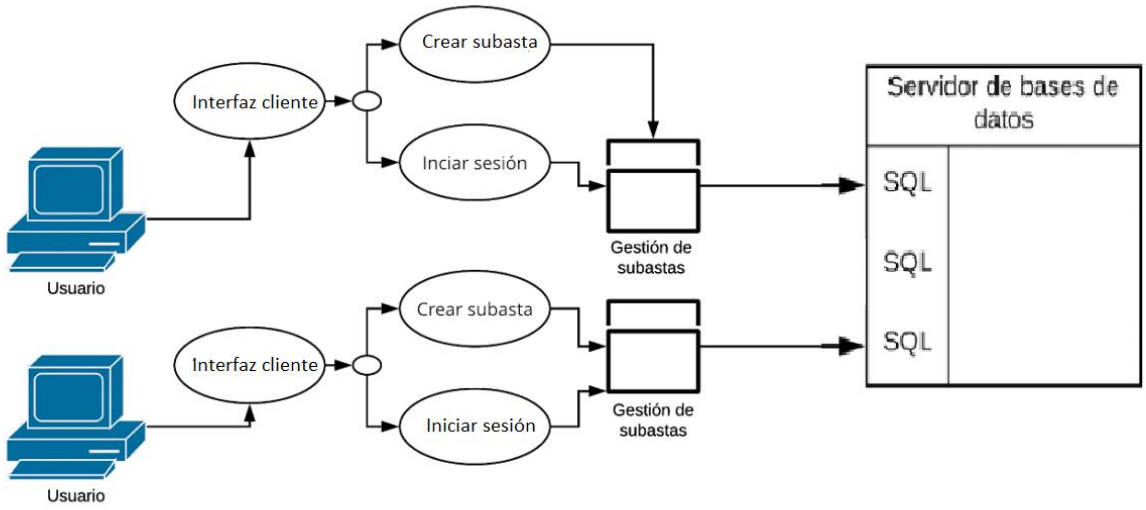
Funciones del sistema

Integrantes
Roberto Navarro
Emmanuel Pech
Daniel Rosales

¿Cómo funciona el sistema?

Estructura base del sistema



El sistema fue diseñado en forma de 3 capas, las cuales están constituidas como se muestra en la imagen. Para la base de datos se empleó MySQL, el cual fue conectado al servidor mediante un manejador (se explicará más adelante en el código). Ahora bien, el servidor se posee un menú bastante simple y conciso, de manera que el usuario simplemente tendrá 5 opciones, de las cuales, las 4 primeras están orientadas al funcionamiento del sistema y, la última, una opción de cierre de las solicitudes.
Como se mencionó, este es el diseño del menú

```
run:
Por favor introduzca su RFC:
->

Por favor introduzca su contraseña:
->
```

```
**********Menu*********
1.-Comprar
2.-Vender
3.-Notificaciones
4.-Ver transacciones
5.-Salir

 introduzca un numero
->
```

```
|||||||Portafolio|||||||
1.- RFCComp= 1234567891 || accionesOperadas=9 || precioOperacion=300.0
||||||||||||||||||||||||
```

```
********Subastas ganadas********
->Precio de la accion: 202.2 estado de la transaccion: comprado
->Precio de la accion: 202.2 estado de la transaccion: comprado
->Precio de la accion: 500.0 estado de la transaccion: comprado
->Precio de la accion: 666.0 estado de la transaccion: comprado
->Precio de la accion: 777.0 estado de la transaccion: comprado
->Precio de la accion: 300.0 estado de la transaccion: comprado
->Precio de la accion: 600.0 estado de la transaccion: comprado
->Precio de la accion: 696.0 estado de la transaccion: comprado
->Precio de la accion: 3.0 estado de la transaccion: comprado

********Subastas perdidas********
->Precio de la accion: 2222.0 estado de la transaccion: perdido

**********************************


********Acciones vendidas********
->Precio de la accion: 200.0 estado de la transaccion: vendido
->Precio de la accion: 202.2 estado de la transaccion: vendido
->Precio de la accion: 999.0 estado de la transaccion: vendido
->Precio de la accion: 300.0 estado de la transaccion: vendido

**********************************
```

Funcionamiento del código

Ahora, se procederá a explicar el funcionamiento, de manera general, del código. No obstante, se mencionarán las secciones del código que más relevancia tengan y las que necesiten una breve explicación.

BankClient

El BankClient es el menú que el usuario tendrá disponible para seleccionar 4 de las 5 opciones disponibles.

El BankClient está conformado por:

- Menu: El cual se encarga de desplegarle todas las opciones disponibles al usuario, de manera que el puede seleccionar la opción que desee usar.

```
System.out.print("**********Menu**********\n"
        + "1.-Comprar\n"
        + "2.-Vender\n"
        + "3.-Notificaciones\n"
        + "4.-Ver transacciones\n"
        + "5.-Salir\n"
        + "\n introduzca un numero\n->");
selected_option = entradaEscaner.nextLine();
```

- User_is_logged: Esta función se encargará del inicio de sesión del usuario, donde se validarán sus credenciales con la base de datos.

```
private static boolean user_is_logged(IRemoteProvince rp) throws RemoteException {
    boolean is_logged = false;
    String usuario = "";
    String contrasena = "";
    boolean salida = false;

    do {
        System.out.print("Por favor introduzca su RFC:\n->");
        usuario = entradaEscaner.nextLine();
        if (usuario.length() == 10) {
            salida = true;
        } else {
            System.out.println("El tamanio del RFC debe ser igual a 10");
        }

    } while (salida == false);

    System.out.print("\nPor favor introduzca su contraseña:\n->");
    contrasena = entradaEscaner.nextLine();

    is_logged = rp.check_user(usuario, contrasena);

    if (is_logged) {
        userRFC = usuario;
    }

    return is_logged;
```

- Execute_compra: Esta función procesará todas las acciones a comprar y se las mostrará al usuario, una vez que el usuario seleccione la opción deseada y cumpla con los puntos necesarios para pujar, el cliente mandará la petición al servidor

```
private static void execute_compra(IRemoteProvince rp) throws RemoteException {
    try {
        ArrayList response = new ArrayList();
        ArrayList<String> infoTransaction = new ArrayList<String>();
        infoTransaction.add(userRFC);

        response = rp.showAll();

        System.out.println("Las opciones de compra son:");
        System.out.println("\n------RFC de empresas-----");

        for (int i = 0; i < response.size(); i++) {
            Compania comp = (Compania) response.get(i);
            System.out.println((i + 1) + ".-" + comp.toString());
        }

        System.out.println("------\n");
        System.out.print("Introduzca el numero de la opcion\n->");
        String option_selected = entradaEscaner.nextLine();
        System.out.print("\nusted seleccionó ->" + response.get(parseInt(option_selected) - 1) + "
        String confirmation = entradaEscaner.nextLine();

        if (confirmation.toLowerCase().equals("s")) {
            Compania comp2 = (Compania) response.get(parseInt(option_selected) - 1);
            infoTransaction.add(comp2.getRFC());
            boolean salida = false;
            String numAcciones = null;
            String ofertaPorAccion = null;
            do {
```

- Execute_venta: Esta función permitirá a los usuarios realizar una venta, solicitando al servidor la información de lo que el usuario tiene.

```
private static void execute_venta(IRemoteProvince rp) throws RemoteException {
    try {
        ArrayList response = new ArrayList();
        ArrayList<String> infoTransaction = new ArrayList<String>();
        infoTransaction.add(userRFC);
        boolean salida = false;
        response = rp.getPortafolio(userRFC);
        String numAcciones = null;
        int valor;
        System.out.println("De que compañia desea vender:");
        imprimirPortafolio(rp);

        System.out.print("Introduzca el numero de la opcion\n->");
        String option_selected = entradaEscaner.nextLine();
        System.out.print("\nusted seleccionó ->" + response.get(parseInt(option_selected) - 1) + "\ne
        String confirmation = entradaEscaner.nextLine();

        if (confirmation.toLowerCase().equals("s")) {
            Transaccion comp2 = (Transaccion) response.get(parseInt(option_selected) - 1);
            infoTransaction.add(comp2.getRFCComp());
            do {

                System.out.print("cuantas acciones desea vender?\n->");
                valor = Integer.parseInt(entradaEscaner.nextLine());
                numAcciones = "-" + valor;
                if(valor > comp2.getAccionesOperadas()){
                    System.out.println("Por favor, ingresa un valor menor a " + comp2.getAccionesOpe
                }else{
                    salida = true;
```

- Execute_notificaciones: Cuando el usuario desee ver sus últimas transacciones, esta función le permitirá visualizar las transacciones que ha hecho (pujas ganadas, perdidas y ventas realizadas)

```
private static void execute_notificaciones(IRemoteProvince rp) throws RemoteException {
    ArrayList response = new ArrayList();
    response = rp.obtenerNotificaciones(userRFC, "comprado");

    System.out.println("\n°°°°°°°°Subastas ganadas°°°°°°°°");
    for (int i = 0; i < response.size(); i++) {
        System.out.println("->" + response.get(i).toString());
    }

    response = rp.obtenerNotificaciones(userRFC, "perdido");
    System.out.println("\n°°°°°°°°Subastas perdidas°°°°°°°°");
    for (int i = 0; i < response.size(); i++) {
        System.out.println("->" + response.get(i).toString());
    }
    System.out.println("\n°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°\n");

    response = rp.obtenerNotificaciones(userRFC, "vendido");
    System.out.println("\n°°°°°°°°Acciones vendidas°°°°°°°°");
    for (int i = 0; i < response.size(); i++) {
        System.out.println("->" + response.get(i).toString());
    }
    System.out.println("\n°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°°\n");
}
```

- getPortafolio: Esta función le permitirá a todos los usuarios obtener información general de la compra y venta de acciones.

```
private static void imprimirPortafolio(IRemoteProvince rp) throws RemoteException {
    ArrayList resp = rp.getPortafolio(userRFC);
    System.out.print("|||||||Portafolio|||||||\n");
    for (int i = 0; i < resp.size(); i++) {
        System.out.println((i + 1) + ".-" + resp.get(i).toString());
    }
    System.out.print("||||||||||||||||||||||||\n");
}
```

IRemoteProvince

El IRemoteProvince es la interfaz que funcionará con el RMI, con el fin de poder implementar el diseño por capas deseado. Como se puede observar, IRemoteProvince invoca todas las funciones que son necesarias para una interacción cliente servidor en esta aplicación.

```
public ArrayList showAll() throws RemoteException;
public ArrayList getPortafolio(String RFC) throws RemoteException;
public void generarTransaccion(Transaccion t) throws RemoteException;
public boolean check_user(String usuario, String contrasena) throws RemoteException;
public void startTransaction(ArrayList<String> infoTransaction)throws RemoteException;
public void enviarPropuesta(Transaccion t)throws RemoteException;
public ArrayList obtenerNotificaciones(String userRFC, String estado)throws RemoteException;
public ArrayList getTransacciones(String userRFC)throws RemoteException;
```

BankServer, ProvinceObject y BankRepository

El BankServer y ProvinceObject son dos clases que van de la mano. El primero funciona como medio para inicializar el servidor y que este pueda recibir peticiones.

BankServer

```
public class BankServer {

  public static void main(String[] args) {
    try {
      //Create and get reference to rmi registry
      Registry registry = LocateRegistry.createRegistry(1099);
      System.setProperty("java.rmi.server.hostname","192.168.1.74");

      //Instantiate server object
      ProvinceObject po = new ProvinceObject();

      //Register server object
      registry.rebind("Bank", po);
      System.out.println("BankServer is created!!!");
    } catch (Exception e) {
      System.out.println(e);
    }
  }
}
```

Es el servidor que estará esperando la llegada de peticiones

ProvinceObject

La siguiente clase invoca todas las funciones que se han ejecutado en el cliente, de manera que tomará la información de la solicitud y procederá a ejecutar cada una de las funciones del servidor, las cuales fueron solicitadas.

```
public class ProvinceObject extends UnicastRemoteObject implements IRemoteProvince {
private static ArrayList<Temporizador> ongoingTransactions = new ArrayList<Temporizador>();
private static final long serialVersionUID = 11L;

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

    @Override
    public ArrayList showAll() throws RemoteException {
      try {
        System.out.println("Invoke show all from " + getClientHost());
      } catch (ServerNotActiveException snae) {
        snae.printStackTrace();
      }
      return bancoRepository.showAll();
    }

    @Override
    public ArrayList getPortafolio(String RFC) throws RemoteException {
      try {
        System.out.println("Invoke get portafolio from " + getClientHost());
      } catch (ServerNotActiveException snae) {
        snae.printStackTrace();
      }
      return bancoRepository.getPortafolio(RFC);
    }
```

BankRepository

El BankRepository, es la clase donde todas las funciones de la aplicación son ejecutadas. Es decir, son las funciones del servidor para poder satisfacer las solicitudes del cliente.
El BankRepository establece las funciones necesarias para procesar las peticiones, de manera que ejecuta las funciones showAll() para obtener la información actual de las diferentes companias que venden acciones. Por el contrario, si el usuario quiere comprar o vender, se ejecutan las soliticudes correspondientes para poder realizar la compra o venta y, así, competir con los otros usuarios.

Por otro lado, algunas funciones realizan su comunicación con la tercera capa, la base de datos. Esta comunicación se realiza mediante el DBManager, el cual contiene funciones que realizan la conexión con la base de datos creada en MySQL.

```
public static ArrayList showAll() {
    ArrayList arr = new ArrayList();
    try {
        String QRY = "SELECT * FROM companias ORDER BY RFC";
        Connection con = DBManager.getInstance().getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(QRY);
        while (rs.next()) {
            Compania compania = new Compania();
            compania.setRFC(rs.getString("RFC"));
            compania.setNumAccionesTot(rs.getInt("numAccionesTot"));
            compania.setNumAccionesDisp(rs.getInt("numAccionesDisp"));
            compania.setValorActualAccion(rs.getFloat("valorActualAccion"));
            arr.add(compania);
        }
    } catch (SQLException se) {
        System.out.println(se);
    }
    return arr;
}
```

La función getPortafolio recibe el RFC del usuario del cual vamos a obtener el portafolio y se conecta a la base de datos para realizar sus funciones. Al final retorna un array con la lista de ítems que posee el usuario (según los requerimientos la tabla SQL debería llamarse usuarios, es por esto que se crea un objeto usuario y no un objeto item).

```
public static ArrayList getPortafolio(String RFC) {
    ArrayList arr = new ArrayList();
    try {
        String QRY = "SELECT * FROM usuarios WHERE RFCUsuario = '" + RFC + "'";
        Connection con = DBManager.getInstance().getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(QRY);
        while (rs.next()) {
            Usuario usuario = new Usuario();
            usuario.setRFCUsuario(rs.getString("RFCUsuario"));
            usuario.setRFCComp(rs.getString("RFCCompania"));
            usuario.setNumAcciones(rs.getInt("numAcciones"));
            usuario.setUltPrecioCompra(rs.getFloat("ultPrecioCompra"));
            arr.add(usuario);
        }
    } catch (SQLException se) {
        System.out.println(se);
    }
    return arr;
}
```

almacenarTransacciones recibe un objeto transacción, realiza una petición con los datos de la transacción e inserta en la base de datos dichos datos.

```
public static void almacenarTransacciones(Transaccion t){
      try {
        Connection con = DBManager.getInstance().getConnection();

        String QRY = "INSERT INTO notificaciones (RFCUsuario, RFCCOMP, fecha,"
        + "accionesOperadas, precioOperacion) values(?,?,?,?,?)";

        PreparedStatement pstmt = con.prepareStatement(QRY);
        pstmt.setString(1, t.getRFCUsuario());
        pstmt.setString(2, t.getRFCComp());
        pstmt.setDate(3, null);
        pstmt.setInt(4, t.getAccionesOperadas());
        pstmt.setFloat(5, t.getPrecioOperacion());

        pstmt.executeUpdate();

        pstmt.close();
    } catch(SQLException se){
        System.out.println(se);
    }
    System.out.println("Acción completada");
}
```

getTransacciones recibe un String que no es otra cosa que el un RFC de diez dígitos. Con esto se obtiene la lista de transacciones que ha realizado el usuario con esa RFC.

```
public static ArrayList getTransacciones(String RFC) {
    ArrayList arr = new ArrayList();
    try {
        String QRY = "SELECT * FROM transacciones WHERE RFCUsuario = '" + RFC + "'";
        Connection con = DBManager.getInstance().getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(QRY);
        while (rs.next()) {
            Transaccion transaccion = new Transaccion();
            transaccion.setRFCUsuario(rs.getString("RFCUsuario"));
            transaccion.setRFCComp(rs.getString("RFCComp"));
            transaccion.setAccionesOperadas(rs.getInt("accionesOperadas"));
            transaccion.setPrecioOperacion(rs.getFloat("precioOperacion"));
            arr.add(transaccion);
        }
    } catch (SQLException se) {
        System.out.println(se);
    }
    return arr;
}
```

generarTransaccion recibe una transacción y la inserta en la base de datos indicando la fecha y la hora en la que sucede la acción. Esto se utiliza para cuando se compra o se vende alguna acción.

```
public static void generarTransaccion(Transaccion t) {
    try {
      Connection con = DBManager.getInstance().getConnection();
      String SQL = "INSERT INTO transacciones (RFCUsuario, RFCComp, fecha,"
              + " accionesOperadas, precioOperacion) values(?,?,?,?,?)";

      PreparedStatement pstmt = con.prepareStatement(SQL);
      pstmt.setString(1, t.getRFCUsuario());
      pstmt.setString(2, t.getRFCComp());
      pstmt.setTimestamp(3, new java.sql.Timestamp(new java.util.Date().getTime()));
      pstmt.setInt(4, t.getAccionesOperadas());
      pstmt.setFloat(5, t.getPrecioOperacion());

      pstmt.executeUpdate();

      pstmt.close();
    } catch (SQLException se) {
      System.out.println(se);
    }
```

Check_user realiza una solicitud a la base de datos con el usuario y contraseña que recibe con la finalidad de garantizar la seguridad de los datos y que únicamente el usuario que posea dicho usuario y contraseña y sólo el pueda acceder al sistema.

```
public static boolean check_user(String usuario, String contrasena){
    boolean user_exist = false;
    ArrayList<String> users = new ArrayList<String>();

    try {
        String QRY = "SELECT * FROM users where usuario='"+usuario+"' AND contrasena='"+contrasena+"
        Connection con = DBManager.getInstance().getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(QRY);

        while (rs.next()) {
            users.add(rs.getString(1));
        }

        if(users.size() > 0){
            user_exist=true;
        }

    } catch (SQLException se) {
        System.out.println(se);
    }

    return user_exist;
}
```

actualizarAccionesDisponibles sustituye el último precio de compra que se haya realizado como el precio actual de la acción.

```
public static void actualizarAccionesDisponibles(Transaccion t) {
    try {

        String QRY = "UPDATE companias SET valorActualAccion =" + t + "WHERE RFC=" + t.getRFCComp();
        Connection con = DBManager.getInstance().getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(QRY);
    } catch(SQLException se){
        System.out.println(se);
    }
    System.out.println("Las acciones disponibles se han actualizado");
}
```

generarGnadorVenta ordena la lista de ArrayList que devuelve el objeto subastas cuando termina el tiempo de dos minutos con la finalidad de encontrar la venta con el menor precio.

```
public static void generarGanadorVenta(ArrayList propuestasCompras) throws SQLException{
    String estado = "vendido";
    ArrayList rechazados;


    Collections.sort(propuestasCompras);


    generarTransaccion((Transaccion) propuestasCompras.get(0));
    actualizarPortafolio((Transaccion) propuestasCompras.get(0));
    actualizarCompanias((Transaccion) propuestasCompras.get(0));
    almacenarNotificaciones((Transaccion)propuestasCompras.get(0), estado);

    propuestasCompras.remove(0);
    if(propuestasCompras.size() > 0){
        rechazados = propuestasCompras;
        estado = "perdido";
        for(int i = 0; i < rechazados.size(); i++){
            almacenarNotificaciones((Transaccion)rechazados.get(i), estado);
        }
    }
}
```

Posteriormente, genera la transacción de venta y genera las notificaciones de los involucrados.
ObtenerNotificaciones devuelve la lista de notificaciones de compra o venta que el usuario generó al momento de entrar en competencia y ganar o perder.

```
public static ArrayList obtenerNotificaciones(String RFC, String estado){
    ArrayList arr = new ArrayList();
    try {
        String QRY = "SELECT * FROM notificaciones WHERE RFCUsuario = '" + RFC + "' AND estado = '"
        Connection con = DBManager.getInstance().getConnection();
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery(QRY);
        while (rs.next()) {
            Notificaciones notificacion = new Notificaciones();
            notificacion.setRFCUsuario(rs.getString("RFCUsuario"));
            notificacion.setFecha(rs.getDate("fecha"));
            notificacion.setPrecioOperacion(rs.getFloat("precioOperacion"));
            notificacion.setEstado(rs.getString("estado"));
            arr.add(notificacion);
        }
    } catch (SQLException se) {
        System.out.println(se);
    }
    return arr;
```

La siguiente función se encarga de guardar en la base de datos las notificaciones que llegan al usuario de tal forma que este pueda verlas.

```
public static void almacenarNotificaciones(Transaccion t, String estado){
      try {
        Connection con = DBManager.getInstance().getConnection();

        String QRY = "INSERT INTO notificaciones (RFCUsuario, fecha,"
        + "precioOperacion, estado) values(?,?,?,?)";

        PreparedStatement pstmt = con.prepareStatement(QRY);
        pstmt.setString(1, t.getRFCUsuario());
        pstmt.setTimestamp(2, new java.sql.Timestamp(new java.util.Date().getTime()));
        pstmt.setFloat(3, t.getPrecioOperacion());
        pstmt.setString(4, estado);

        pstmt.executeUpdate();

        pstmt.close();
    } catch(SQLException se){
        System.out.println(se);
    }
    System.out.println("Acción completada");
}
```

generarGanadorCompra recibe una lista que devuelve el objeto subastas al término del temporizador y lo ordena de tal forma que puede elige al MAYOR postor por la acción.

```
public static void generarGanadorCompra(ArrayList publicaciones) throws SQLException{
    String estado = "comprado";
     ArrayList rechazados;


    Comparator<Integer> comparador = Collections.reverseOrder();
    Collections.sort(publicaciones, comparador);


    generarTransaccion((Transaccion) publicaciones.get(0));
    actualizarPortafolio((Transaccion) publicaciones.get(0));
    actualizarCompanias((Transaccion) publicaciones.get(0));
    almacenarNotificaciones((Transaccion)publicaciones.get(0), estado);
            //rechazados
    publicaciones.remove(0);
    if(publicaciones.size() > 0){
        rechazados = publicaciones;
        estado = "perdido";
        for(int i = 0; i < rechazados.size(); i++){
            almacenarNotificaciones((Transaccion)rechazados.get(i), estado);
        }
    }
}
```

enviarPropuesta intenta encontrar una lista de subastas activas y, en dado caso de no encontrarlo, abre una nueva.

```
//Ahora recibe cualquier tipo de transacción
public static int enviarPropuesta(Transaccion t){
    if(!subastas.isEmpty()){
        for (int i=0; i<subastas.size(); i++) {
            Subasta subastaActiva = (Subasta) subastas.get(i);
            ArrayList listaSubastasActivas = subastaActiva.getPropuestasCompras();
            for(int k=0; k<listaSubastasActivas.size(); k++) {
                Transaccion transaccion = (Transaccion) listaSubastasActivas.get(k);
                if((transaccion.getRFCComp().equals(t.getRFCComp())) && (transaccion.isCompra()
                    subastaActiva.setPropuestasCompras(t);
                    subastaActiva.startTimer();
                    subastas.set(i, subastaActiva);
                    return 1;
                }
            }
        }
    }
    Subasta subasta = new Subasta(id);
    id++;
    subasta.setPropuestasCompras(t);
    subasta.setCompra(t.isCompra());
    subasta.startTimer();
    subastas.add(subasta);
    return 0;
}
```

generarGanadores actualiza las salas o las listas y, posteriormente, llama a los métodos generadores de ganadores según sea compra o venta.

```
public static void generarGanadores(ArrayList propuestasCompras, String id, boolean isCompra) thro
    actualizarListas(id);
    if(isCompra){
        generarGanadorCompra(propuestasCompras);
    } else {
        generarGanadorVenta(propuestasCompras);
    }
}

public static void actualizarListas(String id){
    for (int i=0; i<subastas.size(); i++){
        Subasta subastaActiva = (Subasta) subastas.get(i);
        String subastaId = subastaActiva.getId();
        if (subastaId == id){
            subastas.remove(i);
        }
    }
}
```

DBManager

El DBManager se encarga de establecer las conexiones con la base de datos

```
public final class DBManager {

  private static DBManager _instance = null;
  private Connection _con = null;

  public DBManager() {
        _con = getMySQLConnection();
  }

  //Thread safe instatiate method
  public static synchronized DBManager getInstance() {
    if (_instance == null) {
      _instance = new DBManager();
    }
    return _instance;
  }

  public Connection getConnection() {
    return _con;
  }

  /**
   * Connection to MySQL Database
   */
```

Clases de objetos

Los objetos empleados para la aplicación fueron los siguientes:

Compania

```java
public class Compania implements Serializable {
    private static final long serialVersionUID = 1L;
    private String RFC;
    private int numAccionesTot;
    private int numAccionesDisp;
    private float valorActualAccion;

    public Compania() {

    }
```

Notificaciones

```java
public class Notificaciones implements Serializable {
    private static final long serialVersionUID = 1L;
    private String RFCUsuario;
    private Date fecha;
    private float precioOperacion;
    private String estado;

    public Notificaciones() {

    }
```

Usuario

```java
public class Usuario implements Serializable {
    private static final long serialVersionUID = 1L;
    private String RFCUsuario;
    private String RFCComp;
    private int numAcciones;
    private float ultPrecioCompra;

    public Usuario() {

    }
```

Transaccion

```java
public class Transaccion implements Serializable, Comparable<Transaccion>{
    private static final long serialVersionUID = 1L;
    private String RFCUsuario;
    private String RFCComp;
    private Date fecha;
    private int accionesOperadas;
    private float precioOperacion;

    public Transaccion() {

    }
```

Temporizador

```
public class Temporizador extends Thread {
    private Transaccion ongoing_transaction = null;
    private TimerTask transaction_timer = new Temporizador.MyTask();

    Temporizador(Transaccion t) {
        this.ongoing_transaction = t;
        run();
    }

    class MyTask extends TimerTask {
        public void run() {
            System.out.println("WIP");
        }
    }

    public void run()
    {
        Timer timer = new Timer();

        timer.schedule(transaction_timer, 2*60*1000);
    }
```

Subasta

```
public class Subasta {
    private Temporizador timer;
    private ArrayList propuestasCompras = new ArrayList();
    private boolean compra;
    private String id;
    private boolean flag = false;

    public Subasta(int id) {
        this.id = String.valueOf(id) + UUID.randomUUID();
    }

    public String getId() {
        return this.id;
    }

    public ArrayList getPropuestasCompras() {
        return propuestasCompras;
    }

    public void setPropuestasCompras(Transaccion t) {
        this.propuestasCompras.add(t);
    }

    public boolean isCompra() {
        return this.compra;
    }
```

Estas clases dieron forma a cada una de las solicitudes del cliente al servidor. Cada una está manejada como un objeto del tipo que indican, almacenando información (nombres, fechas, montos, etc). Pero, otros objetos emplean información necesaria para las competencias. Por ejemplo, subastas ejecuta el temporizador para poder definir un intervalo de tiempo destinado a la competencia de pujas. Una vez finalizado, se invoca a las funciones correspondientes para determinar el ganador y el perdedor.