Instituto Politécnico Nacional
"La Técnica al Servicio de la Patria"

INSTITUTO POLITÉCNICO NACIONAL
ESCOM

ESCUELA SUPERIOR DE CÓMPUTO

Vargas Hernández Carlo Ariel

2CM13

Práctica final de programación orientada a objetos

Proyecto: MiniNapster

18/06/2021

Introducción:

Breve historia de Napster: Napster fue lanzado en Mayo del 99. Era un software que permitía intercambiar archivos binarios a través de internet mediante una red peer-to-peer (P2P) que funciona sin estar centralizado, donde no hay servidores que manejen los clientes que se conecten, estos son sustituidos por nodos que son iguales e independieres si uno falla (de forma similar a como funciona bitcoin). Esta aplicación se usó por las personas para compartir música de manera no legal y masiva lo que hizo que los artistas demandaran a Napster, siendo la banda Metallica la primera. Aunque llegó a tener más de 26 millones de usuarios en su mejor momento termino por cerrar en julio de 2001 por una orden judicial.

Como vimos en la pequeña explicación de funcionamiento de Napster, se requiere conocimiento de máquinas remotas, clientes, hilos, eventos y demás para poder replicar un mininapster. Estos conocimientos los adquirimos a lo largo de todo el curso de programación orientada a objetos y los usaremos para programar una versión a escala de este software.

Desarrollo:

Para la creación de las conexiones entre maquinas se usaron sockets cliente y sockets servidor en forma P2P, también se usó JDBC para la conexión con base de datos, para las interfaces se usaron frames y eventos para botones y etiquetas usadas. Para explicar de manera efectiva las diferentes clases nos enfocaremos en la parte de servidor y cliente y en cómo se logró hacer el peer-to-peer. Pero primero revisaremos las clases que se hicieron para los objetos que usaremos en las diferentes clases y servidores.

Clase de usuario (User): Esta clase tiene diferentes métodos públicos de tipo string, uno void y un constructor para asignar u obtener diferentes datos que pertenecen al usuario (nombre, contraseña, etc). Esta clase implementa Serializable.

```
public class User implements Serializable {

    private String name, user, password, ipAddress;


    public User(String name, String user, String password, String
ipAddress) {…..}

    public String getName() {…..}

    public String getPassword() {…..}

    public String getIpAddress() {…..}

    public String getUser() {…..}

    public void setIpAddress(String string){…..}
```

```
}
```

Clase de petición (Request): Esta clase se usa para las peticiones de canciones, al igual que la anterior se implementa Serializable y, a parte del constructor, tiene varios métodos de tipo String que obtienen el nombre, artista, álbum y otros datos de la canción y los devuelve.

```
public class Request implements Serializable{

    private String title, artist, album, genere, user;

    private int maxSize;


    public Request(String title, String artist, String album, int
maxSize, String genere, String user){…..}

    public String getTitle(){…..}

    public String getArtist(){…..}

    public String getAlbum(){…..}

    public int getMaxSize(){…..}

    public String getGenere() {…..}

    public String gerUser(){…..}

}
```

Clase de canción (Song): Esta clase una vez más implementa Serializable, tiene su constructor que asignara los metadatos de la canción en el programa, de no encontrarlos estos datos se llenaran con valores por defecto como: “Artista desconocido” o “Sin género”. Hay varios métodos de tipo string que se dedican a obtener los datos de la canción para que estos campos (o los que se puedan) no sean llenados con los valores con defecto mediante los métodos vacíos (que se encargan de asignar dichos valores).

```
public class Song implements Serializable {

    private String title, artist, genere, album, user, name;

    private int songSize;


    public Song(String title, String artist, String genere, String album,
String user, int songSize, String name) {…..}

    public String getTitle() {…..}

    public String getArtist() {…..}
```

```
    public String getGenere() {…..}
    public String getAlbum() {…..}
    public String getUser() {…..}
    public int getSongSize() {…..}
    public String getName() {…..}
    public void setTitle(String string) {…..}
    public void setName(String string) {…..}
    public void setArtist(String string) {…..}
    public void setGenere(String string) {…..}
    public void setAlbum(String string) {…..}
    public void setUser(String string) {…..}
    public void setSongSize(int Int) {…..}
}
```

Una vez vistas estas clases que crean objetos de su propio tipo ahora veremos las clases de clientes y servidores que las utilizan, empezando por los servidores.

Clase para manejo de la base de datos (ServerBD): Es una clase que no tiene un método principal, pero contiene todos los métodos con los que se conecta, inserta, borra y consulta datos (de usuarios y canciones) en la base de datos, implementa Serializable. Los métodos que solicitan datos a la BD tienen uno o varios bloques try-catch ya que la conexión puede fallar, cada método (a excepción del método para conectar) reciben un dato del tipo User, Request o Song mencionados antes o también string con alguno de los campos necesarios para buscar dentro de la base de datos.

```
public class ServerDB implements Serializable {

    private Connection connection;
    private String driver = "com.mysql.jdbc.Driver",
            URL = "jdbc:mysql://localhost:3306/",
            password = "",
            username = "root",
            database = "dbservernapster";

    private static final String KEY_NME = "name", KEY_ATS = "artist",
```

```
            KEY_GNE = "genere", KEY_ABM = "album", KEY_USR = "user",
            KEY_PTH = "path", KEY_SZE = "size", KEY_TLE = "title";

    public void connect() {…..}
    public void addUser(User user) {…..}
    public void deleteUser(User user) {…..}
    public void addSong(ArrayList<Song> arraySongs) {…..}
    public Boolean verifyUser(User user) {…..}
    public String getIPAddress(String nickName) {…..}
    public ArrayList<Song> getSongs(Request request) {…..}
    public void sessionClose(String user) {…..}
    public void UpdateIPAddress(User user, String newIP) {…..}
    public boolean songAlreadyExist(Song song) {…..}
    public String replaceApostrophe(String string) {…..}
}
```

Hay varias clases servidor que si tienen una clase principal, estas clases se deben ejecutar para que funcione el software. Estas clases crean objetos de tipo sockets, serversockets y los flujos de entrada y salida (InputStream y OutputStream) junto con su puerto sin embargo, dentro de la clase principal hay un bloque try-catch con el código que describe las instrucciones que debe hacer este servidor. Cada una tiene un nombre específico sobre su función, estas clases son:

```
public class addClientServer implements Serializable{…..}
```

Añade un usuario nuevo a la base de datos

public class addSongServer{…..}

Añade a la base de datos las canciones que el usuario solicite

public class deleteSongs implements Serializable{…..}

Elimina canciones del usuario

```
public class deleteUserServer implements Serializable{…..}
```

Elimina un usuario en la base de datos

```
public class getIPAddress implements Serializable{…..}
```

Obtiene la IP del cliente

```
public class getSongsAvailable implements Serializable{…..}
```

Obtiene todas las canciones disponibles en la base de datos dados ciertos parámetros para buscar

```
public class getUserInfo implements Serializable{…..}
```

Obtiene los datos del usuario solicitado

Servidor del cliente: Para lograr el peet-to-peer se creó una clase llamada microServer que permitirá que el cliente se comporte como un servidor. Para lograr el comportamiento que tenía Napster, este servidor se usa para que otros usuarios puedan descargar la música que tiene otro usuario, para enviar esta canción se convierte a paquete de bits. Se crean los sockets necesarios y podemos observar que el flujo de entrada es un objeto de tipo Song (la canción que se solicita). Con un objeto de tipo file almacenamos la ruta de la canción usando preferencias. Se establece el tamaño de los paquetes de bits a 16*1024. Se crea un objeto de tipo FileOutputStream que recibe nuestro objeto tipo File. Mediante un ciclo while evaluamos el envío de los paquetes de bites al cliente (en la condición si lo obtenido es mayor a cero se evita el paquete). Finalmente se cierra la conexión. Todo esto está dentro de diferentes try-catch porque son propensos a fallar.

```
public class microServer implements Serializable {
    public static void main(String args[]) {
        ServerSocket serverSocket;
        ......
        try {
            ……
            while (true) {
                socket = serverSocket.accept();
                input = new ObjectInputStream(socket.getInputStream());
                song = (Song)input.readObject();
                File file = new File(new
preferencesClient().getPath(song.getUser()) + "/" + song.getName());
                long len = file.length();
                byte[] bytes = new byte[16 * 1024];
                InputStream in = new FileInputStream(file);
                OutputStream out = socket.getOutputStream();
                int count;
                while ((count = in.read(bytes)) > 0) {
```

```
                    out.write(bytes, 0, count);
                }
                socket.close();
                input.close();
                System.out.println("Done!");
            }
        } catch (IOException e) {…..} catch (Exception e) {….} finally {
            ……
            }
        }
    }
}
```

Cabe mencionar que todas estas clases imprimen en consola los errores, operaciones, inicio y final de las operaciones que realizan, se hace de esta manera para encapsular y ocultar estos detalles al usuario. Con los servidores y clases que usan estos servidores especificados y explicado el pequeño servidor del cliente que usa el cliente para logar el P2P queda explicar las demás clases el lado del cliente que se usan para que estos interactúen. A continuación estas clases:

```
public class closeSession implements Serializable, Runnable{…..}
```

Esta clase se encarga de cerrar la sesión, crea un socket cliente, crea su propio puerto de uso

```
public class preferencesClient {…..}
```

Esta clase es usada en el servidor del cliente microServer para obtener las preferencias de ruta

```
public class sendSongsData implements Serializable, Runnable{…..}
```

Esta clase se encarga de enviar los datos de la canción que el usuario quiera subir, crea un socket cliente, crea su propio puerto de uso

```
public class SendUserData implements Serializable, Runnable{…..}
```

Esta clase se encarga de enviar los datos de la canción que el usuario quiera subir, crea un socket cliente, crea su propio puerto de uso

```
public class SendUserDelete implements Serializable, Runnable{…..}
```

Esta clase se encarga de enviar los datos del usuario cuando se da de alta, crea un socket cliente, crea su propio puerto de uso

```
public class userLogin implements Serializable, Runnable{…..}
```

Esta clase se encarga de enviar los datos del usuario para iniciar sesión, crea un socket cliente, crea su propio puerto de uso

```
public class userRegistered implements Serializable, Runnable{…..}
```

Verifica si el usuario está registrado, crea un socket cliente, crea su propio puerto de uso.

Dentro de la clase downloadSong creamos dos sockets (socket y socketDownload) los flujos de entrada y salida, un hilo, además de lo necesario para conectarse como dos puertos, dirección IP, host y la canción que se quiere descargar. Dentro del cosntructor de esta clase es donde se inician los sockets, se obtienen los flijos y se inicia el hilo.

Al iniciar el hilo, este nos envía una dirección IP y se inicia el método downloadingSong que recibe un nuevo host en forma de string al que se conecta el cliente para descargar la canción (el nuevo host es otro cliente), Se crea un objeto de tipo FileOutputStream al que le pasaremos el nombre de la canción + .mp3 (ejemplo: La cucaracha.mp3), al igual que en la clase microServer se establece el tamaño de los paquetes de bits a 16*1024 y se usa el mismo while para enviar los paquetes siempre y cuando sean mayores a cero.

```
public class downloadSong implements Runnable, Serializable {

……
    public downloadSong(Song song) {
        System.out.println("downloadSongs");
        this.song = song;
        try {
            System.out.println("Creating thread...");
            thread = new Thread(this);
            System.out.println("Creating socket...");
            socket = new Socket(host, port);
            System.out.println("Creating I/O");
            output = new ObjectOutputStream(socket.getOutputStream());
            input = new ObjectInputStream(socket.getInputStream());
            System.out.println("Thread ready");
            thread.start();
        } catch (IOException e) {…..} catch (Exception e) {…..}
```

```
    }

    public void downloadingSong(String newHost) {
        ……..
        try {
            socketDownload = new Socket(newHost, otherPort);
            output = new
ObjectOutputStream(socketDownload.getOutputStream());
            output.writeObject(song);
            InputStream in = socketDownload.getInputStream();
            System.out.println("microServer.java user connected");
            FileOutputStream out = new FileOutputStream(song.getName() +
".mp3");
            byte[] bytes = new byte[16 * 1024];

            int count;
            while ((count = in.read(bytes)) > 0) {
                out.write(bytes, 0, count);
            }
            //output.close();
            out.close();
            output.close();
            System.out.println("Done!");
        } catch (IOException e) {…..} catch (Exception e) {…..}
    }

    @Override
    public void run() {
        System.out.println("downloadSong.java, run method starts!!!");
        try {
            output.writeObject(song.getUser());
            ipaddress = (String) input.readObject();
            System.out.println("The ip adress is: " + ipaddress);
```

```
            socket.close();
            downloadingSong(ipaddress);
        } catch (IOException e) {…..} catch (Exception e) {…..}
    }
}
```

Cabe mencionar que al igual que las clases de los servidores, estas clases cliente mostraran en consola las acciones realizadas por los métodos, pero si algo falla se le notificara al usuario por medio de pantallas de dialogo en la interfaz gráfica. Hay 3 interfaces gráficas, la primera que es para iniciar sesión, la segunda que es para borrar un usuario, la tercera que es para registrarte y la cuarta en donde se ven las canciones y demás después de haber iniciado sesión.

Inicio de sesión: En esta interfaz hay dos campos, el de usuario y contraseña junto con un botón que dispara el listener para comprobar si el usuario está registrado, de no estarlo hay una etiqueta que te lleva a la pantalla para registrarte y uno que lleva a la pantalla de borrar cuenta.

Registro de usuario: Tiene el campo de texto de nombre, nickname, contraseña y confirmar contraseña además de un botón que dispara un listener que comprueba que el Nick no haya sido ya registrado y de no estarlo registra al usuario. Además hay una etiqueta que regresa a la anterior pantalla.

Borrado de usuario: Tiene los mismos campos que la pantalla de ingresar con la diferencia que solo tiene una etiqueta para regresar a la pantalla de inicio de sesión. Cuando se presiona el botón de borrar usuario primero se comprueba si existe y si existe se borra.

Canciones: En esta pantalla se pueden subir canciones mediante el botón share, buscar canciones buscando alguna característica de estos llenando los campos de artista, álbum, género o nombre de la canción una vez presionado el botón buscar, de no existir dicha canción se notifica y de existir se ve en una tabla con barra de desplazamiento. También hay una etiqueta que indica el nickname del usuario en la parte superior derecha.

Los ActionListener de cada botón tienen el nombre del botón más actionListener, se le pasan los objetos necesarios y se llaman a las funciones de servidor o de cliente antes mencionadas. En el caso de la tabla del frame de canciones obtiene los datos de la base de datos una vez presionado el botón search y dentro de esta tabla con el click derecho se muestra el download para descargar la canción. Ejemplo:

```
private void BTN_iniciaSesionActionPerformed(java.awt.event.ActionEvent
evt) {

        ……

        try {

            if (!user.equals("") && !password.equals("")) {

                User mUser = new User("", user, password, "");

                System.out.println(…..);

                coonectSocket(mUser);

                BTN_iniciaSesion.setEnabled(false);

            }

        } catch (Exception e) {…..}

    }
```

Base de datos: La base de datos dbservernapster (hecha con WampServer) es muy sencilla, consiste en una tabla para los usuarios y una tabla para las canciones que estos usuarios suban. Estas dos tablas tienen los campos mostrados en el diagrama que corresponden a los datos solicitados por el programa a usuarios o que se obtienen de las canciones. La relación de la tabla es que un usuario puede tener una o varias canciones.

Diagrama de la base de datos:

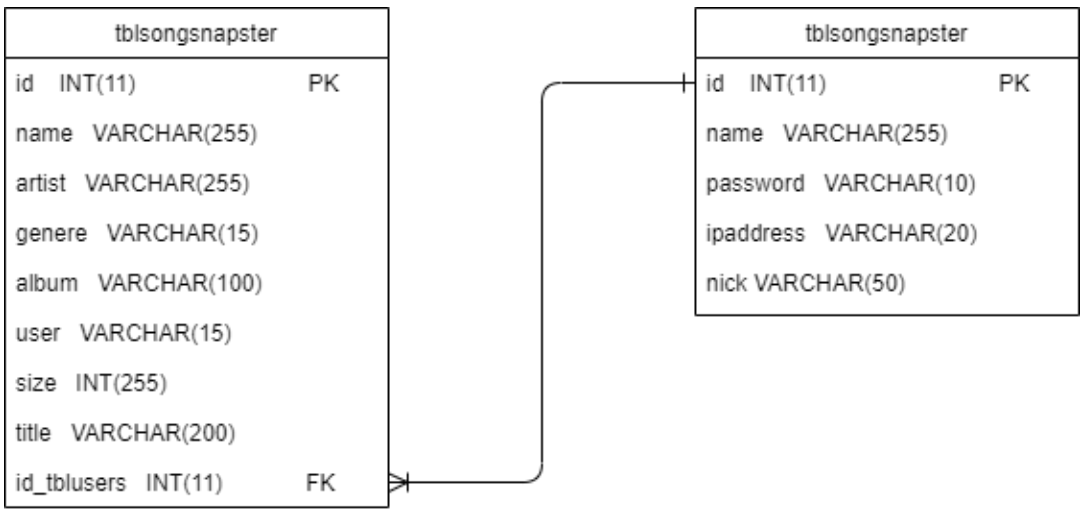

Diagrama de clases: