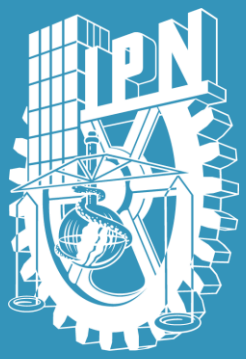

INSTITUTO POLITÉCNICO NACIONAL
ESCUELA SUPERIOR DE CÓMPUTO

ALUMNOS:

CASTILLO MAGAÑA OSCAR ISRAEL

CHÁVEZ BARRERA OSCAR HUITZILIN

HERNÁNDEZ MARTÍNEZ CARLOS DAVID

MALDONADO CARPIO JORGE ENRIQUE

APLICACIONES PARA COMUNICACIONES DE RED – 3CM8

PROFESOR: MORENO CERVANTES AXEL ERNESTO

PRÁCTICA 8 – P2P

INTRODUCCIÓN

Una estructura Peer to Peer (P2), a diferencia de la estructura cliente-servidor, cada Peer o nodo funciona tanto como cliente como servidor, permitiendo que así la comunicación se realice de manera descentralizada, teniendo como ventajas que la desconexión de un nodo no afecte de manera significativa la comunicación entre los demás nodos. Para una implementación sencilla de una estructura se utiliza un socket multicast, el cual, definiendo un grupo al cual unirse con sus iguales permite saber quiénes están en línea, y poder intercambiar información simple, sin mayores problemas entre todos los nodos. Una desventaja de estos sockets es la perdida de información y que, muchas veces se puede saturar la red de paquetes, por ello al momento de mandar mayor cantidad de información se utilizan otros tipos de sockets para permitir intercambiar con mayor fiabilidad, para esta práctica utilizaremos la API de RMI.

RMI (Java Remote Method Invocation) es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y proporciona un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java. Si se requiere comunicación entre otras tecnologías debe utilizarse CORBA o SOAP en lugar de RMI.

RMI se caracteriza por la facilidad de su uso en la programación por estar específicamente diseñado para Java; proporciona paso de objetos por referencia (no permitido por SOAP), recolección de basura distribuida (Garbage Collector distribuido) y paso de tipos arbitrarios (funcionalidad no provista por CORBA).

A través de RMI, un programa Java puede exportar un objeto, con lo que dicho objeto estará accesible a través de la red y el programa permanece a la espera de peticiones en un puerto TCP. A partir de ese momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto.

OBJETIVO

Desarrollar una aplicación con estructura P2P, la cual intercambie archivos mediante RMI y sockets multicast. Determinando la tecnología más óptima para realizar cada operación necesaria al momento de transferir los archivos.

DESARROLLO

La estructura de clases de nuestra aplicación para obtener el objetivo deseado es la siguiente:

- **Lemonwire:** Esta clase hereda de JFrame, será nuestra clase principal, además de nuestra interfaz gráfica. En esta clase mostraremos las opciones para iniciar el socket multicast con un nombre de usuario, así como mostrará una lista de los archivos disponibles en el grupo multicast y un botón para descargar el archivo seleccionado. Su única implementación

importante es el método que se origina al momento de oprimir un botón. La cual es la siguiente:

```
public void actionPerformed(ActionEvent aEvent) {
   JButton clicked = (JButton) aEvent.getSource();
   if (clicked == jbDownload) {
     server.askFile((String)jListFiles.getSelectedValue());
   } else {
     server = new Server(jtfUsername.getText());
     server.start();
     new Thread(this).start();
   }
  }
```

En este método, si el botón que recibe el evento es el de descargar, llamamos a una instancia de la clase Server, y utilizamos su método askFile(String fName) el cual se encargará de obtener el archivo deseado. En caso contrario, el botón seleccionado es el que instancia a la clase Server con el nombre de usuario especificado.

- **Server:** Esta clase es la más importante, dado que se encarga de bastantes tareas, desde actualizar la lista de archivos, recibir los mensajes del grupo multicast, iniciar servidores rmi así como instanciarlos de manera remota. Es la clase que realiza la funcionalidad real de la aplicación. Extiende de la clase Thread para poder ser ejecutada de manera concurrente con la GUI. La implementación más importante es la siguiente:

El constructor de la clase, el cual inicializa un socket multicast y lo une a un grupo, además de que guarda el nombre de usuario escogido para ser utilizado más adelante.

```
public Server(String username) {
   try {
     System.setProperty("java.net.preferIPv4Stack", "true");
     multiSocket = new MulticastSocket(port);
     multiSocket.joinGroup(InetAddress.getByName(address));
     data = new byte[1024];
     dPacket = new DatagramPacket(data, data.length);
     dataFolder = new File("./Data");
     myArrListFiles = new ArrayList<String>();
     arrListFiles = new ArrayList<String>();
     peers = new ArrayList<String>();
     this.username = username;
     askedFile = "";
   } catch(Exception e) {
     e.printStackTrace();
   }
```

```
  }
```

Al momento de correr nuestro hilo, utilizaremos el método run, el cual simplemente se encarga de crear un ciclo infinito, que cada segundo actualizará la lista de los archivos propios, la enviará y recibirá los mensajes que se reciban en el grupo. Su implementación es la siguiente:

```
public void run() {
   try {
    while (true) {
      Thread.sleep(1000);
      updateFiles();
      sendFilesList();
      arrListFiles.addAll(myArrListFiles);
      for (int i = 0; i < 15; i++) {
        receiveFileList();
      }
    }
   } catch(Exception e) {
    e.printStackTrace();
   }
 }
```

El siguiente método es el que se encarga de leer los mensajes recibidos del grupo, los procesa de tal manera que si encuentra un carácter < significa que el usuario desea descargar el archivo llamado así, y el programa verifica si se tiene este archivo y se manda una confirmación de si lo tiene realmente, si es así, además inicia un servidor RMI con el nombre de usuario, para que el otro programa sepa dónde encontrar los archivos. Si se recibe un carácter > significa que un usuario tiene el archivo solicitado lo tiene ese nombre de usuario, lo agrega a una lista de peers que tienen el archivo y lo solicita posteriormente. En caso contrario, verifica si no ha agregado anteriormente el archivo, si no lo ha hecho lo agrega a la lista de archivos.

```
public void receiveFileList() {
   try {
    boolean add = true;
    Arrays.fill(data, (byte)0);
    multiSocket.receive(dPacket);
    String received = new String(data);
    String cleaned = "";
    for (int i = 0; i < received.length(); i++) {
      if ((int)received.charAt(i) != 0)
        cleaned += received.charAt(i);
    }
    if (cleaned.charAt(0) == '<') {
      add = false;
```

```
        sendConfirmation(cleaned.substring(1));
      }
      if (cleaned.charAt(0) == '>') {
        add = false;
        if (askedFile.equals(cleaned.substring(cleaned.indexOf("<") + 1))) {
          peers.add(cleaned.substring(1, cleaned.indexOf("<")));
          System.out.println("Active peers: " + peers.size());
          for (String user : peers) {
            System.out.println("User: " + user);
          }
        }
      }
      if (!arrListFiles.contains(cleaned) && add) {
        arrListFiles.add(cleaned);
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Las demás implementaciones consisten en una creación de un servidor RMI, su instanciación y la obtención de un arreglo de bytes parcial del mismo, construyendo así el archivo.

CONCLUSIÓN

La estructura de programas Peer to Peer (P2P), resulta más fácil de implementar mediante sockets multicast, esto es porque esta clase de sockets permiten una comunicación en la cual no existen servidores como tal, sino, que los sockets se unen a un grupo para empezar la comunicación. Es por ello que en programas de esta índole resultan mucho más prácticos para buscar y solicitar información a todos los sockets posibles, pero al momento de comunicarse buscando información más compleja, resulta mucho más fácil utilizar sockets de flujo, los cuales aumentan la fiabilidad de la información, pero a costa de limitar la comunicación entre un cliente y un servidor únicamente. Es por todo esto, que se utilizó RMI al momento de saber que un Peer tenía algún archivo, asegurando una mayor fiabilidad en que llegarían los paquetes correctamente.