemoteException;
    public void registrarDem(int id, TProducto dem) throws RemoteException;
    public void getOfertas(int id) throws RemoteException;
    public void borrarOferta(Oferta of, int id) throws RemoteException;
}
```

#### 1.1.4 ServicioAutenticacionImpl:

Es la clase que implementa la interfaz remota del serv autenticación. Consta de 4 mapas en los cuales se mantienen las correspondencias de id (nº de identificación obtenido a través de su servicio de autenticación) con nombres y viceversa, para clientes y proveedores, de modo que pueda almacenar esos vínculos para verificaciones de id y/o nombres que estén autenticados en el sistema, a fin de permitirles o denegarles el acceso a los servicios del resto del sistema.

```
private HashMap<Integer, String> cliente_id_nameMap = new HashMap<Integer, String>();
private HashMap<String, Integer> cliente_name_idMap = new HashMap<String, Integer>();
private HashMap<Integer, String> distrib_id_nameMap = new HashMap<Integer, String>();
private HashMap<String, Integer> distrib_name_idMap = new HashMap<String, Integer>();
```

En cuanto al constructor, no tiene ninguna función especial mas allá de llamar al constructor de la superclase (Object unicamente en este caso).

Los dos métodos principales que implementan los métodos remotos de la interfaz remota se muestran a continuación:

```
public int getAutenticacion(String name, int tipo) throws RemoteException {
    switch (tipo) {
        case 0:
            if (cliente_name_idMap.containsKey(name)){
                System.out.println(cliente_name_idMap.get(name));
                return cliente_name_idMap.get(name);
            } else {
                int iDnumber = new Random().nextInt();
                cliente_name_idMap.put(name, iDnumber);
                cliente_id_nameMap.put(iDnumber, name);
                System.out.println(iDnumber);
            }
    }
}
```

```

        return iDnumber;
    }
    case 1:
        if (distrib_name_idMap.containsKey(name)) {
            System.out.println(distrib_name_idMap.get(name));
            return distrib_name_idMap.get(name);
        } else {
            int iDnumber = new Random().nextInt();
            distrib_name_idMap.put(name, iDnumber);
            distrib_id_nameMap.put(iDnumber, name);
            System.out.println(iDnumber);
            return iDnumber;
        }
    default:
        System.out.println("valores de tipo: 0 - cliente , 1 - distribuidor");
        return -1;
    }
}

```

Este es el método mediante el cual los clientes/distribs obtienen su id de autenticación y se registran en el sistema como autenticados; en función del tipo de actor (cliente (0) , distribuidor (1)) se selecciona con un switch el mapa correspondiente, desde el cual se busca la id para el nombre pasado como parámetro, y en caso de no existir se crea un numero aleatorio mediante la clase Random, el cual es introducido en ambos mapas (id -> nombre y viceversa, para hacer búsquedas por nombre o por id), y finalmente se devuelve dicho id, o un código de error (-1) en caso de que se pase un tipo de actor no válido (diferente de 0 o 1).

**NOTA3:** si bien es posible que al generarse un nuevo número pseudoaleatorio se genere el mismo que uno ya preexistente (de modo que dos actores diferentes tengan el mismo id), la posibilidad de esto es bastante remota, ya que se utiliza el método sin limite del Random (nextInt()) el cual genera  $2^{32}$  posibles valores con igual probabilidad aproximadamente.

```

public String getDistriName(int aut, int id) throws RemoteException {
    if (checkID(aut)){
        return distrib_id_nameMap.get(id);
    } else {
        System.out.println("cliente no autenticado , operacion denegada");
        return null;
    }
}

```

Mediante este método podemos obtener el nombre de un distribuidor cuya id hayamos pasado como parámetro, en caso de que el cliente este autenticado en el sistema , lo cual se comprueba a través de checkID. Este método se usa para obtener el nombre de un distribuidor a la hora de proceder a la compra de una oferta en la que esté interesado.

Los 3 métodos restantes de la clase corresponden a métodos locales para listar a los clientes y distribuidores autenticados en el sistema (el cual es usado de forma local en la interfaz de usuario del regulador), y el mencionado checkID que comprueba la existencia de una id registrada. Su código no tiene especial interés por lo que lo omitimos en la memoria.



### 1.1.5 ServicioMercanciasImpl:

Es la implementacion de la interfaz remota del serv de mercancias. Contiene dos atributos de instancia que representan los registros de “cliente-demandas” y “distribuidor-lista de ofertas”, a través de HashMaps.

```
private HashMap<Integer, List<Oferta>> id_ofertas = new HashMap<Integer, List<Oferta>>();
private HashMap<Integer, Set<TProducto>> id_demandas = new HashMap<Integer, Set<TProducto>>();
```

El constructor es igual que en el caso del serv de autenticación.

Contiene 4 métodos que implementan los métodos remotos de la interfaz y que se describen a continuación:

- registrarOf(int , Oferta): basicamente comprueba la existencia del id pasado como parámetro, en cuyo caso obtiene la lista de ofertas asociada a ese id y añade la nueva oferta, pasada asimismo como param, a la lista poniendo de nuevo en el mapa la entrada modificada con la nueva lista. En caso de que no exista esa id en el mapa, crea una nueva lista de ofertas, añade la oferta y pone la lista en el mapa junto con el id.

```
public void registrarOf(int id, Oferta of) throws RemoteException {
    if(id_ofertas.containsKey(id)) {
        List<Oferta> ofs = id_ofertas.get(id);
        if(ofs == null){
            System.out.println(" lista de ofertas nula, creamos una");
            ofs = new LinkedList<Oferta>();
            imprOfertas(ofs);
        }
        ofs.add(of);
        id_ofertas.put(id, ofs);
    } else {
        List<Oferta> ofs = new LinkedList<Oferta>();
        ofs.add(of);
        id_ofertas.put(id, ofs);
    }
}
```

- registrarDem(int, TProducto): hace la misma operacion que en el caso anterior, con la diferencia de que el mapa no esta compuesto por un entero (id) y una lista, sino por un entero y un conjunto (hashSet) ya que los clientes pueden hacer demandas de varios tipos de productos diferentes entre si, y no varias demandas de un mismo tipo (ya que no tendría sentido). No mostramos el código por ser muy similar al anterior.
- getOfertas(int): se utiliza para obtener e imprimir las ofertas en las que estaría interesado un cliente según sus preferencias de tipos de producto. Se utilizan dos bucles for anidados que recorren los distribuidores por su id y los tipos de producto en que está registrado el cliente, mientras van añadiendo las ofertas de cada distribuidor para cada tipo de producto demandado y las imprimen clasificadas por distribuidor. Se hace uso de

un par de métodos privados `getOfertasDistri(int, TProducto)` y `imprOfertas(List<Oferta>, int)`, que obtienen las ofertas de un tipo de producto para un distribuidor concreto e imprimen dicha lista de ofertas para un distribuidor, respectivamente. Se omite el código de los privados ya que no es especialmente significativo.

```
public void getOfertas(int id){
    Set<TProducto> prods = id_demandas.get(id);
    Set<Integer> distri_ids = id_ofertas.keySet();
    List<Oferta> ofs_buscadas = new LinkedList<Oferta>();

    for(Integer d_id: distri_ids){
        for(TProducto tp: prods){
            List<Oferta> auxofs = getOfertasDistri(d_id, tp);
            ofs_buscadas.addAll(auxofs);
        }
        imprOfertas(ofs_buscadas, d_id);
        ofs_buscadas.clear();
    }
    System.out.println("creada lista con todas las ofertas de todos
        los distribs para las prefs del cliente");
}
```

- `borrarOferta(Oferta, int)`: se utiliza para borrar la oferta pasada de la entrada del registro (mapa) del distribuidor cuya id es pasada como param. Se utiliza cuando un cliente completa una compra de una oferta, el cual pasa este mensaje a través de la interfaz remota para que el serv de mercancías actualice su registro eliminando la oferta indicada del distribuidor al que se le compró. Se obtiene la lista de ofertas del distribuidor, y se recorre buscando una oferta cuyos atributos coincidan exactamente con los de la oferta especificada, para en caso afirmativo eliminarla de la lista y actualizar la entrada del mapa con la nueva lista.

```
public void borrarOferta(Oferta of, int id){
    try {
        List<Oferta> ofs = id_ofertas.get(id);
        // buscamos la oferta a eliminar
        for(Oferta o: ofs){
            if(o.getTipo().equals(of.getTipo()) && o.getKilos() == of.getKilos()
                && o.getPrecio() == of.getPrecio()){
                // si coinciden los atributos de la oferta la eliminamos de la lista
                // suponemos que solo puede existir una sola oferta con = atributos
                ofs.remove(o);
            }
        }
        // cambiamos la entrada del mapa con la nueva lista de ofertas (eliminada of)
        id_ofertas.put(id, ofs);
    } catch (Exception e) {
        e.printStackTrace();
        System.out.println("error al eliminar oferta");
    }
}
```

Asimismo, esta clase posee 2 métodos locales para uso por parte de la interfaz de usuario en el regulador, para listar las ofertas y las demandas registradas en los mapas que posee como atributos, así como una serie de métodos privados para imprimir las ofertas o demandas registradas. No tiene especial relevancia su código, de modo que no se mostrará aquí.

## 1.2 Package Distribuidor:

### 1.2.1 Distribuidor:

Representa al distribuidor del sistema de comercio. Posee una serie de atributos que especifican, su n<sup>o</sup> de autenticación (id), su nombre, el tipo de actor (cliente (0), distribuidor (1): para propósitos de autenticación) y los 3 servicios de los que hace uso, uno local y gestionado por el mismo, el servicio de ventas, y 2 obtenidos de forma remota, el serv de autenticación y el de mercancías.

```
private int id;
private String name;
private static final int TIPO_D = 1; // tipo 1: distribuidor
private ServicioVentaImpl servVentas;
private ServicioAutenticacionInterface sai;
private ServicioMercanciasInterface smi;
```

El constructor se encarga de inicializar los atributos del distribuidor. Inicialmente lee un nombre desde la entrada (usando la clase de interfaz de texto TextUI [1.4.3]) y asigna ese nombre al atributo correspondiente. Posteriormente obtiene el registro de RMI y una vez hecho, levanta su servicio de ventas (mediante el método privado levantarVentas() [1.2.1]) y obtiene las interfaces remotas de los servicios de autenticación y mercancías exportados y registrados por el regulador. Para ello los busca en el registro mediante los nombres por los cuales les vinculó el regulador. En el caso del serv de autenticación, hace uso de el ya para obtener su id, que guarda en su atributo correspondiente.

```
public Distribuidor() throws RemoteException {
    System.out.println("Como te llamas, distribuidor?");
    name = TextUI.readLine();
    System.out.println("Distribuidor: " + name);
    Registry registry = LocateRegistry.getRegistry();
    try {

        levantarVentas();
        sai = (ServicioAutenticacionInterface) registry.lookup("Autenticador");
        id = sai.getAutenticacion(name, TIPO_D);
        System.out.println(name + ": " + id);
        smi = (ServicioMercanciasInterface) registry.lookup("ServMercs");

    } catch (NotBoundException e) {
        e.printStackTrace();
    }
}
```

El método privado levantarVentas() se utiliza de la misma forma que se utilizaban [1.1.1] levantarAut() y [1.1.1] levantarMercs() en el regulador. Se obtiene la ruta de la interfaz a exportar, se crea un nuevo objeto de implementación de dicha interfaz, se exporta dicho objeto a RMI y se vincula en el registro con un nombre por el cual deberán buscarle los procesos que deseen usar dicha interfaz remota.

NOTA4: en este caso el nombre por el que se registra no es una cadena fija como en el caso de los servs de aut y mercs del regulador, ya que pueden existir varios procesos distribuidores y por tanto deben tener nombres diferentes, para que los clientes que quieran comprarles puedan buscar sus respectivos servicios de venta en el registro [1.1.4]. De este modo el nombre por el que se vincula es

el propio nombre introducido inicialmente al arrancar el proceso y el cual esta guardado en el atributo name del distribuidor.

```
registro.rebind(this.name, remoteVenta);
```

Asimismo, el distribuidor tiene 3 métodos de interfaz de usuario que se corresponden con las 3 operaciones básicas que se pueden realizar sobre el:

- `introducirOferta()`: Crea una nueva oferta, añadiéndola a su servicio de ventas y registrándola en el serv de mercancías del regulador con el id obtenido de su autenticación. Pide por pantalla los atributos de la nueva oferta a añadir en formato Tipo de Producto, cantidad (kg) y precio (€), y a continuación mediante el uso de la clase de utilidad `TextUI` devuelve un array de strings que se usa para crear la oferta a través del constructor basado en strings de esta clase [1.4.1]. Posteriormente la añade a su lista de ofertas, contenida en el servicio de ventas que mantiene como atributo, y la registra en el serv de mercancías mediante la interfaz remota obtenida en el constructor. En caso de excepción se muestra una información del formato admitido y se muestra la traza del error.

```
public void introducirOferta() throws RemoteException {
    try {
        String[] ops = TextUI.input("Introducir Oferta",
            new String[]{TProducto.getTiposProd(), "Cantidad (kg)", "Precio (euros)"});
        if(ops != null){
            Oferta of = new Oferta(ops[0], ops[1], ops[2]);
            servVentas.añadirOferta(of);
            smi.registrarOf(id, of);
        } else {
            System.out.println("Introduzca formato de oferta correcto: Tipo, kilos, precio");
        }
    } catch (Exception e) {
        System.out.println("Introduzca formato de oferta permitido: Tipo, kilos, precio");
        System.out.println(TProducto.getTiposProd());
        System.out.println("Cantidad (kg): int");
        System.out.println("Precio (euros): double");
        e.printStackTrace();
    }
}
```

- `mostrarVentas()`: imprime un listado de las ventas realizadas por el distribuidor de un tipo de producto especificado. Pide por pantalla el tipo de producto del que queremos visualizar las ventas e imprime un listado de ellas, básicamente llamando al método `listarVentas()` de su serv de ventas. En caso de error (tipo de producto incorrecto) muestra por pantalla una lista de los tipos de producto permitidos, llamando al método `getTiposProd()` [1.4.2] del tipo enumerado `TProducto`, el cual devuelve una lista de los tipos que contiene, de modo que si añadimos mas tipos de productos esta parte del código no necesite ser modificada.

- salir(): finaliza la ejecución del proceso distribuidor, desvinculando previamente el nombre de la interfaz remota del registro, y quitando (unexport) el objeto remoto de RMI de modo que no pueda ser accedido una vez que esta instancia de distribuidor haya finalizado.

```
public void salir() throws RemoteException, NotBoundException {
    String op = TextUI.input(" Salir ", " Enter para continuar ");
    if (op != null) {
        Registry registry = LocateRegistry.getRegistry();
        registry.unbind(this.name);
        UnicastRemoteObject.unexportObject(servVentas, true);
        System.exit(0);
    } else {
        System.out.println(" Operacion cancelada ");
    }
}
```

El último método es el main de la clase, que muestra el menu de opciones y ejecuta un bucle while mediante el cual se pueden ir seleccionando opciones de la interfaz de usuario de la clase hasta que se selecciona salir() y la ejecución finaliza, de la misma forma que el main del regulador.

### 1.2.2 ServicioVentaInterface:

Es la interfaz remota del servicio de venta que utilizan los clientes para comprar productos al distribuidor, y la cual es exportada a RMI a tal fin. Solo cuenta con un método remoto comprarProducto(), mediante el cual los clientes realizan la mencionada accion.

```
public interface ServicioVentaInterface extends Remote {

    public boolean comprarProducto(Oferta of) throws RemoteException;

}
```

### 1.2.3 ServicioVentaImpl:

Implementa la interfaz remota anterior. Mantiene dos atributos que representan las listas de ofertas que posee y la lista de ventas efectuadas.

```
private List<Oferta> ofertas = new LinkedList<Oferta>();
private List<Oferta> ventas = new LinkedList<Oferta>();
```

El constructor sencillamente llama al constructor de la supeclase (Object en este caso también). Por completitud se ha incluido un constructor que recibe una oferta como parámetro para inicializar un distribuidor (y su serv de ventas) con una oferta concreta, sin embargo en la práctica no se utiliza, ya que las ofertas se van añadiendo a través de la interfaz de usuario de texto del distribuidor.

- `comprarProducto(Oferta)`: implementa el método remoto de la interfaz remota, mediante el cual un cliente puede comprar un producto del serv de ventas asociado a un distribuidor concreto. Básicamente intenta obtener una oferta de la lista de ofertas que posee (método privado [1.2.3] `getOferta(Oferta)`) y en caso afirmativo la añade a la lista de ventas, la elimina de la lista de ofertas (de esto realmente se encarga el método privado) e imprime por pantalla la oferta adquirida (se usa el método `printOferta()` [1.4.1] de la clase `Oferta` para ello). En caso de que no exista dicha oferta o que haya sido mal especificada se muestra el correspondiente mensaje de error.

**NOTA5:** Esta operacion se ha realizado de tal forma que un cliente puede comprar una oferta completa y no cantidades parciales de una oferta, pudiendo un distribuidor tener mas de una oferta de un mismo tipo de producto.

```
public boolean comprarProducto(Oferta of) throws RemoteException {
    Oferta offer = getOferta(of);
    if(offer != null){
        ventas.add(offer);
        System.out.println("Transaccion completada con exito , oferta adquirida");
        offer.printOferta();
        // éxito
        return true;
    } else {
        System.out.println("Operacion de compra no completada");
        // error
        return false;
    }
}
```

El método privado `getOferta(Oferta)` básicamente recorre la lista de ofertas comparando los atributos de la oferta especificada como param con los de cada oferta de la lista, y en caso afirmativo borra esa entrada de la lista y devuelve el objeto oferta, o null si no la encuentra.

**NOTA6:** en este caso se ha tomado como supuesto que una oferta debe estar especificada unívocamente por sus 3 atributos (tipo, kilos, precio).

```
private Oferta getOferta(Oferta of) {
    if(!ofertas.isEmpty()){
        for(Oferta o: ofertas){
            if(o.getTipo().equals(of.getTipo()) && o.getKilos() == of.getKilos()
                && o.getPrecio() == of.getPrecio()){
                ofertas.remove(o);
                return o;
            }
        }
    }
    return null;
}
```

El resto de métodos de la clase son métodos locales invocados por el distribuidor a través de la interfaz de usuario para añadir ofertas o para listar las ventas de un determinado tipo de producto. No se muestran sus códigos ya que no revisten especial interés.

## 1.3 Package Cliente:

### 1.3.1 Cliente:

Representa al cliente del sistema de comercio. Cuenta con una serie de atributos similares a los del distribuidor que especifican su id (cid - client id), su nombre (cname - client name), su tipo ( para propósitos de autenticación - cliente (0) ) y los 3 servicios de los que hace uso y que obtiene de forma remota mediante una consulta en el registro (autenticación, mercancías, ventas).

```
private int cid;  
private String cname;  
private static final int TIPO_C = 0; // tipo 0: cliente  
private ServicioAutenticacionInterface csai;  
private ServicioMercanciasInterface csmi;  
private ServicioVentaInterface csvi;
```

En el constructor se inicializan los atributos de la misma forma que en el caso del distribuidor, con la excepción de que el cliente no necesita levantar ningún servicio (ya que no posee ninguno), simplemente obtiene los que necesita buscándoles en el registro por su nombre. En este caso obtiene el serv autenticacion y el de mercancías en el constructor, donde tambien obtiene su id y su nombre, sin embargo, difiere la obtención del serv de ventas del distribuidor hasta el momento de efectuar una compra [1.3.1] , ya que el serv de ventas al que acceda dependera de la oferta en la que esté interesado y por tanto variará de un distribuidor a otro.

```
public Cliente() {  
    System.out.println(" Cliente , introduzca su nombre: ");  
    cname = TextUI.readLine();  
    System.out.println(" Cliente: " + cname);  
    try {  
        // obtenemos acceso al registro  
        Registry registry = LocateRegistry.getRegistry();  
        // obtenemos las Remote IF de los servicios necesarios  
        csai = (ServicioAutenticacionInterface) registry.lookup("Autenticador");  
        // obtenemos la id de autenticacion  
        cid = csai.getAutenticacion(cname, TIPO_C);  
        System.out.println(" Cliente: " + cname + " (" + cid + ") " + "autenticado");  
        // obtenemos el serv mercs del regulador  
        csmi = (ServicioMercanciasInterface) registry.lookup("ServMercs");  
    } catch (NotBoundException | RemoteException e) {  
        e.printStackTrace();  
    }  
}
```

- comprarMercancia(): a través de la interfaz de texto se va pidiendo al usuario que ingrese el id del distribuidor y los atributos (tipo, kg, precio) de la oferta que desea comprar, lo cual se guarda en un array de strings de 4 elementos, siendo el 1º el id y los otros 3 los atributos de la oferta. Con el id (pasado a int) se busca en el serv de autenticacion el nombre del distribuidor (getDistriName() [1.1.4] ) y se obtiene la instancia del serv de ventas correspondiente buscándola en el registro (lookup()) por el nombre del distribuidor obtenido. A continuación se compra la oferta a través de la interfaz remota obtenida (comprarProducto() [1.2.3]) y si se completa

con éxito la compra se envía el mensaje al serv de mercancías para que la borre del registro (borrarOferta() [1.1.5]). En caso de que alguno de los pasos falle se muestra un mensaje informativo del error.

```
public void comprarMercancia() throws RemoteException {
    try {
        String[] campos = new String[]{ "Id del distribuidor", TProducto.getTiposProd(), "Cantidad (kg)", "Precio (euros)"};
        String[] ops = TextUI.input("Comprar Producto", campos);
        if(ops != null && ops.length == 4){
            // la oferta la forman los 3 ultimos strings del array leído
            Oferta of = new Oferta(ops[1], ops[2], ops[3]);
            // id del distri es el primer string del array leído
            int id = Integer.parseInt(ops[0]);
            // intentamos acceder al serv ventas del distribuidor con esa id
            Registry registry = LocateRegistry.getRegistry();
            String dis_name = csai.getDistriName(cid, id);
            if (dis_name != null){
                csvi = (ServicioVentaInterface) registry.lookup(dis_name);
                // compramos la oferta especificada (tratar oferta incorrecta en svi)
                boolean completada = csvi.comprarProducto(of);
                if(completada) {
                    // mensaje al smi para que la borre
                    System.out.println("Compra ok: borrando oferta del serv mercancías");
                    csmi.borrarOferta(of, id);
                } else {
                    System.out.println("Compra no completada");
                }
            } else {
                System.out.println("nombre de distribuidor null");
            }
        } else {
            System.out.println("Formato de solicitud de compra incorrecto");
        }
    } catch (Exception e){
        e.printStackTrace();
        System.out.println("error de acceso al s.mercs del distribuidor");
    }
}
```

- introducirDemanda(): solicita por pantalla el tipo de producto en que se quiere inscribir y lo registra en el serv de mercancías mediante el método remoto correspondiente (registrarDem() [1.1.5])

```
public void introducirDemanda() throws RemoteException {
    try {
        String op = TextUI.input("Introducir Demanda", "Seleccione " + TProducto.getTiposProd());
        if(TProducto.valueOf(op) != null){
            csmi.registrarDem(cid, TProducto.valueOf(op));
        } else {
            System.out.println("demanda rechazada");
        }
    } catch (IllegalArgumentException e) {
        System.out.println("Demanda no permitida: " + TProducto.getTiposProd());
        e.printStackTrace();
    }
}
```

- recibirOfertas(): obtiene las ofertas de los tipos de productos en los que se ha inscrito previamente como demandante, agrupadas por distribuidor para que pueda seleccionar alguna de ellas y comprarla posteriormente. Similar al método anterior pero con la invocación remota del método apropiado del serv de mercancías (getOfertas() [1.1.5])



```

public void recibirOfertas() throws RemoteException {
    try{
        String op = TextUInput("Recibir Ofertas", "Segun demandas inscritas previamente (Enter para cont)");
        if (op != null){
            System.out.println("entregando lista de ofertas solicitadas");
            csmi.getOfertas(cid);
        } else {
            System.out.println("Operacion cancelada");
        }
    } catch (NullPointerException e) {
        System.out.println("Ofertas nulas");
        e.printStackTrace();
    }
}

```

- salir(): simplemente finaliza la ejecución del proceso cliente (si el usuario confirma la operación [y/n] ), ya que en este caso no necesita desvincular ni desexportar ningún objeto remoto, ya que no gestiona ninguno. El código de este método es bastante trivial de modo que no se mostrará.

Finalmente el método main del cliente es similar a sus contrapartes del regulador y del distribuidor; muestra un menú y ejecuta las opciones seleccionadas repetidamente (while(true)) hasta que se selecciona salir(), punto en el cual finaliza el proceso.

## 1.4 Package Common:

### 1.4.1 Oferta:

Esta clases representa una oferta de un determinado producto, la cual esta especificada por sus atributos tipo de producto, cantidad, precio. Aparte de esto tiene un atributo constante estático que representa el nº de serie utilizado en las tareas de marshalling y unmarshalling ya que esta clase esta definida como serializable, de modo que en la comunicación remota se pasa el objeto completo y no solo una interfaz remota como en el caso de los objetos Remote.

```

private static final long serialVersionUID = 1496479494762190865L;
private int kilos;
private double precio;
private TProducto tipo;

```

Tiene 3 constructores, uno de los cuales (sin params) inicializa los atributos a 0 y null, otro con 3 parámetros de los tipos adecuados que inicializa cada atributo segun el parametro pasado, y un tercero que es el que mas se utiliza en el sistema que recibe 3 strings y los convierte a los tipos adecuados para inicializar los atributos.

```

public Oferta ()
{
    kilos = 0;
    precio = 0.0;
    tipo = null;
}

```

```

public Oferta (TProducto tipo , int kilos , double precio)
{
    this.kilos = kilos ;
    this.precio = precio ;
    this.tipo = tipo ;
}

public Oferta (String t , String k , String p){
    this.tipo = TProducto.valueOf(t);
    this.kilos = Integer.parseInt(k);
    this.precio = Double.parseDouble(p);
}

```

El resto de métodos son los típicos getters y setters, que no merecen mayor atención, exceptuando el método `printOferta()` usado para imprimir una oferta con un formato concreto.

```

public void printOferta () {
    System.out.println("=====");
    System.out.println("-----Oferta-----");
    System.out.println("=====");
    System.out.println("Producto: " + tipo);
    System.out.println("Cantidad: " + kilos + " kg");
    System.out.println("Precio: " + precio + " €");
    System.out.println("=====");
}

```

#### 1.4.2 TProducto:

Es un tipo enumerado que representa los diferentes valores que puede tomar un producto y contiene un método estático para imprimir por pantalla una lista de los valores que contiene:

```

public enum TProducto {
    ARROZ, LENTEJAS, GARBANZOS, JUDIAS, SOJA;

    public static String getTiposProd(){
        TProducto[] tps = TProducto.values();
        String valoresprod = "Tipo de Producto: (" ;
        for(TProducto tp: tps){
            valoresprod = valoresprod + tp.toString();
            valoresprod = valoresprod + " ";
        }
        valoresprod = valoresprod + ")";
        return valoresprod;
    }
}

```

#### 1.4.3 TextUI:

Es una clase de utilidad que especifica la interfaz de usuario de texto, aportando una serie de métodos estáticos para crear menus e imprimir por pantalla los pasos de las opciones seleccionadas. Posee 2 atributos estáticos que representan la consola del sistema y un stream de lectura de cadenas de caracteres.

```

private static Console console = System.console();
private static BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

```

Asimismo posee varios métodos estáticos que definen la interfaz de usuario basada en texto.

- `input(String, String[])`: muestra los pasos a seguir en la realización de alguna de las opciones del menú, cuando tiene varios pasos de realización. El primer parámetro es el nombre de la opción seleccionada, y el segundo un array de strings que representa las indicaciones que va mostrando por pantalla para que el usuario vaya aportando una respuesta a cada una de ellas, respuestas que va guardando en otro array de strings que es devuelto en la llamada al método.

```
public static String[] input(String name, String[] msgs) {
    String[] inputs = new String[msgs.length];

    System.out.println("=== " + name + " ===");

    for (int i = 0; i < msgs.length; i++) {
        inputs[i] = input(msgs[i]);
    }

    return inputs;
}
```

Hace uso del método privado `input(String)`, el cual imprime el mensaje que se le pasa por parámetro y devuelve el string que el usuario introduce por teclado, si este confirma la entrada ([y/n]), y null en caso contrario.

```
private static String input(String msg) {
    System.out.print(msg);
    String line = readLine();

    Boolean opt = null;
    do {
        System.out.print(" Confirmar entrada [y/n] ");
        String yn = readLine();

        if (yn.startsWith("y")) opt = true;
        if (yn.startsWith("n")) opt = false;
    }
    while (opt == null);
    System.out.println();
    if (opt) return line;
    return null;
}
```

- `menu(String, String[])`: crea un menú de texto con el nombre del primer parámetro y con las opciones especificadas en el array de strings pasado como 2º parámetro, devolviendo el número de opción correspondiente (las opciones se numeran de 1..n, pero se corresponden con las posiciones del array, de modo que la opción 1 es la posición `entradas[0]`, y de este modo devuelve el número de opción - 1, para igualarlo a la entrada correspondiente a la opción del array).

```
public static int menu(String name, String[] entradas) {
    System.out.println("=== " + name + " ===");
    System.out.println(" Seleccione una opción.\n");

    for (int i = 0; i < entradas.length; i++) {
```

```

        System.out.println((i + 1) + ".- " + entradas[i]);
    }
    int opt = -1;
    do {
        opt = Integer.parseInt(readLine().trim());

        if (opt - 1 >= entradas.length || opt <= 0) {
            System.out.println("Opcion no disponible: seleccione una opcion del 1 al " + entradas.length);
            opt = -1;
        }
    } while (opt == -1);
    System.out.println();
    return opt - 1;
}

```

- `input(String, String)`: es la version sobrecargada del método `input(String, String[])`, para opciones que solo tengan un paso o pregunta y una sola respuesta.
- `readLine()`: lee una linea de caracteres acabada en retorno de linea desde la entrada estándar y la devuelve como un string. Se hace uso internamente de este método en la mayoría de métodos de esta clase.

```

public static String readLine() {
    if (console != null) return console.readLine();

    try {
        return reader.readLine();
    }
    catch (IOException e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    }
}

```

#### 1.4.4 Utils:

Es otra clase de utilidad que tiene un método estático para obtener la ruta de acceso a una interfaz para prestar apoyo a la hora de exportar una interfaz remota a java RMI. Obtiene la ruta del código de la clase que se le pasa como parámetro y la añade a la propiedad de java RMI especificada. En caso de que ya exista un ruta especificada en la propiedad la añade a la/las rutas anteriores con un espacio entre ellas, de modo que se conserven todas.

```

public class Utils {
    public static final String CODEBASE = "java.rmi.server.codebase";

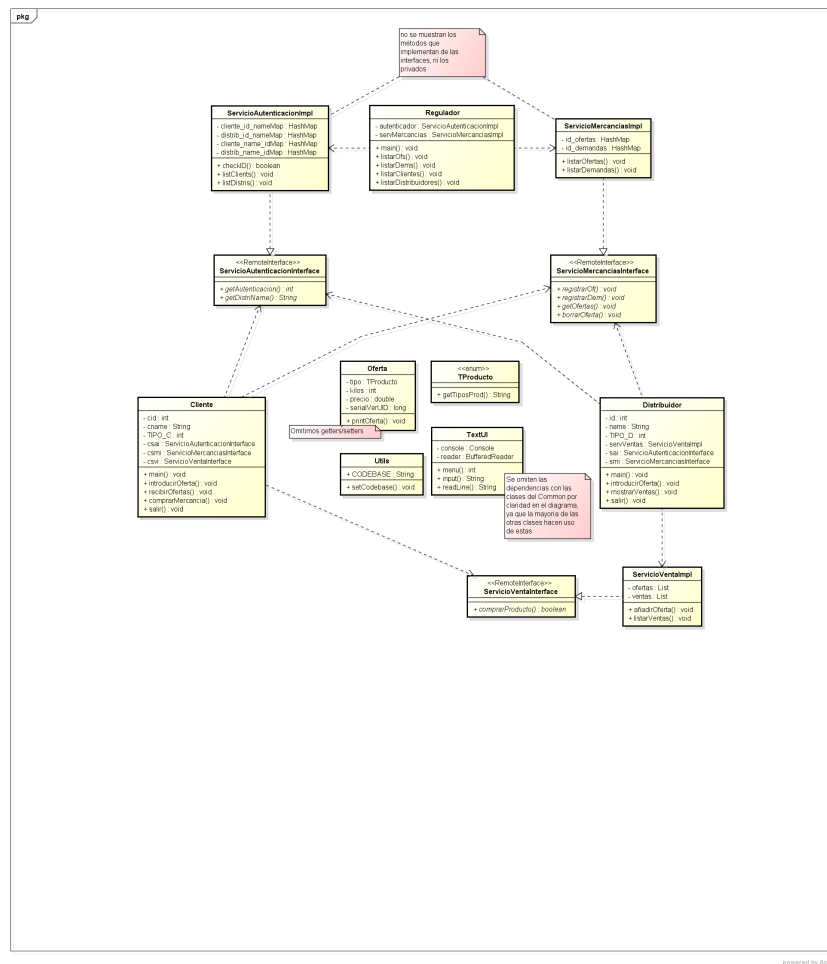
    public static void setCodebase(Class<?> c)
    {
        System.out.println("obteniendo ruta de clase");
        String rutaDeClase = c.getProtectionDomain().getCodeSource().getLocation().toString();
        System.out.println("ruta de clase obtenida ok");

        String previousPath = System.getProperty(CODEBASE);

        if (previousPath != null && !previousPath.isEmpty()) {
            rutaDeClase = previousPath + " " + rutaDeClase;
        }
        System.setProperty(CODEBASE, rutaDeClase);
    }
}

```

## 2 Diagrama de Clases:



Como se comenta en la nota del diagrama, hemos obviado las flechas de dependencias con las 4 clases del paquete common para evitar que el diagrama se volviera demasiado lioso al introducir muchas asociaciones, ya que la mayoría

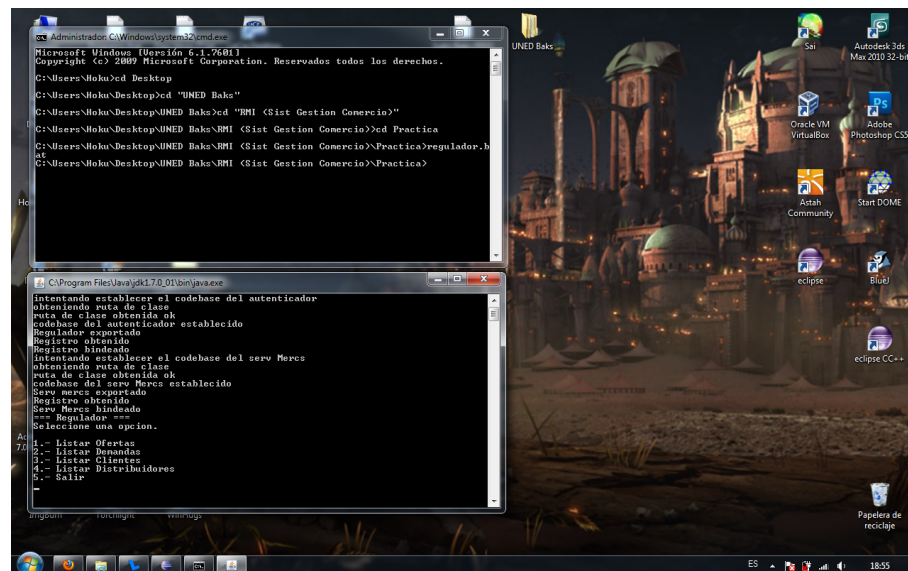
del resto de clases hacen uso, en mayor o menor medida, de esas 4 clases de utilidad (sobre todo de oferta y TextUI).

### 3 Operativa del Sistema:

A continuacion se muestra un ejemplo de ejecución del sistema con una serie de pantallazos representativos de la operativa del sistema:

Para iniciar el sistema se ejecutan los archivos .bat que se encargan de ejecutar los .jar correspondientes:

- Screen1 - Ejecucion de regulador.bat: se observa como se inicia el proceso regulador, en el cual se han añadido una serie de sysouts indicativos del paso del proceso de exportar las interfaces remotas. Finalmente se muestra el menu de opciones del regulador.



```
Administrador: C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\Maku>cd Desktop
C:\Users\Maku\Desktop>cd "UNED Baku"
C:\Users\Maku\Desktop\UNED Baku>cd "RMI <Sist Gestion Comercio>"
C:\Users\Maku\Desktop\UNED Baku\RMI <Sist Gestion Comercio>>cd Practica
C:\Users\Maku\Desktop\UNED Baku\RMI <Sist Gestion Comercio>\Practica>regulador.bat
C:\Users\Maku\Desktop\UNED Baku\RMI <Sist Gestion Comercio>\Practica>

C:\Program Files\Java\jdk1.7.0_01\bin\java.exe
Intentando establecer el codebase del autenticador
obteniendo ruta de clase
ruta de clase obtenida ok
codebase del autenticador establecido
Regulador exportado
Registro obtenido
Registro bindeado
Intentando establecer el codebase del serv Mercs
obteniendo ruta de clase
ruta de clase obtenida ok
codebase del serv Mercs establecido
Serv mercs exportado
Registro obtenido
Serv Mercs bindeado
== Regulador ==
Seleccione una opcion.
1.- Listar Ofertas
2.- Listar Demandas
3.- Listar Clientes
4.- Listar Distribuidores
5.- Salir
7:~
```

- Screen2: Se ejecutan 2 distribuidores y un cliente: se observa como se solicita el nombre de cada uno de ellos y se obtienen las ids correspondientes para finalizar mostrando el menu de opciones de cada uno. Se observa como en la ventana del regulador se muestran las ids autenticadas.

```

Administrador: C:\Windows\system32\cmd.exe
Microsoft Windows [Versión 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.

C:\Users\Hoku\Desktop>cd "UNED Baka"
C:\Users\Hoku\Desktop\UNED Baka>cd "RMI (Sist Gestion Comercio)"
C:\Users\Hoku\Desktop\UNED Baka\RMI (Sist Gestion Comercio)>cd Practica
C:\Users\Hoku\Desktop\UNED Baka\RMI (Sist Gestion Comercio)\Practica>regulador.bat
C:\Users\Hoku\Desktop\UNED Baka\RMI (Sist Gestion Comercio)\Practica>distribuidor.bat
C:\Users\Hoku\Desktop\UNED Baka\RMI (Sist Gestion Comercio)\Practica>cliente.bat
C:\Users\Hoku\Desktop\UNED Baka\RMI (Sist Gestion Comercio)\Practica>

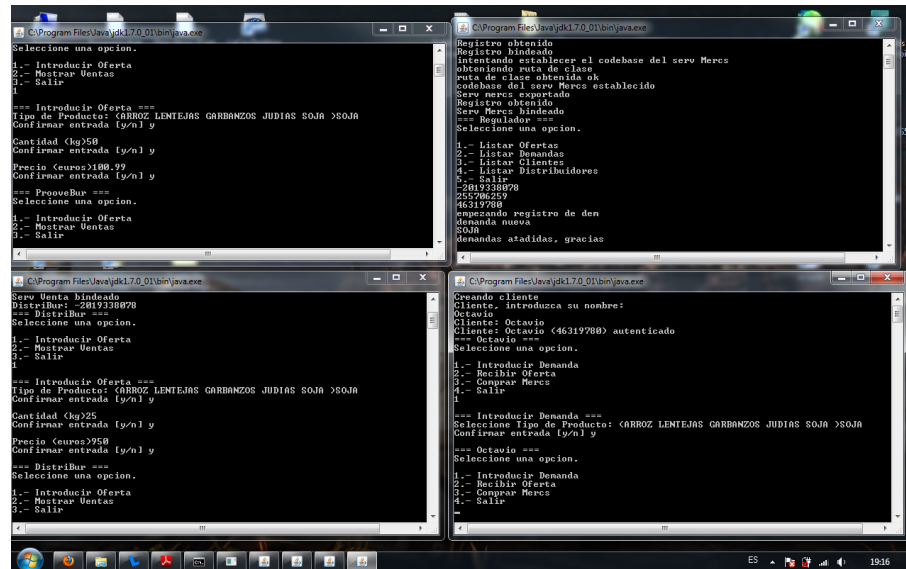
C:\Program Files\Java\jdk1.7.0_01\bin\java.exe
obteniendo ruta de clase
ruta de clase obtenida ok
codebase del autenticador establecido
Regulador exportado
Registro obtenido
Intentando establecer el codebase del serv Mercs
obteniendo ruta de clase
ruta de clase obtenida ok
codebase del serv Mercs establecido
Serv Mercs exportado
Registro obtenido
Serv Mercs bindeado
== Regulador ==
Seleccione una opcion.
1.- Listar Ofertas
2.- Listar Demandas
3.- Listar Clientes
4.- Listar Distribuidores
5.- Salir
6.- 2019338878
635796259
46319788

C:\Program Files\Java\jdk1.7.0_01\bin\java.exe
Como te llamas, distribuidor?
ProveeBur:
Intentando establecer el codebase del serv Venta
obteniendo ruta de clase
ruta de clase obtenida ok
codebase del serv Venta establecido
Serv Venta exportado
Registro obtenido
Serv Venta bindeado
ProveeBur: 255786259
== ProveeBur ==
Seleccione una opcion.
1.- Introducir Oferta
2.- Retirar Ventas
3.- Salir

C:\Program Files\Java\jdk1.7.0_01\bin\java.exe
Creando cliente
Cliente: Introduzca su nombre:
Octavio
Cliente: Octavio
Cliente: Octavio (46319788) autenticado
== Octavio ==
Seleccione una opcion.
1.- Introducir Demanda
2.- Retirar Oferta
3.- Comprar Mercs
4.- Salir

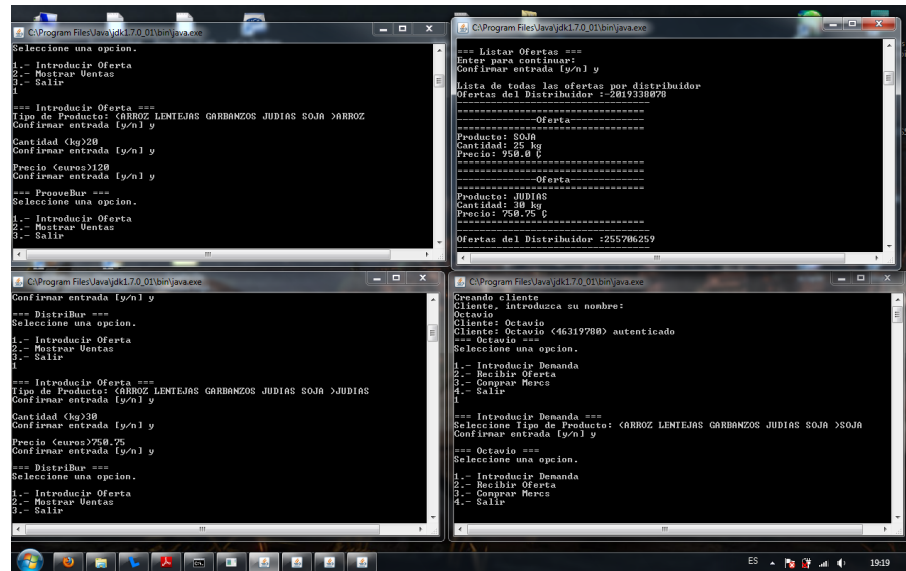
```

- Screen3: Se introduce una oferta en cada distribuidor y se registra al cliente como demandante de ese producto (SOJA).

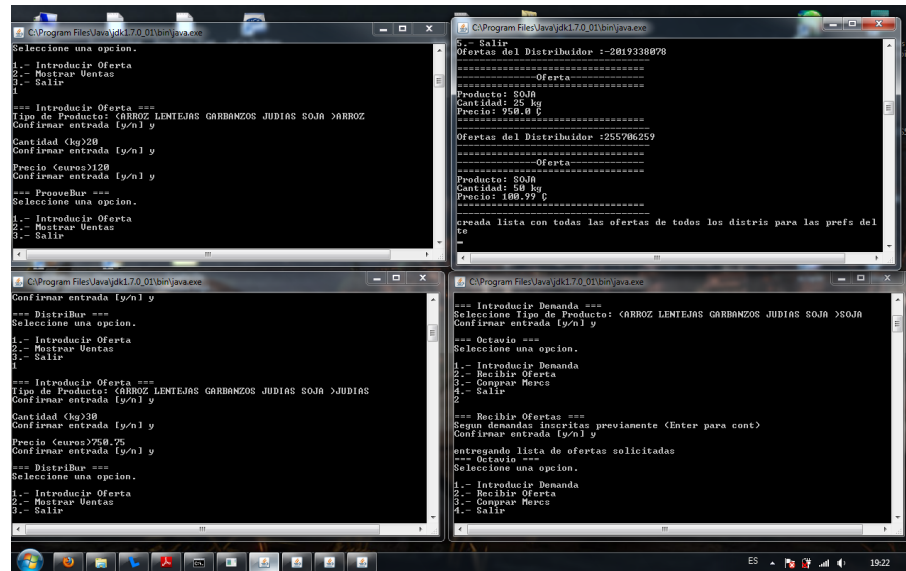




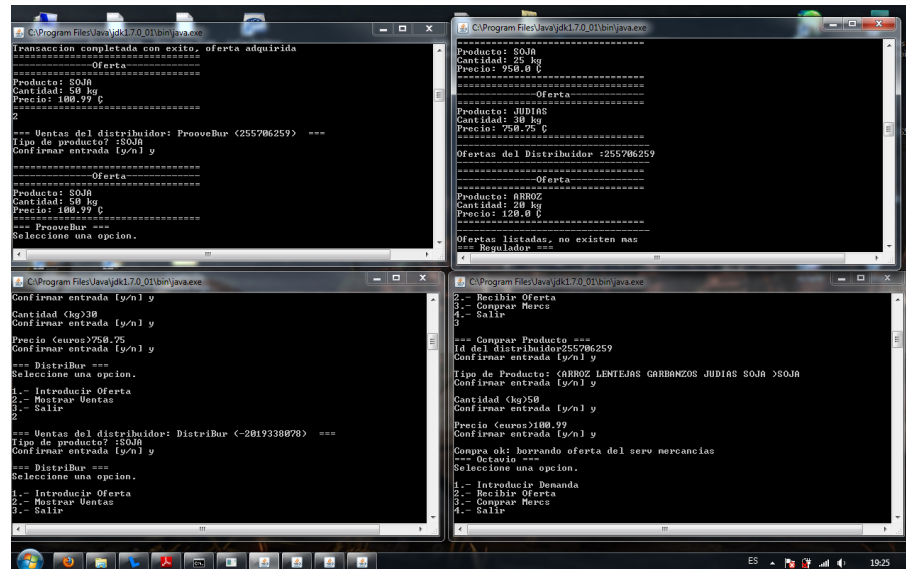
- Screen4: Ambos distribuidores introducen una oferta mas de otro producto y se listan las ofertas en el regulador:



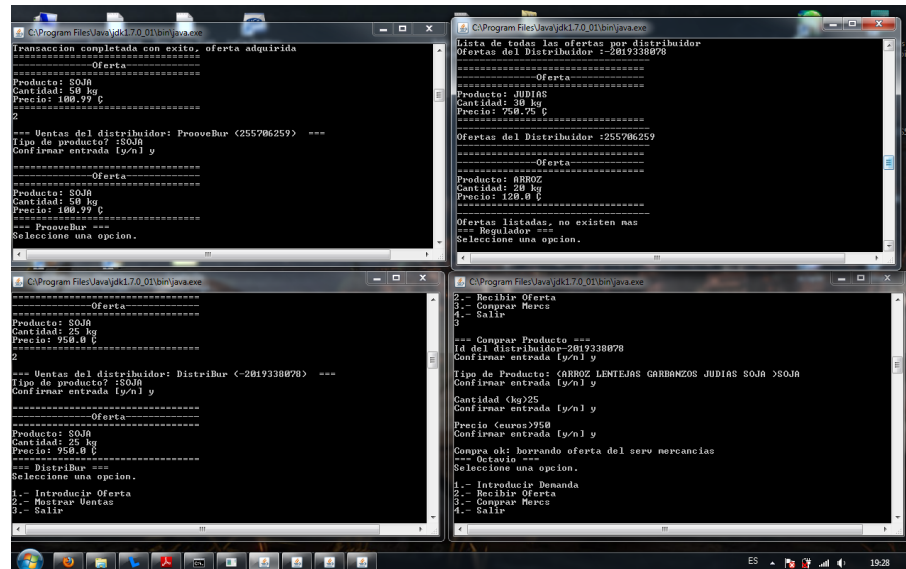
- Screen5: El cliente solicita recibir las ofertas de soja existentes, y estas se muestran en la ventana del regulador agrupadas por distribuidor



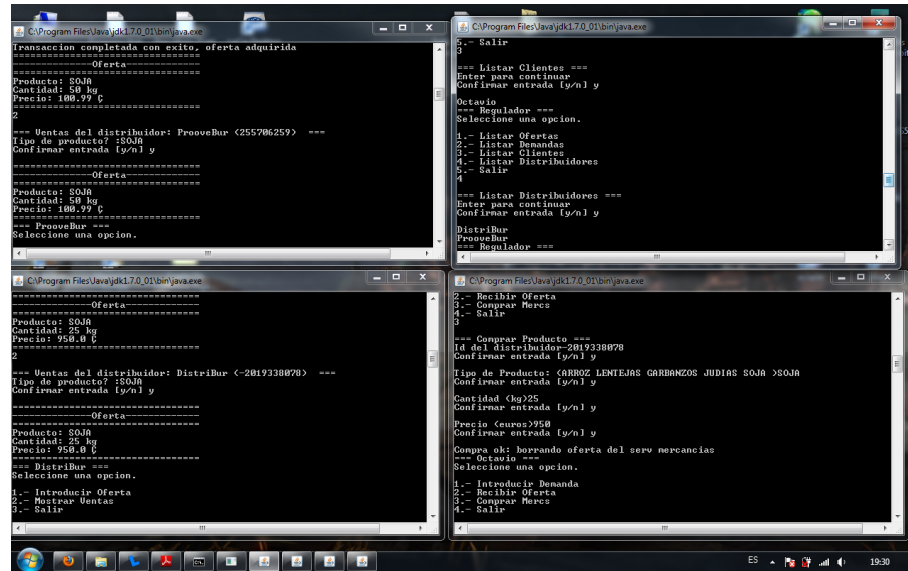
- Screen6: El cliente procede a comprar la oferta mas barata y se muestran las ventas en el distribuidor, listandose asimismo las ofertas en el regulador para verificar que ya no esta vigente esa oferta:



- Screen7: El cliente procede a comprar la otra oferta y se muestran las ventas en el otro distribuidor, listandose asimismo las ofertas en el regulador para verificar que ya no esta vigente tampoco esa otra oferta:



- Screen8: El regulador lista los clientes y distribuidores existentes.



## 4 Conclusiones, Notas, etc.:

### 4.1 Recopilacion de decisiones de diseño:

- [1.1.1] los métodos de interfaz que especifica el enunciado entiendo que van a ser usados por los usuarios del programa para visualizar listas de clientes, distribuidores, ofertas y demandas, y no para ser usados por los otros actores del sistema de comercio.
- [1.1.3] en mi caso, he establecido que un regulador puede realizar varias ofertas de un mismo producto pero los clientes deben comprar la oferta completa y no solamente una parte de ella.
- [1.1.4] he establecido que la autenticación se realice por generación de un nuevo random cada vez, ya que si bien cabe la posibilidad de que se generen 2 id's iguales para diferentes actores, la posibilidad es remota dado que hay  $2^{32}$  valores posibles practicamente equiprobables
- [1.2.1] para buscar a un distribuidor concreto y comprarle una oferta un cliente debe buscarle por su nombre, ya que es este nombre el que usa el distribuidor para vincular la interfaz remota de su servicio de ventas en el registro
- [1.2.3] una oferta debe quedar especificada unívocamente por sus 3 atributos (tipo, kilos, precio).
- se ha decidido mantener en el regulador los clientes y distribuidores autenticados hasta que se finaliza la ejecución del regulador, de modo que si vuelven a conectarse (con el mismo nombre se entiende) ya esten autenticados en el sistema de la vez anterior.
- [3] inicialmente habia hecho que desde el bat que ejecuta el regulador se ejecutara asimismo el rmiregistry, sin embargo he decidido modificar esto, de modo que sea la propia clase regulador de java, mediante su método main la que se encargue de crear el registro [1.1.1], para evitar problemas de rutas al ejecutar el programa en otras máquinas.
- el código de las clases de utilidad Utils y TextUI ha sido adaptado del código de Fermín Silva, cuya colaboración quiero agradecer al compartir el video explicativo de RMI que se detalla en la bibliografía.

## 5 Bibliografía:

- Sistemas Distribuidos - Conceptos y diseño: 3ª Ed. (G.Coulouris)
- [API Java RMI](#)

- [Tutorial Java RMI](#)
- [VideoTutorial RMI Fermin Silva](#)