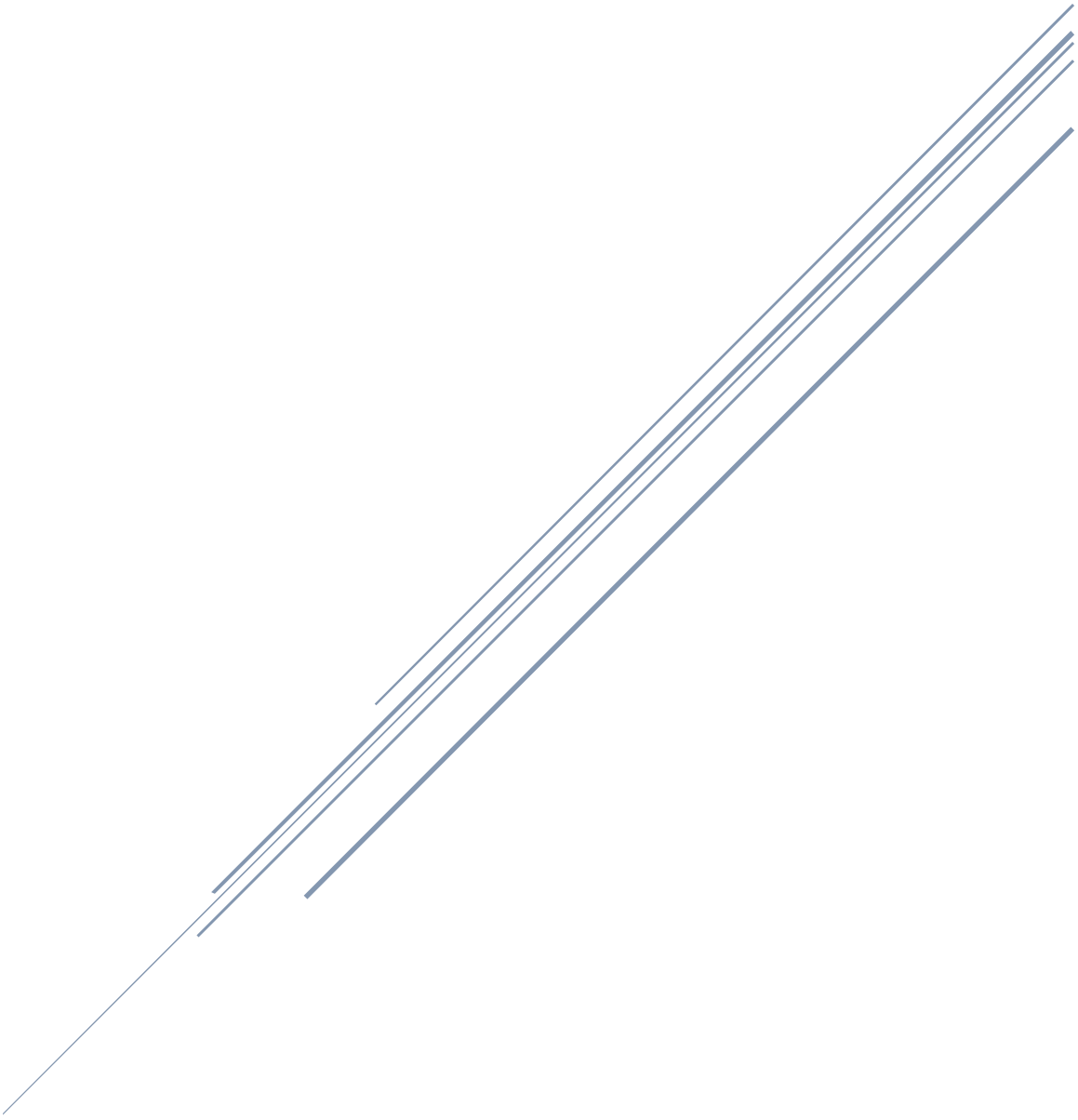
Tecnologías para el Desarrollo de Aplicaciones para Dispositivos Móviles. Parte 3. Hardware y Redes



Estudiante Abdiel Contreras.

Práctica Final Desarrollo de un Chat: Consiste en crear una aplicación tipo chat que necesariamente tiene que emplear la filosofía cliente/servidor implementada mediante Sockets tradicionales de Java en Android.

Requerimientos mínimos de la aplicación:

- ✓ El servidor acepta la conexión de un cliente. --→ SI ✓
- ✓ Debes ser capaz de poder comunicar un dispositivo real con otro y enviar secuencias de texto. --→ SI ✓
- ✓ Los dos equipos podrán tanto leer como escribir mensajes. -→ SI ✓
- ✓ Como la aplicación debe funcionar en dispositivos reales, además del código fuente, se proporcionará el ejecutable con las instrucciones de instalación: mínima versión de Android, SDK, etc… -→ SI ✓
- ✓ El código desarrollado debe estar más o menos comentado (en castellano), explicando lo que realiza las funciones importantes implementadas por el alumno para el desarrollo de la aplicación concreta (no las clases o funciones genéricas). → SI ✓

  Enriquecimientos:

  - ✓ Comentar las funciones del código facilitado y explicar lo que hace. → SI ✓
  - ✓ Enriquecimiento visual de la app: globos conversación, colores, etc.-→ SI ✓
  - ✓ Múltiples clientes conectados a un único servidor (sala de chat).-→ SI ✓
  - ✓ Envío de conversación cifrada.---→ NO ✗
  - ✓ Posibilidad de rechazar conexión por parte del servidor. --→ NO ✗
  - ✓ Validación de datos (puerto, IP).-→ SI ✓
  - ✓ Posible descubrimiento de usuarios (servidor).-- → NO ✗

- ✓ Envío de imágenes, además de texto, entre el cliente y servidor.-- → SI ✓

- ✓ Mostrarlo a los compañeros y explicar su desarrollo en la clase. ---→ SI ✓

- ✓ El alumno puede proponer otras mejoras al profesor.-- → SI ✓

Introducción: Aplicación de Chat basada en Cliente/Servidor con Sockets en Java para Android

El desarrollo de una aplicación de chat basado en la arquitectura cliente/servidor es un proyecto que ejemplifica los principios fundamentales de la comunicación en red. Este tipo de sistema sigue un modelo en el que se caracteriza por un servidor centralizado gestiona las conexiones y la comunicación entre múltiples clientes. La implementación mediante Sockets en Java permite establecer una comunicación en tiempo real utilizando el protocolo TCP/IP, la cual garantiza un canal confiable para la comunicación y para el intercambio de cualquier tipo información, sea texto, imágenes, documentos entre otros, esto será permitido entre distintos dispositivos.

Resumen del proyecto:

La aplicación consta de dos componentes principales:

El servidor:

Es el núcleo central del sistema, el cual se encarga de escuchar y aceptar conexiones entrantes desde múltiples clientes. Su función principal y la cual es encargada es la de gestionar la distribución de mensajes, asegurándose de que los datos enviados por un cliente lleguen a los destinatarios adecuados. Este servidor se implementa utilizando sockets del paquete estándar de Java (java.net.ServerSocket y Socket), lo que permite establecer conexiones de red de manera eficiente y confiable.

El cliente:

El cliente es la parte del sistema que establece una conexión con el servidor y envía/recibe mensajes en tiempo real. Utilizando un socket, el cliente se conecta al servidor a través de la red, facilitando la comunicación bidireccional entre ambos. En java, esta funcionalidad se implementa mediante la clase java.net.Socket, que permite al cliente conectar con el servidor específicamente la dirección IP y el puerto correspondiente. Una vez establecida la conexión, el cliente puede enviar

mensajes de texto, imágenes entre otros, recibir una respuesta del servidor y a su vez mantener una interacción constante y eficiente.

La aplicación desarrollada puede enviar y recibir mensajes en tiempo real utilizando una interfaz intuitiva. Los mensajes del cliente se transmiten al servidor, que los redirige a otros clientes conectados.

Objetivos principales:

- ✓ Diseñar un sistema basado en la arquitectura cliente/servidor.
- ✓ Implementar la comunicación en tiempo real utilizando Java Sockets.
- ✓ Permitir la transmisión de datos de manera continua entre cliente y el servidor.
- ✓ Integrar las capacidades de Android para crear una interfaz de usuario amigable y eficiente.
- ✓ Garantizar la confiabilidad y la gestión de múltiples usuarios simultáneamente.

Características clave:

- ✓ Conexión persistente: Los sockets permiten mantener una comunicación continua entre cliente y servidor.
- ✓ Multicliente: El servidor debe gestionar múltiples conexiones simultáneamente mediante hilos.
- ✓ Gestión de mensajes: El servidor distribuye los mensajes recibidos a los clientes correspondientes.
- ✓ Interfaz amigable: La aplicación Android presenta una experiencia intuitiva para los usuarios del chat.

Este proyecto es una práctica integral de la programación en red y el desarrollo móvil, donde se combinan conceptos fundamentales de comunicación cliente/servidor, multihilos y diseño de aplicaciones Android.

Instrucciones de instalación: mínima versión de Android, SDK:

- ✓ Minimo sdk api 24 android 7.0
- ✓ Grdle version 8.0
- ✓ Compile SDK versión 34(API 34)

Para la instalación de la APP:

- ✓ Este código servidor acepta muchos clientes conectados, envía msj y recibe, al igual que imágenes.
- ✓ Para probar instale las apk de cliente en distintos móviles y la apk del servidor en un móvil (Las APK estarán en la carpeta Abdiel Contreras y comprimida, también se aportarán los códigos Clientes y Servidor)
- ✓ primero proceda a iniciar el servidor luego inicie todos los clientes que quiera y luego a chatear entre clientes.

PERMISOS: Usados en el Cliente y Servidor:

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

DEPENDENCIAS: Usados en el Cliente y Servidor:

```
dependencies {
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.10.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
}
```

Código del servidor:

Cifrado de un mensaje usando AES, es una implementación de funciones para cifrar y descifrar mensajes usando el algoritmo **AES** (Advanced Encryption Standard) en Java, utilizando la librería **javax.crypto**.

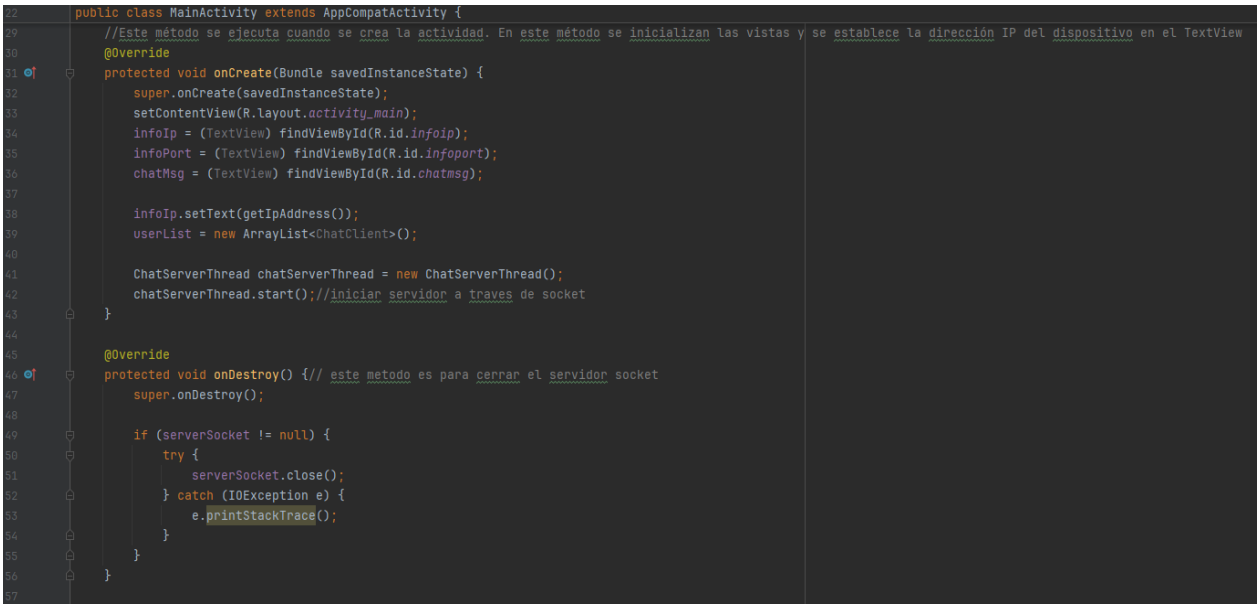
- ✓ El cliente y el servidor necesitan compartir la misma clave secreta para poder cifrar y descifrar correctamente los mensajes.
- ✓ El cliente enviará un mensaje cifrado al servidor, el cual lo descifrará.
- ✓ El servidor procesará el mensaje y enviará una respuesta cifrada al cliente.
- ✓ El cliente recibirá la respuesta cifrada y la descifrará.

Sabiendo ésta metodología se intentó implementar estos conceptos en el proyecto, pero No se pudo implementar al 100% debido a que solo cifraba y descifraba solo textos cortos, por consecuencia tuve que quitar el cifrado, de igual manera muestro mi código:

MainActivity.java | CryptoUtils.java

```java
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class CryptoUtils {  no usages
    // Generar una clave secreta AES de 128 bits
    public static SecretKey generateAESKey() throws Exception {  no usages
        KeyGenerator keyGenerator = KeyGenerator.getInstance( algorithm: "AES");
        keyGenerator.init( keysize: 128); // Tamaño de clave de 128 bits para mayor seguridad
        return keyGenerator.generateKey();
    }

    // Cifrar un mensaje con AES
    public static String encryptAES(String message, SecretKey key) throws Exception {  no usages
        Cipher cipher = Cipher.getInstance( transformation: "AES");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] encrypted = cipher.doFinal(message.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }

    // Descifrar un mensaje con AES
    public static String decryptAES(String encryptedMessage, SecretKey key) throws Exception {  no usages
        Cipher cipher = Cipher.getInstance( transformation: "AES");
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] decrypted = cipher.doFinal(Base64.getDecoder().decode(encryptedMessage));
        return new String(decrypted);
    }

    // Convertir una cadena a una clave AES (por ejemplo, de una contraseña)
    public static SecretKey getAESKeyFromString(String keyStr) {  no usages
        return new SecretKeySpec(keyStr.getBytes(),  algorithm: "AES");
    }
}
```

Este **método void onCreate** el cual se ejecuta cuando se crea la actividad. En este método se inicializan las vistas y se establece la dirección IP del dispositivo en el TextView. De igual manera se visualiza el método onDestroy() el cual es el encargado de cerrar correctamente el servidor del socket cuando la actividad o componente se destruye, cabe mencionar que es un método de ciclo de vida de la clase o service en Android.



```
public class MainActivity extends AppCompatActivity {
    //Este método se ejecuta cuando se crea la actividad. En este método se inicializan las vistas y se establece la dirección IP del dispositivo en el TextView
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        infoIp = (TextView) findViewById(R.id.infoip);
        infoPort = (TextView) findViewById(R.id.infoport);
        chatMsg = (TextView) findViewById(R.id.chatmsg);

        infoIp.setText(getIpAddress());
        userList = new ArrayList<ChatClient>();

        ChatServerThread chatServerThread = new ChatServerThread();
        chatServerThread.start();//iniciar servidor a traves de socket
    }

    @Override
    protected void onDestroy() {// este metodo es para cerrar el servidor socket
        super.onDestroy();

        if (serverSocket != null) {
            try {
                serverSocket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Este método **void run()** Se encarga de aceptar conexiones de clientes y crear hilos para manejar cada conexión entrante.

```
    private class ChatServerThread extends Thread {// Esta clase interna hereda de Thread y se utiliza para crear y administrar el servidor socket  2 usages
        @Override
        public void run() {// Este metodo Se encarga de aceptar conexiones de clientes y crear hilos para manejar cada conexion entrante.
            Socket socket = null;

            try {
                serverSocket = new ServerSocket(SocketServerPORT);
                MainActivity.this.runOnUiThread(new Runnable() {

                    @Override
                    public void run() {
                        infoPort.setText("Puerto Asignado: "
                                + serverSocket.getLocalPort());
                    }
                });

                while (true) {
                    socket = serverSocket.accept();
                    ChatClient client = new ChatClient();
                    userList.add(client);
                    ConnectThread connectThread = new ConnectThread(client, socket);
                    connectThread.start();
                }

            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                if (socket != null) {
                    try {
                        socket.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }

        }
    }
```

Este **método run()** se encarga de iniciar el servidor del socket, el cual escucha un puerto especifico.

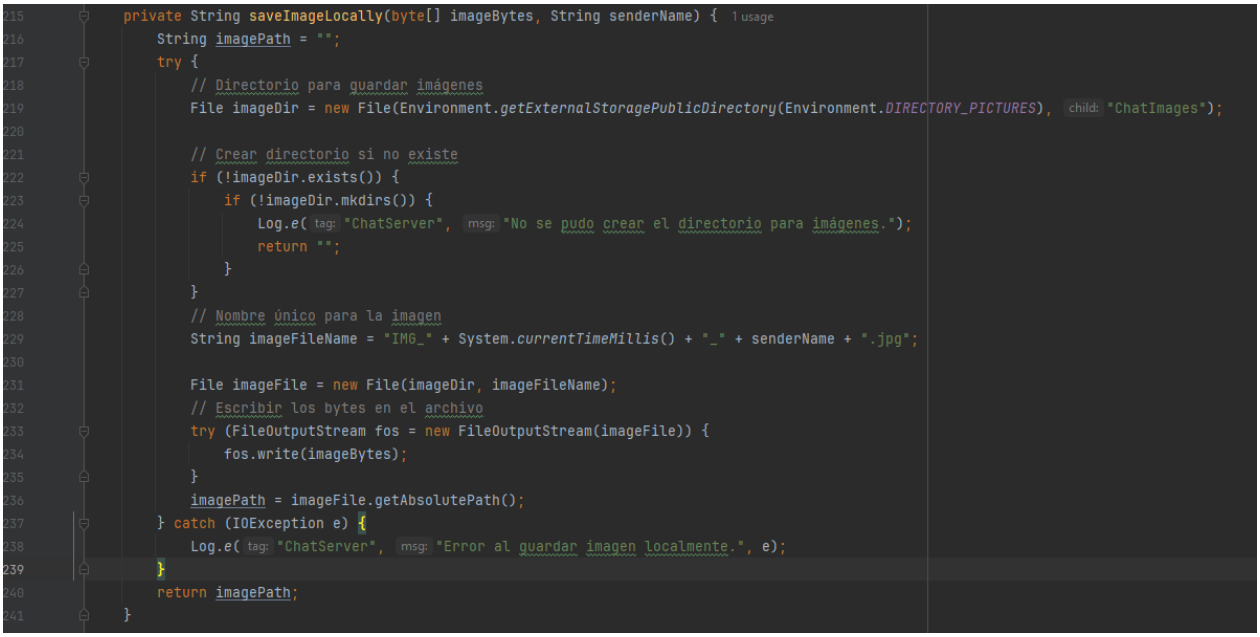
Se podría decir que en general se encarga de crear un servidor de sockets que acepta múltiples conexiones de clientes. Para cada cliente que se conecta, se crea un hilo separado para manejar la comunicación con ese cliente, lo que permite manejar múltiples clientes simultáneamente de manera eficiente.

Este es el **metodo void broadcastImage():** Envía una imagen a todos los clientes conectados, exceptuando al remitente, y actualiza el historial de chat en la interfaz de usuario para reflejar que una imagen fue enviada.

```
182     private void broadcastImage(String sender, byte[] imageBytes) {  1 usage
183         for (ChatClient client : userList) {
184             try {
185                 if (!client.name.equals(sender)) {
186                     DataOutputStream dataOutputStream = new DataOutputStream(client.socket.getOutputStream());
187                     dataOutputStream.writeUTF( str: "IMAGE");
188                     dataOutputStream.writeInt(imageBytes.length);
189                     dataOutputStream.write(imageBytes);
190                     dataOutputStream.flush();
191                 }
192             } catch (IOException e) {
193                 e.printStackTrace();
194             }
195         }
196         msgLog += sender + " envió una imagen.\n";
197         MainActivity.this.runOnUiThread(() -> chatMsg.setText(msgLog));
198     }
199
200     private void broadcastMsg(String sender, String message) {  3 usages
201         for (ChatClient client : userList) {
202             try {
203                 if (!client.name.equals(sender)) {
204                     DataOutputStream dataOutputStream = new DataOutputStream(client.socket.getOutputStream());
205                     dataOutputStream.writeUTF( str: "TEXT");
206                     dataOutputStream.writeUTF( str: sender + ": " + message);
207                     dataOutputStream.flush();
208                 }
209             } catch (IOException e) {
210                 e.printStackTrace();
211             }
212         }
213     }
```

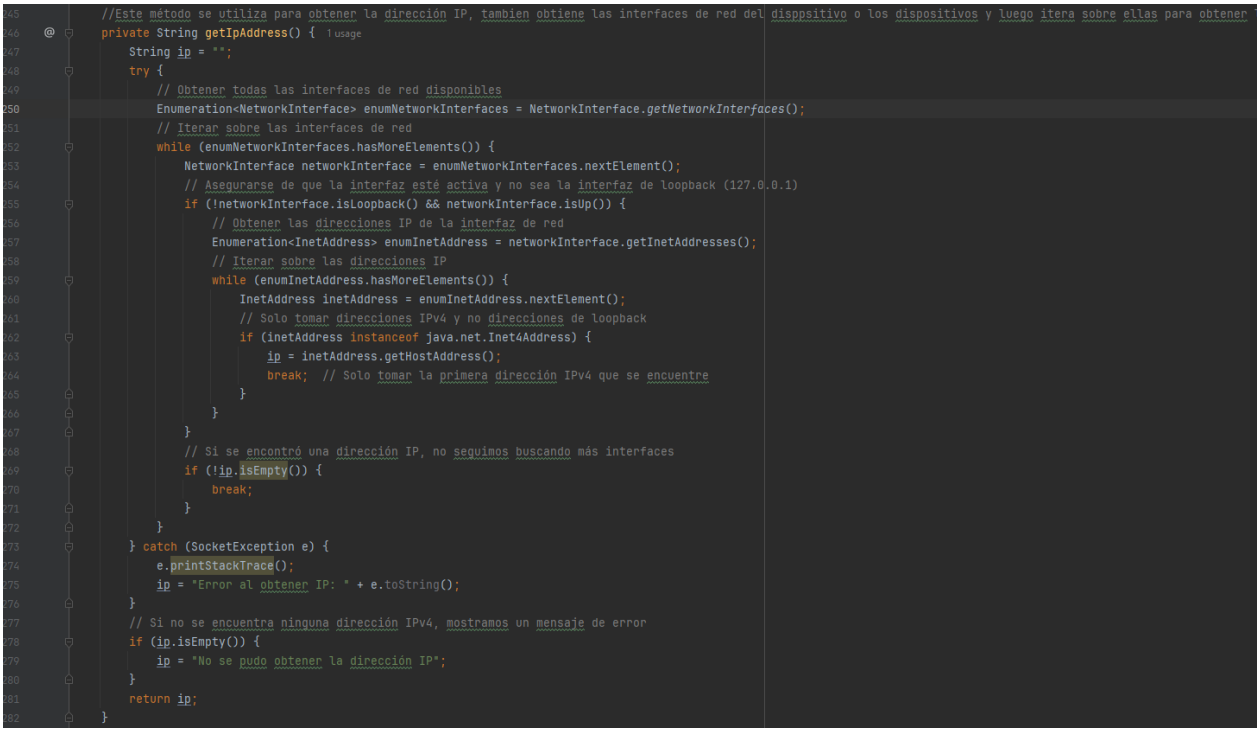
El **método broadcastMsg** permite enviar un mensaje de texto a todos los clientes conectados a un servidor de chat, excluyendo al remitente. Se asegura de que cada cliente reciba el mensaje correctamente, utilizando un DataOutputStream para transmitir el mensaje en formato UTF-8. Además, maneja excepciones en caso de fallos en la transmisión.

El método **saveImageLocally()** se encarga de guardar una imagen (representada como un array de bytes) en el almacenamiento local del dispositivo Android, dentro del directorio de imágenes públicas. El nombre del archivo de la imagen es generado de forma única utilizando la hora actual y el nombre del remitente.



```
private String saveImageLocally(byte[] imageBytes, String senderName) {  1 usage
    String imagePath = "";
    try {
        // Directorio para guardar imágenes
        File imageDir = new File(Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_PICTURES), child: "ChatImages");

        // Crear directorio si no existe
        if (!imageDir.exists()) {
            if (!imageDir.mkdirs()) {
                Log.e( tag: "ChatServer",  msg: "No se pudo crear el directorio para imágenes.");
                return "";
            }
        }
        // Nombre único para la imagen
        String imageFileName = "IMG_" + System.currentTimeMillis() + "_" + senderName + ".jpg";

        File imageFile = new File(imageDir, imageFileName);
        // Escribir los bytes en el archivo
        try (FileOutputStream fos = new FileOutputStream(imageFile)) {
            fos.write(imageBytes);
        }
        imagePath = imageFile.getAbsolutePath();
    } catch (IOException e) {
        Log.e( tag: "ChatServer",  msg: "Error al guardar imagen localmente.", e);
    }
    return imagePath;
}
```

El método **getIpAddress** obtiene la dirección IP local del dispositivo. Este método busca las interfaces de red disponibles en el dispositivo, verifica que no sean interfaces de loopback (como la dirección 127.0.0.1), y luego busca una dirección IPv4 activa para devolverla.



```
//Este método se utiliza para obtener la dirección IP, tambien obtiene las interfaces de red del dispositivo o los dispositivos y luego itera sobre ellas para obtener
@ private String getIpAddress() {  1 usage
    String ip = "";
    try {
        // Obtener todas las interfaces de red disponibles
        Enumeration<NetworkInterface> enumNetworkInterfaces = NetworkInterface.getNetworkInterfaces();
        // Iterar sobre las interfaces de red
        while (enumNetworkInterfaces.hasMoreElements()) {
            NetworkInterface networkInterface = enumNetworkInterfaces.nextElement();
            // Asegurarse de que la interfaz esté activa y no sea la interfaz de loopback (127.0.0.1)
            if (!networkInterface.isLoopback() && networkInterface.isUp()) {
                // Obtener las direcciones IP de la interfaz de red
                Enumeration<InetAddress> enumInetAddress = networkInterface.getInetAddresses();
                // Iterar sobre las direcciones IP
                while (enumInetAddress.hasMoreElements()) {
                    InetAddress inetAddress = enumInetAddress.nextElement();
                    // Solo tomar direcciones IPv4 y no direcciones de loopback
                    if (inetAddress instanceof java.net.Inet4Address) {
                        ip = inetAddress.getHostAddress();
                        break;  // Solo tomar la primera dirección IPv4 que se encuentre
                    }
                }
            }
            // Si se encontró una dirección IP, no seguimos buscando más interfaces
            if (!ip.isEmpty()) {
                break;
            }
        }
    } catch (SocketException e) {
        e.printStackTrace();
        ip = "Error al obtener IP: " + e.toString();
    }
    // Si no se encuentra ninguna dirección IPv4, mostramos un mensaje de error
    if (ip.isEmpty()) {
        ip = "No se pudo obtener la dirección IP";
    }
    return ip;
}
```

El **metodo getIpAddress:** Es el que obtiene la dirección IP local de un dispositivo buscando las interfaces de red activas y válidas. Solo devuelve una dirección IPv4 (y no direcciones de loopback o IPv6). Si encuentra una dirección IP válida, la retorna; si no, devuelve un mensaje de error indicando que no se pudo obtener la IP. Además, maneja excepciones que podrían ocurrir durante el proceso de obtención de la dirección IP.

Código Cliente:

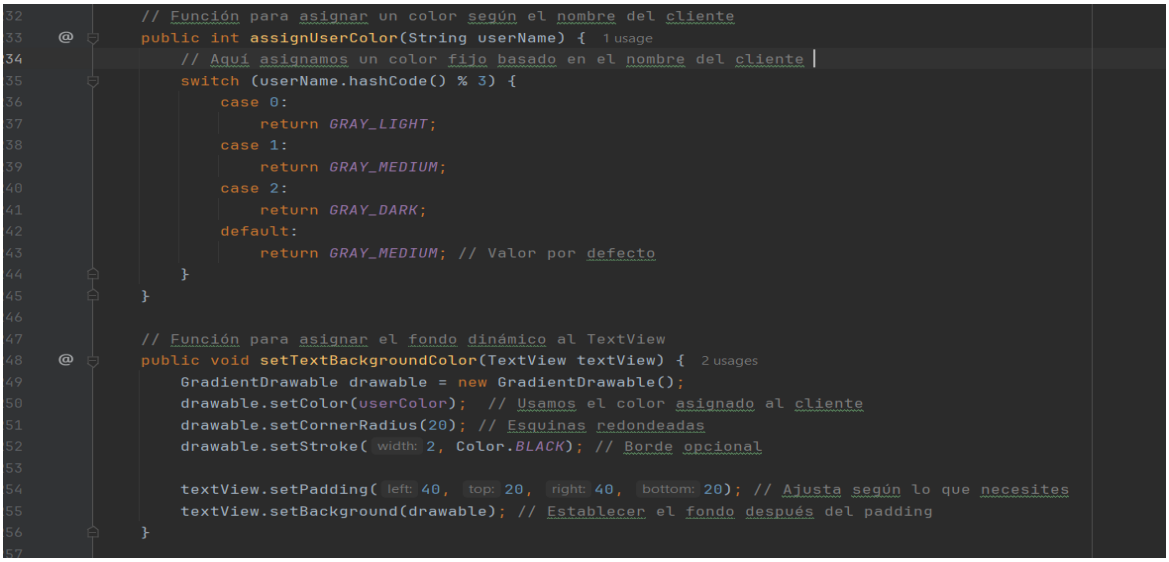
El método **isValidIPAddress():** valida si una cadena de texto proporcionada es una dirección IP válida en formato **IPv4**. Para ello, utiliza una expresión regular que verifica que la cadena cumple con el formato estándar de una dirección IPv4. Si la IP es válida según este formato, el método devuelve **true**, de lo contrario, devuelve **false**.

El método **isServerAvailable()** verifica si un servidor está accesible en una dirección y puerto específicos. Intenta abrir una conexión de socket y, si se puede establecer, devuelve **true**. Si no se puede conectar, captura la excepción **IOException** y devuelve **false**. Este método se utiliza típicamente para comprobar si un servidor está en línea y disponible para recibir conexiones en un puerto específico.

```
216     // Método para verificar si el servidor está disponible
217     private boolean isServerAvailable(String address, int port) {  1 usage
218         try (Socket socket = new Socket(address, port)) {
219             return true; // Conexión exitosa
220         } catch (IOException e) {
221             return false; // Error al conectar
222         }
223     }
224
225     // Método para validar una dirección IP
226 @   private boolean isValidIPAddress(String ip) {  1 usage
227         // Expresión regular para validar IPv4
228         String ipPattern = "^((25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\\.){3}(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)$";
229         return ip.matches(ipPattern);
230     }
231
```

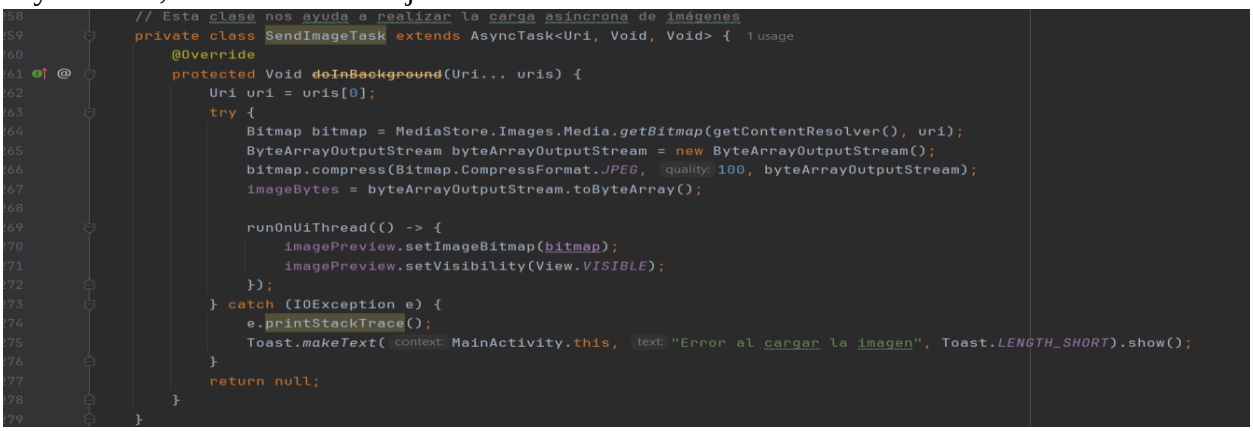
La función **assignUserColor:** Lo que hace es asignar un color basado en el nombre del cliente, solo marca tres tipos de colores para simular la conversión, se pueden anexar más colores si se requiere.

La función **setTextBackgroundColor:** Es la encargada de configurar un fondo dinámico único y estilizado para un TextView usando ese color asignado.



```
// Función para asignar un color según el nombre del cliente
public int assignUserColor(String userName) {  1 usage
    // Aquí asignamos un color fijo basado en el nombre del cliente
    switch (userName.hashCode() % 3) {
        case 0:
            return GRAY_LIGHT;
        case 1:
            return GRAY_MEDIUM;
        case 2:
            return GRAY_DARK;
        default:
            return GRAY_MEDIUM; // Valor por defecto
    }
}

// Función para asignar el fondo dinámico al TextView
public void setTextBackgroundColor(TextView textView) {  2 usages
    GradientDrawable drawable = new GradientDrawable();
    drawable.setColor(userColor);  // Usamos el color asignado al cliente
    drawable.setCornerRadius(20); // Esquinas redondeadas
    drawable.setStroke( width: 2, Color.BLACK); // Borde opcional

    textView.setPadding( left: 40,  top: 20,  right: 40,  bottom: 20); // Ajusta según lo que necesites
    textView.setBackground(drawable); // Establecer el fondo después del padding
}
```

La clase **SendImageTask** es una subclase de AsyncTask que se utiliza para cargar una imagen de manera asíncrona en segundo plano, lo que evita bloquear el hilo principal de la interfaz de usuario (UI) mientras se realiza la carga. También cabe mencionar que la comprime y la guarda como bytes. Luego, actualiza la interfaz de usuario mostrando la imagen en un ImageView. Si hay un error, muestra un mensaje de advertencia.



```
// Esta clase nos ayuda a realizar la carga asincrona de imágenes
private class SendImageTask extends AsyncTask<Uri, Void, Void> {  1 usage
    @Override
    protected Void doInBackground(Uri... uris) {
        Uri uri = uris[0];
        try {
            Bitmap bitmap = MediaStore.Images.Media.getBitmap(getContentResolver(), uri);
            ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();
            bitmap.compress(Bitmap.CompressFormat.JPEG,  quality: 100, byteArrayOutputStream);
            imageBytes = byteArrayOutputStream.toByteArray();

            runOnUiThread(() -> {
                imagePreview.setImageBitmap(bitmap);
                imagePreview.setVisibility(View.VISIBLE);
            });
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText( context: MainActivity.this,  text: "Error al cargar la imagen", Toast.LENGTH_SHORT).show();
        }
        return null;
    }
}
```

Metodo sendImage: Este método toma una imagen en formato de bytes, la envía al servidor en tres pasos: primero un mensaje indicando que es una imagen, luego el tamaño de la imagen, y finalmente los propios bytes de la imagen. Si hay un error durante el proceso, lo maneja imprimiendo la excepción.

Metodo sendMsg: Este método envía un mensaje de texto al servidor en dos pasos: primero indica que es un mensaje de tipo "TEXT" y luego envía el contenido del mensaje. Si ocurre algún error, lo maneja imprimiendo la excepción.

```java
// Método para enviar una imagen al servidor
private void sendImage(byte[] imageBytes) {  1 usage
    try {
        dataOutputStream.writeUTF( str: "IMAGE");
        dataOutputStream.writeInt(imageBytes.length);
        dataOutputStream.write(imageBytes);
        dataOutputStream.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}


// Método para enviar un mensaje de texto al servidor
private void sendMsg(String msg) {  1 usage
    try {
        dataOutputStream.writeUTF( str: "TEXT");
        dataOutputStream.writeUTF(msg);
        dataOutputStream.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

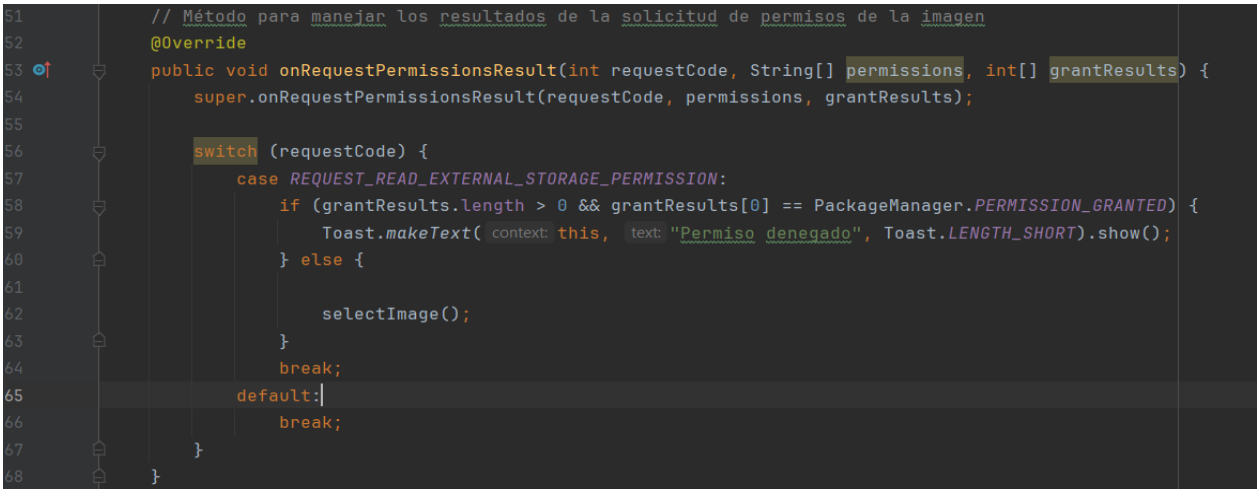
Este **método addMessageToChat** agrega un mensaje de texto al chat, ajustando su estilo según si el mensaje fue enviado por el usuario actual o por otro usuario. Los mensajes del usuario actual tienen un fondo personalizado, mientras que los mensajes de otros usuarios tienen un fondo gris claro. Luego, el mensaje se agrega al contenedor de la interfaz para que sea visible en el chat.

```java
private void addMessageToChat(String message) {  1 usage
    LinearLayout chatContainer = findViewById(R.id.chatImagesContainer);
    TextView chatText = new TextView( context: MainActivity.this);
    chatText.setText(message);
    chatText.setPadding( left: 10,  top: 10,  right: 10,  bottom: 10);
    chatText.setTextColor(Color.BLACK);

    // Estilo dinámico según el remitente
    if (message.startsWith(editTextUserName.getText().toString() + ":")) {
        // Mensaje enviado por el usuario actual
        setTextBackgroundColor(chatText);
    } else {
        // Mensaje de otro usuario (puedes asignar otro color o estilo aquí)
        GradientDrawable drawable = new GradientDrawable();
        drawable.setColor(Color.LTGRAY); // Color diferente para mensajes de otros usuarios
        drawable.setCornerRadius(20);
        drawable.setStroke( width: 2, Color.BLACK);
        chatText.setBackground(drawable);
    }

    LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(
            LinearLayout.LayoutParams.WRAP_CONTENT,
            LinearLayout.LayoutParams.WRAP_CONTENT
    );
    params.setMargins( left: 10,  top: 10,  right: 10,  bottom: 10); // Márgenes entre mensajes
    chatText.setLayoutParams(params);

    chatContainer.addView(chatText); // Agregar el mensaje al contenedor
}
```

Este **Metodo disconnect()** envía un mensaje de desconexión al servidor, cierra la conexión del socket y marca al cliente como desconectado. Si ocurre algún error en el proceso, lo maneja imprimiendo la excepción.

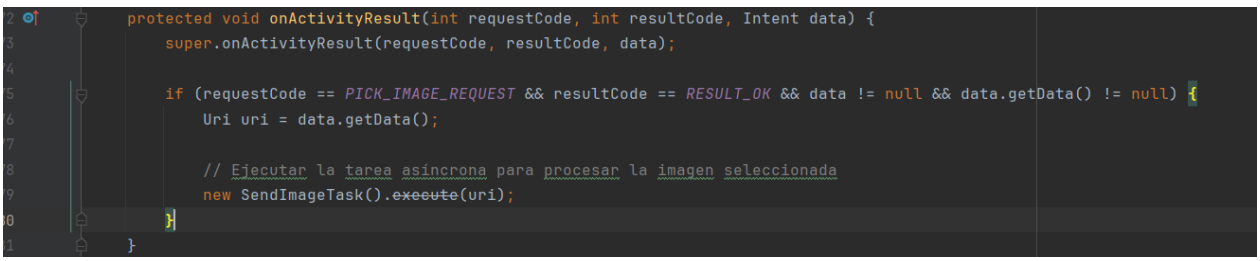
```
// Método para desconectar el cliente y enviar un mensaje de desconexión al servidor
private void disconnect() {  1 usage
    try {
        if (socket != null && !socket.isClosed()) {
            dataOutputStream.writeUTF( str: "DISCONNECT");  // Enviar la señal de desconexión
            dataOutputStream.flush();
            goOut = true;
            socket.close();  // Cerrar el socket
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

El **metodo onRequestPermissionsResult:** Este método maneja la respuesta a una solicitud de permisos para acceder al almacenamiento externo. Si el permiso es concedido, muestra un mensaje diciendo que fue denegado. Si el permiso es denegado, permite al usuario seleccionar una imagen llamando al método selectImage()



```
// Método para manejar los resultados de la solicitud de permisos de la imagen
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions, int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    switch (requestCode) {
        case REQUEST_READ_EXTERNAL_STORAGE_PERMISSION:
            if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                Toast.makeText( context: this,  text: "Permiso denegado", Toast.LENGTH_SHORT).show();
            } else {

                selectImage();
            }
            break;
        default:
            break;
    }
}
```

Este **método onActivityResult()** maneja el resultado de la selección de una imagen por parte del usuario. Si la imagen es seleccionada correctamente, obtiene su URI y ejecuta una tarea asíncrona (SendImageTask) para procesarla.



```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == PICK_IMAGE_REQUEST && resultCode == RESULT_OK && data != null && data.getData() != null) {
        Uri uri = data.getData();

        // Ejecutar la tarea asíncrona para procesar la imagen seleccionada
        new SendImageTask().execute(uri);
    }
}
```

El **metodo void run()**:Este método maneja la comunicación entre un cliente y un servidor en un hilo en segundo plano. Recibe mensajes de texto e imágenes, los procesa y actualiza la interfaz de usuario, y también envía mensajes de texto al servidor. Además, gestiona errores de conexión y cierra los recursos de manera adecuada al finalizar la comunicación.

```
//metodo que establece una conexion con una dirrecion ip y un puerto especifico
@Override
public void run() {
    try {
        socket = new Socket(dstAddress, dstPort);

        // Objetos para enviar y recibir datos a través del socket
        dataOutputStream = new DataOutputStream(socket.getOutputStream());
        dataInputStream = new DataInputStream(socket.getInputStream());

        // Enviar el nombre del usuario al servidor
        dataOutputStream.writeUTF(name);
        dataOutputStream.flush();

        while (!goOut) {
            // Verificar si hay datos disponibles
            if (dataInputStream.available() > 0) {
                String msgType = dataInputStream.readUTF(); // Leer el tipo de mensaje (texto o imagen)

                if (msgType.equals("TEXT")) {
                    String msg = dataInputStream.readUTF();
                    // Actualizar la interfaz de usuario con el mensaje recibido
                    MainActivity.this.runOnUiThread(() -> addMessageToChat(msg));
                    msgLog += msg + "\n"; // Actualizamos el log de mensajes
```

Para más información detallada se puede mirar el código está bastante documentado.

El alumno puede proponer otras mejoras al profesor: Se podría enviar documentos y la ubicación ó audios.

Las Imágenes en tiempo reales Cliente y Servidor: estarán comprimidas en la carpeta Abdiel Contreras