Práctica 3

Integrantes:

- **Flores Morales Aldahir Andrés**
- **Velasco Jiménez Luis Antonio**

Fecha de entrega: 01/12/2024

Materia: Aplicaciones para comunicaciones en red

Grupo: 6CM1

Introducción

Hoy día, en muchos tipos de aplicaciones se cuenta con un servicio de chat, ya sea con fines de diversión o de negocio. Este tipo de servicio es muy socorrido cuando se trata de brindar una comunicación más personalizada que la brindada por medios tales como correo electrónico, o foros. En el chat la comunicación puede ser fluida, es decir, en tiempo real y además pueden interactuar dos o más personas a la vez.

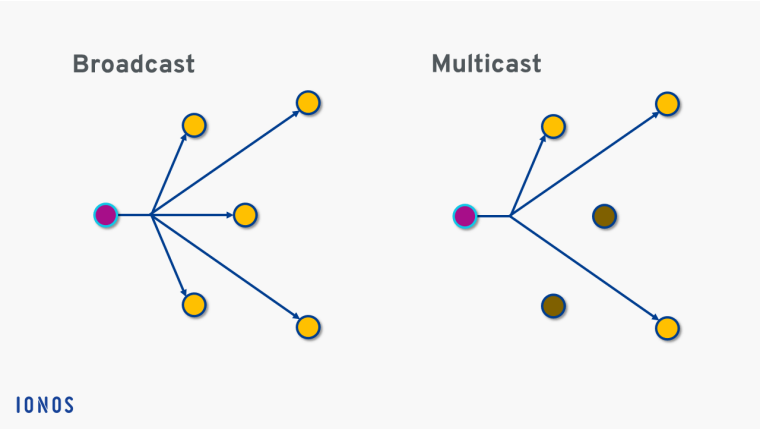
A continuación, se revisan algunos de los conceptos importantes que serán de gran utilidad para el desarrollo de la práctica.

Multicast

Multicast: Las comunicaciones multicast permiten el envío de datos desde un emisor a muchos receptores (uno-a-muchos), o desde muchos emisores a muchos receptores (muchos-a-muchos) si la gestión de los grupos se realiza de forma adecuada. Los envíos a muchos receptores se realizan de forma simultánea, de tal manera que se puede reducir el número de canales de comunicación.

En la actualidad los conmutadores que conectan los nodos de una red tienen soporte para administrar los grupos multicast. Estos grupos multicast pueden crecer o disminuir dinámicamente. Los nodos se unen (join) a un grupo multicast si están interesados en recibir tráfico dirigido a la dirección multicast de dicho grupo y lo deja (leave) cuando dejan de estar interesados.

El Internet Group Management Protocol (IGMP) permite llevar a cabo la comunicación entre los nodos y los conmutadores de la red.

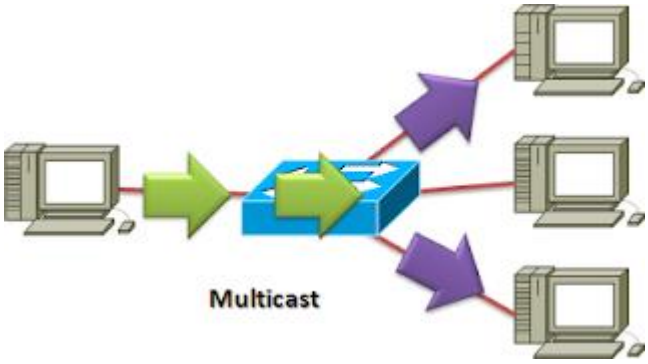


Multicast es un método de transmisión de datos que permite enviar información a varios usuarios de una red al mismo tiempo. En este modelo, el emisor selecciona previamente a los destinatarios, a diferencia de la difusión amplia o broadcast, donde todos los dispositivos de la red reciben el mismo dato.

Multicast es un modelo de transmisión de datos sobre redes IP que se adapta a diferentes modelos de distribución de contenidos, como uno-a-muchos o muchos-a-muchos.

Algunas de las ventajas de multicast son: Escalabilidad, Rendimiento, Menor gasto de capital.

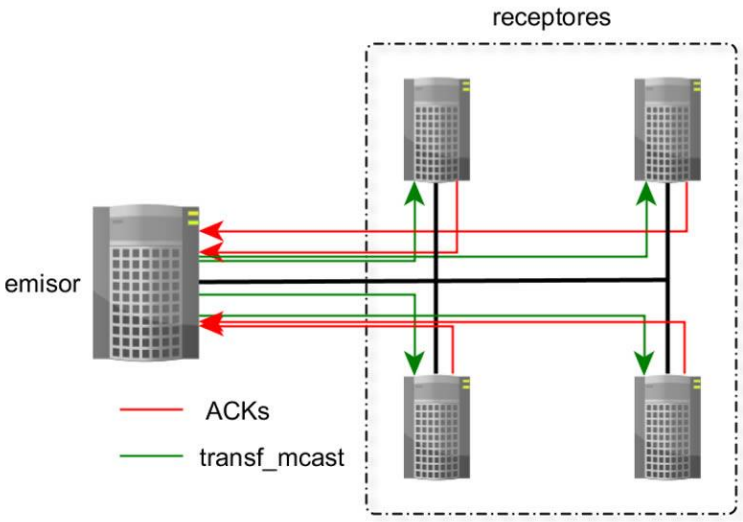
Un ejemplo de uso de multicast es la videoconferencia en vivo, donde una fuente multicast envía tráfico a un grupo multicast que incluye computadoras, dispositivos y teléfonos IP.



Sockets de multidifusión

A veces nos interesa que un ordenador pueda enviar un mensaje por red y que este sea recibido por otros ordenadores simultáneamente. Para ello están las direcciones multicast. Son direcciones en el rango 224.0.0.0 a 239.255.255.255. La 224.0.0.0 está reservada y no puede usarse.

Enviando mensajes por estas direcciones, cualquier otro ordenador en la red que las escuche podría leer dicho mensaje, independientemente de cuál sea la IP real de ese ordenador. Es decir, si un ordenador quiere enviar un mensaje simultáneamente a varios, puede hacerlo enviando el mensaje a una de estas IPs, los demás ordenadores deben estar a la escucha de dichas IPs para recibir el mensaje.



Los sockets de multidifusión son un tipo de socket que se utilizan para distribuir datos a múltiples destinatarios de manera eficiente.

La multidifusión es un método de redes IP que permite enviar datos a varios usuarios de una red desde un único punto. En este método, los paquetes se envían a una dirección de multidifusión designada, y luego se distribuyen a todos los miembros suscritos a esa dirección.

Los sockets son canales de comunicación que permiten que procesos no relacionados intercambien datos localmente y entre redes. Un socket tiene un tipo, se asocia a un proceso en ejecución y puede tener un nombre.

En Java, la clase MulticastSocket proporciona una interfaz de datagramas para multicast IP. Esta clase permite pertenecer a grupos multicast y tiene dos constructores alternativos:

- MulticastSocket(), que crea el socket en cualquiera de los puertos locales libres.
- MulticastSocket(int port), que crea el socket en el puerto local indicado.

Desarrollo

En esta práctica debes implementar una aplicación chat que permita a los usuarios comunicarse entre sí a través de una sala común, así como mensajes privados haciendo uso de sockets de multidifusión. La aplicación también permitirá a los usuarios enviar emoticones, imágenes y archivos. El formato de los mensajes a utilizar por parte de la aplicación será el siguiente:

La lógica del programa estará implementada en el lado del cliente.

Tal y como lo solicita, dentro de la clase Cliente.java se tiene implementado la mayoría de las operaciones y los llamados a los distintos métodos para poder iniciar y cerrar el servidor y la unión de los clientes. El programa Principal deberá unirse a la dirección de grupo 230.1.1.1 y usar el puerto 4000 para leer todos los mensajes enviados por los usuarios de la sala común.

Para este punto , la función main o principal se encuentra en la clase Cliente. Cuando esta se ejecuta se hace el llamado a la clase Servidor y se inicia el grupo en la dirección 230.1.1.1 y el puerto 4000.

En la siguiente imagen se observa la clase Servidor y su método constructor.

```
private Servidor()
{
    try
    {
        cl = new MulticastSocket(puerto);
        System.out.println("Cliente conectado desde: " + cl.getLocalPort());
        cl.setReuseAddress(true);
        cl.setTimeToLive(1);
        grupo = InetAddress.getByName(ip);
        cl.joinGroup(grupo);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Ilustración 1. Método Servidor donde se inicializa el servidor multicast.

En la Ilustración 2 se puede observar la línea donde se inicializa el servidor en el método iniciarChat que se utiliza al ejecutar el programa.

```
private void iniciarChat() throws IOException
{
    Inicio login = new Inicio(new javax.swing.JFrame(), true);
    login.setVisible(true);
    nombre = login.getNombre();

    if(nombre == null)
    {
        System.exit(0);
    }
    else
    {
        nombre.trim();
        Servidor.getInstance().iniciarSesion(nombre);
    }
}
```

Ilustración 2. Llamada al método iniciarSesión.

Cuando un usuario ejecute la aplicación, éste deberá proporcionar un nombre de usuario, entonces se enviará un mensaje a la dirección de grupo 230.1.1.1 al puerto destino 4000 con el mensaje "<inicio>[nombre-usuario]". Todos los clientes del chat deberán actualizar su lista de usuarios en línea al recibir el mensaje.

En este punto se utiliza el método iniciarSesion que se encuentra en la clase Servidor, que es llamada por el método main cuando se inicia un nuevo cliente y este se une a la sala del chat (grupo multicast). A su vez este método envía el mensaje "<inicio>" concatenado con el nombre del usuario al servidor. En la Ilustración 3 se muestra el código de lo anterior descrito.

```
public void iniciarSesion(String nombreOrigen) throws IOException
{
    String msj = "<inicio>" + nombreOrigen;
    b = msj.getBytes();
    packet = new DatagramPacket(b, b.length, grupo, puerto);
    cl.send(packet);
}
```

Ilustración 3. Método iniciar sesión.

Una vez enviado el mensaje de inicio, la aplicación cliente deberá iniciar la lectura del socket para recibir posibles mensajes de otros clientes.

En la Ilustración 4 se muestra el método actualizar el cual se encarga de mantener la comunicación para recibir cualquier mensaje privado o público (grupo), para ello utiliza un switch case que permite saber cuándo un cliente desea enviar un mensaje privado o a la sala del grupo.

```
private void actualizar() throws IOException
{
    Mensaje mensaje = Servidor.getInstance().recibe();
    mostrarMensaje(mensaje);

    if (mensaje.getNombreOrigen() != null)
    {
        switch (mensaje.getId())
        {
            case U_CONECTADO_ID:
                if (!mensaje.getNombreOrigen().equals(nombre))
                {
                    iniciarSesion(mensaje.getNombreOrigen());
                }
                break;

            case U_DESCONECTADO_ID:
                finSesion(mensaje.getNombreOrigen());
                break;

            case MENSAJE_PRIVADO_ID:
                if (!mensaje.getNombreOrigen().equals(nombre))
                {
                    if (mensaje.getNombreDestino().equals(nombre))
                    {
                        visualizarMensajePrivado(mensaje);
                    }
                }
                break;

            case MENSAJE_PUBLICO_ID:
                if (mensaje.getNombreOrigen().equals(nombre)) {
                    mensaje.setNombreOrigen("Tú");
                }
                visualizarMensajePublico(mensaje);
                break;
```

Ilustración 4. Método actualizar.

Cuando llegue un mensaje, este deberá ser interpretado con base al tipo de mensaje y en su caso, deberán realizarse las acciones correspondientes.

Relacionado con el punto anterior, cuando un usuario envía un mensaje público o privado este es interpretado con las etiquetas adecuadas, para ello se utilizan métodos que se

encuentran en la clase Servidor que a su vez son llamados por la clase Cliente. Estos métodos se observan en la Ilustración 5 que se muestra a continuación.

```
public void mensajeASala(String mensaje, String nombreOrigen) throws IOException
{
    String msj = "<msj>" + "<" + nombreOrigen + ">" + " " + mensaje + " ";
    b = msj.getBytes();
    packet = new DatagramPacket(b, b.length, grupo, puerto);
    cl.send(packet);
}

public void mensajePrivado(String nombreOrigen, String nombreDestino, String mensaje) throws IOException
{
    String msj = "<privado>" + "<" + nombreOrigen + ">" + "<" + nombreDestino + ">" + " " + mensaje + " ";
    b = msj.getBytes();
    packet = new DatagramPacket(b, b.length, grupo, puerto);
    cl.send(packet);
}
```

Ilustración 5. Métodos para envío de mensajes.

En caso de que el mensaje recibido sea un mensaje privado, solamente la aplicación del usuario que tenga asociado el nombre de usuario del destinatario indicado en el mensaje será quien muestre dicho mensaje en pantalla. Las demás aplicaciones ignorarán esos mensajes.

Dentro de la clase Cliente se encuentra el método mostrado en la Ilustración 6, este se encarga de distinguir si se trata de un mensaje privado o público dependiendo de la elección del cliente a través de la interfaz gráfica. Este se encarga de realizar lo solicitado, ignorar los mensajes que sean privados en caso de que estos no vayan dirigidos a la usuario y en caso contrario el mensaje se muestra a toda la sala.

```
@Override
public void actionPerformed(ActionEvent ae)
{
    if(ae.getSource().equals(Enviar))
    {
        try
        {
            if(nombreDestino.equals(GRUPO))
            {

                Servidor.getInstance().mensajeASala(Texto.getText(), nombre);
            }
            else
            {
                Servidor.getInstance().mensajePrivado(nombre, nombreDestino, Texto.getText());
            }

            Texto.setText("");
        }
        catch (IOException ex)
        {
            Logger.getLogger(Cliente.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
```

Ilustración 6. Envío de mensajes.

Cuando el mensaje recibido contenga emoticones, éstos deberán ser interpretados mostrándose las imágenes correspondientes en pantalla.

Por último, en la Ilustración 7 y 8 se muestra el método formatoAMensaje que está contenida la clase Emoticono. Esta se encarga de asociar una serie de formatos o secuencia de caracteres a los 6 emoticonos utilizados. Para ello se utiliza un while que lee y analiza cualquier tipo de mensaje enviado por el grupo multicast.

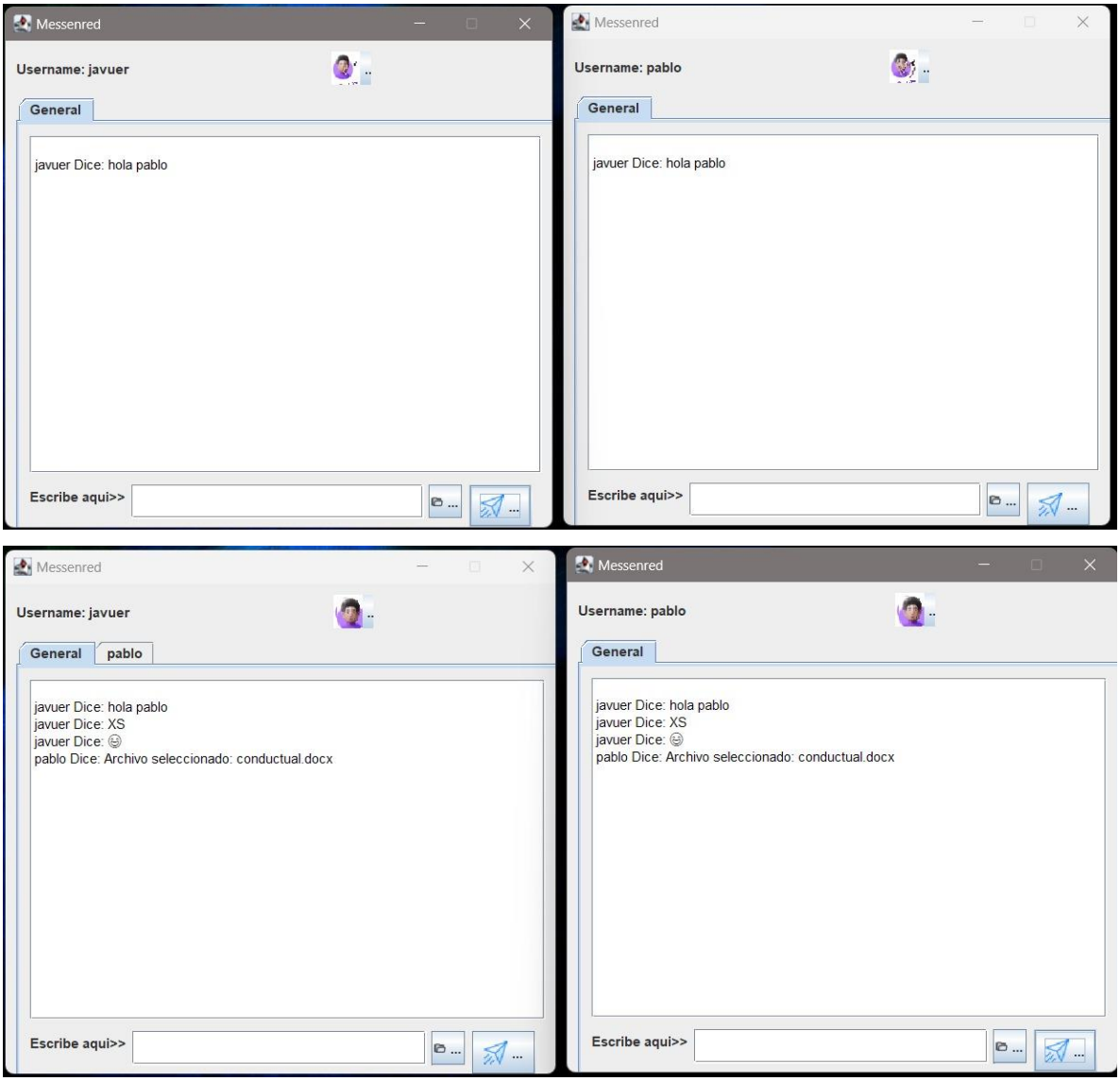
Para poder enviar los emojis usamos primitivas, en donde dependiendo de la letrea que se ingrese, se va a enviar el emoji predeterminado para esa letra. Para implementar esto utilizamos un switch para que fuera más eficiente.

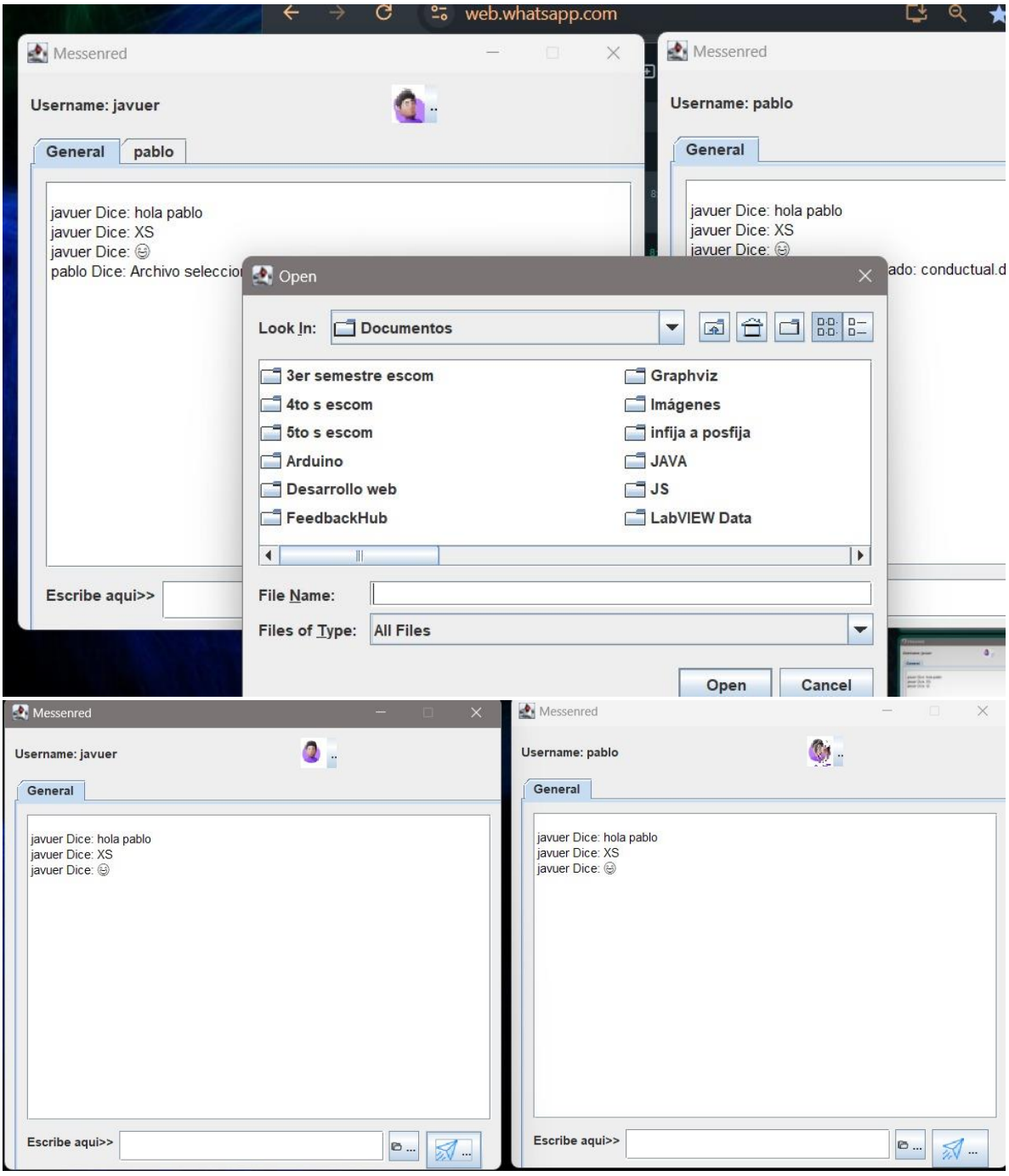
```
            }else{
                //break;
            }
        }
        contactos.add(e: contacto);
        System.out.println(x: contactos);
        //contactos.remove(Nombre);
    }else if(mensaje.startsWith(prefix: "S<msj>")){
        String remitente = "";
        mensaje = mensaje.substring(beginIndex: 6);

        mensaje = mensaje.replace(target: "XQ", replacement:"\uD83D\uDE04");
        mensaje = mensaje.replace(target: "XW", replacement:"\uD83D\uDE03");
        mensaje = mensaje.replace(target: "XE", replacement:"\uD83D\uDE0A");
        mensaje = mensaje.replace(target: "XR", replacement:"\uD83D\uDE1C");


        if(mensaje.contains(s: "<privado>")){
            String destinatario = "";
            int i = 1;

            mensaje = mensaje.substring(beginIndex: 9);
            while(Character.isLetter(ch: mensaje.charAt(index: i))){
                remitente = remitente + mensaje.charAt(index: i);
                i++;
            }
            mensaje = mensaje.substring(i + 1);

            i = 1;
            System.out.println(x: mensaje);
            while(Character.isLetter(ch: mensaje.charAt(index: i))){
```

Ilustración 7. Método para interpretar emoticonos (2).

Pruebas

Para poder crear una sala se necesitan conectar 2 usuarios o más. Una vez que se crea esto, el usuario puede mandar un mensaje a la sala o a otro cliente que esté conectado. Para enviar los emojis se tiene que usar una primitiva, dependiendo de lo que se quiera enviar. Una vez que se envía el mensaje se borrara cuando el cliente reciba otro, ya que no guarda los mensajes. En la parte derecha se encuentran todos los usuarios que se empiezan a conectar una vez que el servidor arranca.




web.whatsapp.com
Messenred
Username: javuer
General
pablo
javuer Dice: hola pablo
javuer Dice: XS
javuer Dice: 😉
pablo Dice: Archivo seleccio
Escribe aqui>>
Messenred
Username: pablo
General
javuer Dice: hola pablo
javuer Dice: XS
javuer Dice: 😉
ado: conductual.d
Open
Look In:
Documentos
3er semestre escom
4to s escom
5to s escom
Arduino
Desarrollo web
FeedbackHub
Graphviz
Imágenes
infija a posfija
JAVA
JS
LabVIEW Data
File Name:
Files of Type:
All Files
Open
Cancel
Messenred
Username: javuer
General
javuer Dice: hola pablo
javuer Dice: XS
javuer Dice: 😉
Escribe aqui>>
Messenred
Username: pablo
General
javuer Dice: hola pablo
javuer Dice: XS
javuer Dice: 😉
Escribe aqui>>

Conclusiones

Con esta práctica se pudo entender de mejor manera el funcionamiento de los sockets de datagrama, pero con un enfoque a la comunicación de multidifusión, además, esto nos muestra un ejemplo práctico de este tipo de sockets ya que parecen no tener utilidad.

También, con el desarrollo de esta práctica se pudo aprender a cómo utilizar y desarrollar una interfaz gráfica en Java.

Por último, esta aplicación nos permite visualizar y pensar en más usos de los sockets de datagrama, ya que, estos cuentan con distintas desventajas, pero a su vez con ventajas con respecto a los sockets TCP que nos permiten realizar acciones que los sockets TCP limitan.

Bibliografía

- https://www.uaeh.edu.mx/scige/boletin/huejutla/n9/r1.html
- https://www.ionos.mx/digitalguide/servidores/know-how/multicast/
- https://chuwiki.chuidiang.org/index.php?title=Socket_multicast_en_java