



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO



-----APLICACIONES EN COMUNICACIONES EN RED-----

PRÁCTICA 7:

Aplicación P2P

Alumnos:

- Hernández Rodríguez Armando Giovanni
- Meza Vargas Brandon David
- Muñoz Lozano Areli
- Quintero Maldonado Iván
- Shim Kyuseop

Grupo:

3CM16

Profesor:

Moreno Cervantes Axel Ernesto

Índice

| | |
|--------------------------------------|----|
| Introducción..... | 4 |
| Desarrollo | 5 |
| Servidor..... | 5 |
| <i>FoundFile.java</i> | 5 |
| <i>MD5Checksum.java</i> | 5 |
| <i>MulticastServer.java</i> | 5 |
| <i>RMIsearch.java</i> | 6 |
| <i>RMIserver.java</i> | 6 |
| <i>StartsServer.java</i> | 7 |
| <i>UnicastServer.java</i> | 7 |
| Cliente | 8 |
| <i>DataFromServer</i> | 8 |
| <i>Files</i> | 9 |
| <i>MulticastClient</i> | 10 |
| <i>MulticastServersWatcher</i> | 11 |
| <i>UnicastClient</i> | 12 |
| <i>RMIClient</i> | 14 |
| <i>App</i> | 15 |
| Pruebas de funcionamiento | 20 |
| Conclusiones..... | 23 |
| Bibliografía..... | 24 |

Índice de ilustraciones

| | |
|---|----|
| Ilustración 1. MulticastServer | 6 |
| Ilustración 2. RMIServer | 7 |
| Ilustración 3. UnicastServer | 8 |
| Ilustración 4. DataFromServer. | 9 |
| Ilustración 5. Clase Files | 10 |
| Ilustración 6. Obteniendo a grupo multicast | 10 |
| Ilustración 7. Método run del cliente Multicast..... | 11 |
| Ilustración 8. MulticastServersWatcher | 12 |
| Ilustración 9. UnicastClient. | 14 |
| Ilustración 10. RMIClient..... | 15 |
| Ilustración 11. Constructor..... | 16 |
| Ilustración 12. Iniciando clientes..... | 17 |
| Ilustración 13. initComponents | 19 |
| Ilustración 14. searchBtnEvent | 19 |
| Ilustración 15. Iniciando servidor. | 20 |
| Ilustración 16. Práctica P2P interfaz..... | 20 |
| Ilustración 17. Archivos a buscar..... | 21 |
| Ilustración 18. Información del archivo encontrado..... | 21 |
| Ilustración 19. Archivo descargado. | 22 |
| Ilustración 20. Archivo guardado en carpeta correspondiente. | 22 |

Introducción

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.

La invocación se compone de los siguientes pasos:

- Encapsulado (marshalling) de los parámetros (utilizando la funcionalidad de serialización de Java).
- Invocación del método (del cliente sobre el servidor). El invocador se queda esperando una respuesta.
- Al terminar la ejecución, el servidor serializa el valor de retorno (si lo hay) y lo envía al cliente.
- El código cliente recibe la respuesta y continúa como si la invocación hubiera sido local.

Desarrollo

Servidor

El proyecto Java “Servidor” cuenta con 7 clases:

1. FoundFile.java
2. MD5Checksum.java
3. MulticastServer.java
4. RMISearch.java
5. RMIServer.java
6. StartsServer.java
7. UnicastServer.java

Se procederá a dar una breve explicación de estas clases y las partes importantes de código para el funcionamiento de la aplicación.

FoundFile.java

Contiene los métodos `getFileName()`, `setFileName()`, `getPath`, `setPath()`, `getMd5()` y `setMd5()` para establecer el nombre y dirección así como el MD5 perteneciente a un archivo

MD5Checksum.java

Permite obtener el MD5 de un archivo, a continuación se explica el procedimiento. Primero se crea un flujo de entrada con el archivo seleccionado, de este se leen cierta cantidad de bytes y se guardan en un arreglo de tipo byte, se crea un `MessageDigest` que será el encargado de crear y actualizar el digesto correspondiente de acuerdo con la cantidad de bytes leídos. Luego, cuando se termina la lectura del archivo se cierra el flujo de entrada y se regresa el digesto correspondiente. Después, se convierte el digesto obtenido en formato hexadecimal a cadena de texto y se obtiene el resultado. Para ello se utilizan los métodos `createChecksum()` y `getMD5Checksum()`

MulticastServer.java

Este servidor básicamente se encargará de crear un hilo, el cual anunciará cada 5 segundos el puerto de servicio del servidor de flujo.

En la siguiente imagen se observa el código implementado para esta clase, se puede notar que también se dispone del método `send()` de tipo booleano que permite enviar el anuncio. Si el resultado obtenido del método `send()` es `true` el hilo permanecerá dormido 5 segundos, en caso contrario no lo hará.

```

public void run(){
    try {
        group = InetAddress.getByName(Constants.MULTICAST_ADDRESS);
        while (true){
            send("MulticastServer");
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    } catch (IOException e){
        e.printStackTrace();
        System.exit(2);
    }
}

public Boolean send(String message){
    try {
        MulticastSocket sendSocket = new MulticastSocket(Constants.MULTICAST_PORT);
        sendSocket.joinGroup(group);
        DatagramPacket packet = new DatagramPacket(message.getBytes(), message.length(),
        group, Constants.MULTICAST_PORT);
        sendSocket.send(packet);
        sendSocket.close();
        return true;
    } catch (IOException e){
        e.printStackTrace();
        return false;
    }
}

```

Ilustración 1. MulticastServer

RMISearch.java

Realiza una extensión “Remote” que nos ayuda a indicar que la interfaz se identifica a si misma como una interfaz cuyos métodos pueden ser invocados desde otra máquina virtual Java

RMIserver.java

En esta clase se implementa un servicio de búsqueda de archivos, el cual recibe como parámetro de entrada el nombre y dirección del archivo a ser buscado, y devuelve como salida el nombre y path del archivo encontrado, así como el MD5 del archivo encontrado. Sino se encuentra el archivo se retorna la cadena de texto “unknow”. El método encargado de realizar esta acción es searchFile() el cual devuelve un objeto de tipo FoundFile



Ilustración 2. RMIServer

StartsServer.java

Permite iniciar los servidores: UnicastServer, RMIServer y MulticastServer

UnicastServer.java

Este servidor únicamente se encargará de recibir peticiones de descarga y enviar los archivos, o fragmentos de archivos solicitados al cliente. El método sendFile()

El envío o descarga del archivo se almacena en el directorio de la aplicación cliente. El proceso de descarga es el siguiente, se hace una instancia de la clase DataOutputStream con ella se establece el flujo de salida para escribir los datos del archivo, los cuales se almacenan en un arreglo binario de tamaño 1500 bytes. Al usuario se le muestra el porcentaje y la cantidad de bytes leídos del archivo mediante una barra de progreso, cuando la cantidad de bytes recibidos es igual al tamaño se termina de escribir pues el archivo se descargó completamente. Finalmente se cierran los flujos de entrada, salida y el cliente. Lo cual se puede notar en la siguiente imagen.



Ilustración 3. UnicastServer

Cliente

A continuación se presenta la explicación de la parte del cliente, aquí hay 1 clase y una interfaz que se usan en el servidor y son las mismas, por esta razón no se explicarán.

DataFromServer

Esta clase se encarga de establecer la información que contendrá cada servidor, en este caso es la dirección, su temporizador y el puerto.


```

public class DataFromServer implements Serializable {
    private String address;
    private int temp;
    private int port;

    public DataFromServer(String address, int temp, int port) {
        this.address = address;
        this.temp = temp;
        this.port = port;
    }

    public DataFromServer(){}

    public String getAddress() { return address; }

    public void setAddress(String address) {
        this.address = address;
    }

    public int getTemp() { return temp; }

    public void setTemp(int temp) {
        this.temp = temp;
    }

    public int getPort() { return port; }

    public void setPort(int port) { this.port = port; }
}

```

Ilustración 4. DataFromServer.

Files

Esta clase nos va a servir como una base de datos dentro de nuestro programa, pues contendrá el archivo que está en el servidor, el archivo que ha sido encontrado en el servidor y la lista de servidores que tenemos disponibles, además de contar con un método que nos sirve para agregar servidores.

```

public class Files {
    private String fileInServer = new String();
    List<DataFromServer> servers = new ArrayList<>();
    private FoundFile foundFile = new FoundFile();

    public String getFileInServer() { return fileInServer; }

    public void setFileInServer(String fileInServer) { this.fileInServer = fileInServer; }

    public List<DataFromServer> getServers() { return servers; }

    public void setServers(List<DataFromServer> servers) { this.servers = servers; }

    public FoundFile getFoundFile() { return foundFile; }

    public void setFoundFile(FoundFile foundFile) { this.foundFile = foundFile; }
    public void addServer(DataFromServer dfs) { servers.add(dfs); }
}

```

Ilustración 5. Clase Files

MulticastClient

Primeramente hacemos la conexión al servidor multicast obtenemos el grupo multicast, esto lo hacemos directamente en el constructor como se ve a continuación.

```

public MulticastClient(Files db){
    this.db = db;
    try {
        group = InetAddress.getByName(Constants.MULTICAST_ADDRESS);
    } catch (UnknownHostException e){
        e.printStackTrace();
        System.exit( status: 1);
    }
}

```

Ilustración 6. Obteniendo a grupo multicast

En el método run nos unimos al grupo multicast y nos mantenemos a la escucha de todos los datagramas que lleguen, a partir de estos vamos obteniendo la información que será IP del anunciante y los datos donde estará el puerto del servidor de flujo del servidor anunciante. Cada que se reinicia el temporizador vamos agregando ese servidor a nuestra lista de servidores.

```

public void run(){
    try {
        MulticastSocket socket = new MulticastSocket(Constants.MULTICAST_PORT);
        socket.joinGroup(group);
        String msg = "";
        for(;;){
            byte []buffer = new byte[Constants.BUFFER_SIZE];
            DatagramPacket recv = new DatagramPacket(buffer, buffer.length);
            socket.receive(recv);
            byte []data = recv.getData();
            msg = new String(data);
            msg = msg.trim();

            /**Verifying if server is already saved*/
            DataFromServer curServer = new DataFromServer(recv.getAddress().toString().substring(1), temp: 6, recv.getPort());
            /**Adding server*/
            servers = db.getServers();
            int pos = isInList(servers, curServer);
            if(pos == -1){
                db.addServer(curServer);
            }else{
                servers.get(pos).setTemp(6);
                db.setServers(servers);
            }
        }
    }catch (IOException e){
        e.printStackTrace();
        System.exit( status: 2);
    }
}

```

Ilustración 7. Método run del cliente Multicast

MulticastServersWatcher

En esta clase vamos a estar actualizando la lista de servidores cada segundo, si un servidor no se reporta, esto es cuando el temporizador llega a 0, se elimina de la lista de servidores, en caso de un datagrama sea anunciado la dirección de una entrada ya existente en la lista, el contador será reiniciado.

```

public MulticastServersWatcher(Files db) { this.db = db; }

/**
 * We check for the timer initialized in 6 seconds waiting for a datagram
 * if a datagram is not captures we delete the server from the list
 */
public void run(){
    int i = 0;
    for(;;){
        try {
            List<DataFromServer> servers = db.getServers();
            if(servers.size() != 0){

                for (i = 0; i < servers.size(); i++){
                    /**Removing server if is not reported*/
                    if(servers.get(i).getTemp() == 0)
                        servers.remove(i);
                    else
                        servers.get(i).setTemp(servers.get(i).getTemp()-1);
                }
            }
            Thread.sleep( millis: 1000);
        }catch (InterruptedException e){
            Thread.currentThread().interrupt();
        }
    }
}

```

Ilustración 8. MulticastServersWatcher

UnicastClient

En este cliente lo que hacemos es básicamente un cliente de flujo que se encargará de conectarse al servidor de flujo y recibirá los archivos que el servidor le mando, en otras palabras se encargará de descargarlos.

```

public class UnicastClient extends Thread{
    private App app;
    private Socket cl;
    private final Files db;

    public UnicastClient(Files db, App app){
        this.app = app;
        this.db = db;
    }

    /**
     * Method that connect to the local server
     */
    public void serverConnection(){
        try {
            cl = new Socket(db.getFileInServer(), Constants.UNICAST_PORT);
            System.out.println(Constants.CLIENT_CONNECTED);
        } catch (Exception e){
            e.printStackTrace();
        }
    }

    /**
     * Method that download the specified file
     */

```

```

public void receiveFile(){
    if(cl != null){
        try {
            DataOutputStream dos = new DataOutputStream(cl.getOutputStream());
            /**Sending file we wanna download
            dos.writeUTF(db.getFoundFile().getPath());
            dos.flush();
            /**Waiting for the response*/
            DataInputStream dis = new DataInputStream(cl.getInputStream());
            /**Receiving file*/
            String name = dis.readUTF();
            long tam = dis.readLong();
            DataOutputStream dosFile = new DataOutputStream(new FileOutputStream(String.format(Constants.FILES_PATH,name)));
            byte []b = new byte[Constants.FILE_BUFFER_SIZE];
            long received = 0;
            int percentage = 0;
            int n = 0;
            while(received < tam){
                n = dis.read(b);
                dosFile.write(b, off: 0, n);
                dosFile.flush();
                received += n;
                percentage = (int)((received*100) / tam);
                app.setProgressBar(percentage);
            }
            dos.close();
            dosFile.close();
            dis.close();
            cl.close();
            showMessageDialog( parentComponent: null, Constants.FILE_RECEIVED_SUCCESSFULLY);
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}

```

Ilustración 9. UnicastClient.

RMIClient

En el cliente RMI lo que hacemos es hacer la búsqueda del archivo solicitado, obtenemos el registro creado en el servidor a partir del puerto usado en el servidor RMI, y buscamos el stub llamado Search, este nos entregará el archivo que se esta pidiendo.

```

public void searchForFile(String file){
    int i = 0;
    try {
        List<DataFromServer> servers = db.getServers();
        if(servers.size() != 0){
            for(i = 0; i < servers.size(); i++){
                Registry registry = LocateRegistry.getRegistry(servers.get(i).getAddress(), Constants.RMI_PORT);
                RMISearch stub = (RMISearch) registry.lookup( name: "Search");
                FoundFile response = stub.search(file);
                if(!response.getFileName().equals("unknown")){
                    app.fileFound(response.getMd5(), response.getPath(), response.getFileName(), servers.get(i).getAddress(), isVisible: true);
                    db.setFoundFile(response);
                    db.setFileInServer(servers.get(i).getAddress());
                    //line to make download button appears
                    app.setDownloadVisibility(true);
                }else{
                    //GUI LINES
                    app.fileFound( md5: "", path: "", filename: "", server: "", isVisible: false);
                    app.setDownloadVisibility(false);
                }
            }
        }
    } catch (Exception e){
        e.printStackTrace();
    }
}

```

Ilustración 10. RMIClient

App

En esta parte tenemos la interfaz gráfica del usuario, básicamente consta de un botón para buscar el archivo solicitado y una vez encontrado nos muestra su información y se habilita un botón para descargar el archivo y una barra de progreso de la descarga se hace visible. A continuación se muestra el código de esta clase.

```

public class App extends JFrame {
    private JPanel panel;
    private JTextField searchFile;
    private JButton searchBtn;
    private JButton downloadBtn;
    private JProgressBar downloadProgress;
    private JFrame window;
    private JLabel fileName;
    private JLabel filePath;
    private JLabel fileMD5;
    private JLabel fileServer;
    private JLabel fileNameText;
    private JLabel appTitle;

    Files db = new Files();
    UnicastClient uClient = new UnicastClient(db, app: this);
    MulticastClient mClient = new MulticastClient(db);
    MulticastServersWatcher msw = new MulticastServersWatcher(db);
    RMIClient rmiClient = new RMIClient(db, app: this);

    public App(){
        Toolkit myScreen = Toolkit.getDefaultToolkit();
        Dimension screenSize = myScreen.getScreenSize();

        window = new JFrame();

        window.setSize((screenSize.width / 4), height: screenSize.height / 2);
        window.setLocation( x: screenSize.width / 4, y: screenSize.height / 4);
        System.out.println(screenSize.width / 2 + "altuura: " + screenSize.height / 2);
        window.setResizable(true);
        window.setTitle(String.format(Constants.WINDOW_TITLE));
        window.setLocationRelativeTo(null);
        window.setDefaultCloseOperation(EXIT_ON_CLOSE);
        initComponents();

        fileName.setVisible(false);
        filePath.setVisible(false);
        fileMD5.setVisible(false);
        fileServer.setVisible(false);
        downloadProgress.setVisible(false);
        downloadBtn.setVisible(false);
        window.setVisible(true);
        clientsStart();
    }
}

```

Ilustración 11. Constructor

En el método `clientsStart` empezamos la ejecución de los clientes.

```
public void clientsStart(){  
    mClient.start();  
    msw.start();  
    rmiClient.start();  
    uClient.start();  
}
```

Ilustración 12. Iniciando clientes

En la parte de `initComponents` lo único que hacemos es acomodar los elementos de la interfaz gráfica de una manera agradable al usuario.

```

public void initComponents(){
    panel = new JPanel();
    panel.setLayout(null);
    window.getContentPane().add(panel);
    panel.setBackground(new Color( r: 255, g: 246, b: 234));

    fileNameText = new JLabel(Constants.FILE_NAME_TEXT);
    fileNameText.setBounds( x: 10, y: 80, width: 150, height: 40);
    panel.add(fileNameText);

    appTitle = new JLabel(Constants.WINDOW_TITLE);
    appTitle.setBounds( x: 150, y: 10, width: 200, height: 40);
    appTitle.setFont(new FontUIResource( name: "Roboto", Font.BOLD, size: 30));
    panel.add(appTitle);

    searchFile = new JTextField();
    searchFile.setBounds( x: 130, y: 85, width: 200, height: 30);
    panel.add(searchFile);

    searchBtn = new JButton( text: "Buscar");
    searchBtn.setBackground(Color.WHITE);
    searchBtn.setBounds( x: 333, y: 85, width: 125, height: 30);
    searchBtn.setFont(new FontUIResource( name: "Roboto", Font.BOLD, size: 15));
    panel.add(searchBtn);

    fileServer = new JLabel(Constants.FILE_SERVER_TEXT);
    fileServer.setBounds( x: 10, y: 150, width: 200, height: 40);
    fileServer.setFont(new FontUIResource( name: "Roboto", Font.BOLD, size: 15));
    panel.add(fileServer);

    fileName = new JLabel(Constants.FILE_NAME_TEXT);
    fileName.setBounds( x: 10, y: 200, width: 200, height: 40);
    fileName.setFont(new FontUIResource( name: "Roboto", Font.BOLD, size: 15));
    panel.add(fileName);
}

```

```

filePath = new JLabel(Constants.FILE_PATH_TEXT);
filePath.setBounds(x: 10, y: 250, width: 200, height: 40);
filePath.setFont(new FontUIResource(name: "Roboto", Font.BOLD, size: 15));
panel.add(filePath);

fileMD5 = new JLabel(Constants.FILE_MD5_TEXT);
fileMD5.setBounds(x: 10, y: 300, width: 200, height: 40);
fileMD5.setFont(new FontUIResource(name: "Roboto", Font.BOLD, size: 15));
panel.add(fileMD5);

downloadBtn = new JButton(Constants.DOWNLOAD_BTN_TEXT);
downloadBtn.setBackground(Color.WHITE);
downloadBtn.setBounds(x: 10, y: 400, width: 200, height: 40);
downloadBtn.setFont(new FontUIResource(name: "Roboto", Font.BOLD, size: 15));
panel.add(downloadBtn);

downloadProgress = new JProgressBar();
downloadProgress.setBounds(x: 220, y: 410, width: 220, height: 20);
panel.add(downloadProgress);

/**Btns event listeners*/
searchBtn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) { searchBtnEvent(); }
});

downloadBtn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) { downloadBtnEvent(); }
});

```

Ilustración 13. initComponents

Al presionar el botón de buscar se ejecuta el siguiente método el cual manda a llamar a la búsqueda por RMI.

```

public void searchBtnEvent(){
    //System.out.println("preseed btn");
    List<DataFromServer> servers = db.getServers();
    if(!servers.isEmpty())
        rmiClient.searchForFile(searchFile.getText());
    else
        JOptionPane.showMessageDialog(parentComponent: null, Constants.EMPTY_SERVERS_MESSAGE, title: "Busqueda", JOptionPane.WARNING_MESSAGE);
}

```

Ilustración 14. searchBtnEvent

El botón de descargar hace la conexión al servidor de flujo para recibir el archivo, es decir, descargarlo.

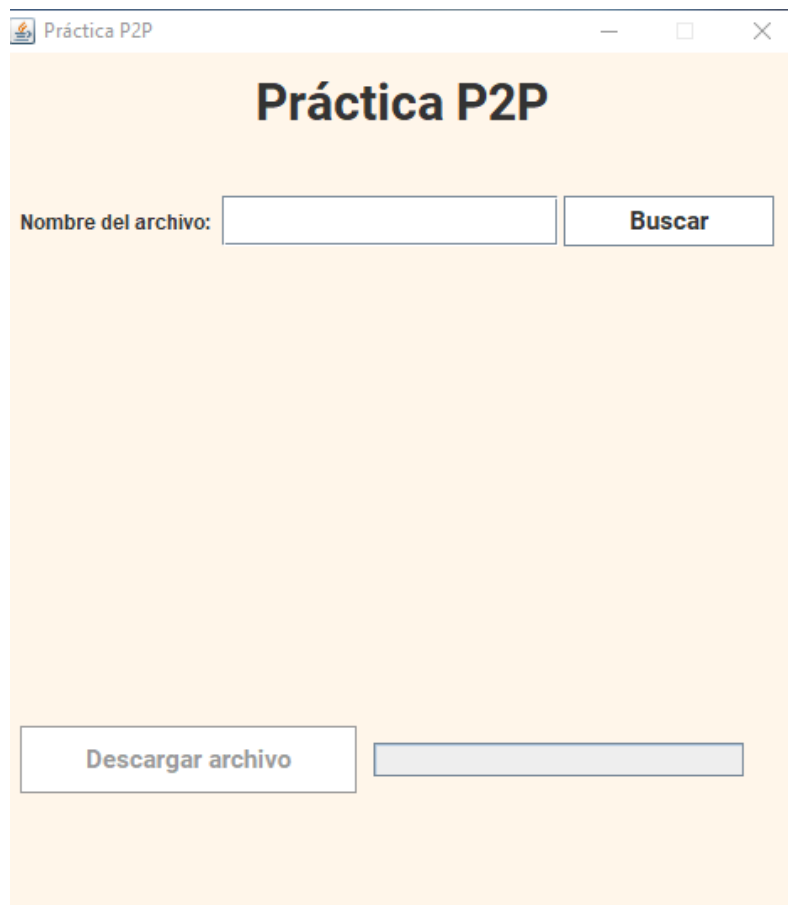
Pruebas de funcionamiento

Para comenzar con el programa tenemos que correr la clase startServers de parte del servidor, este nos mostrará el siguiente mensaje.

```
run:  
Servidor iniciado
```

Ilustración 15. Iniciando servidor.

Una vez corrido nuestro servidor podemos correr el cliente con la clase App del cliente, al correrla tenemos la siguiente interfaz.



Práctica P2P

Nombre del archivo:

Ilustración 16. Práctica P2P interfaz

Para buscar archivos tenemos que buscar alguno que se encuentre en la carpeta establecida, la cual contiene los siguientes archivos.

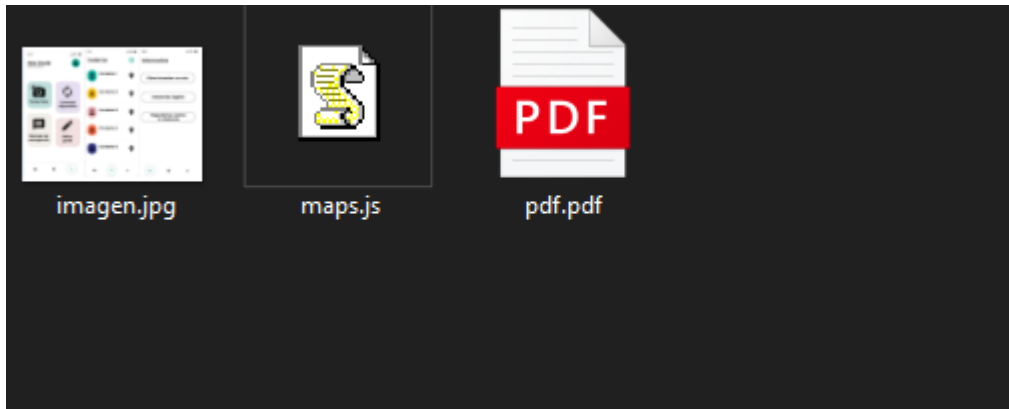


Ilustración 17. Archivos a buscar.

Al momento de hacer la búsqueda de un archivo se nos muestra la siguiente información.

Práctica P2P

Práctica P2P

Nombre del archivo:

Server del archivo: 192.168.56.1

Nombre del archivo: maps.js

Ruta del archivo: D:\PC\Práctica7\Práctica7\Servidor\files\maps.js

MD5 del archivo: b8f3a48dd8708a5a1f6c0ce17cdf7a66

Ilustración 18. Información del archivo encontrado.

Una vez encontrado el archivo se nos habilita el botón para poder descargarlo y se guardará en la carpeta destinada para las descargas como se muestra en las siguientes dos imágenes.

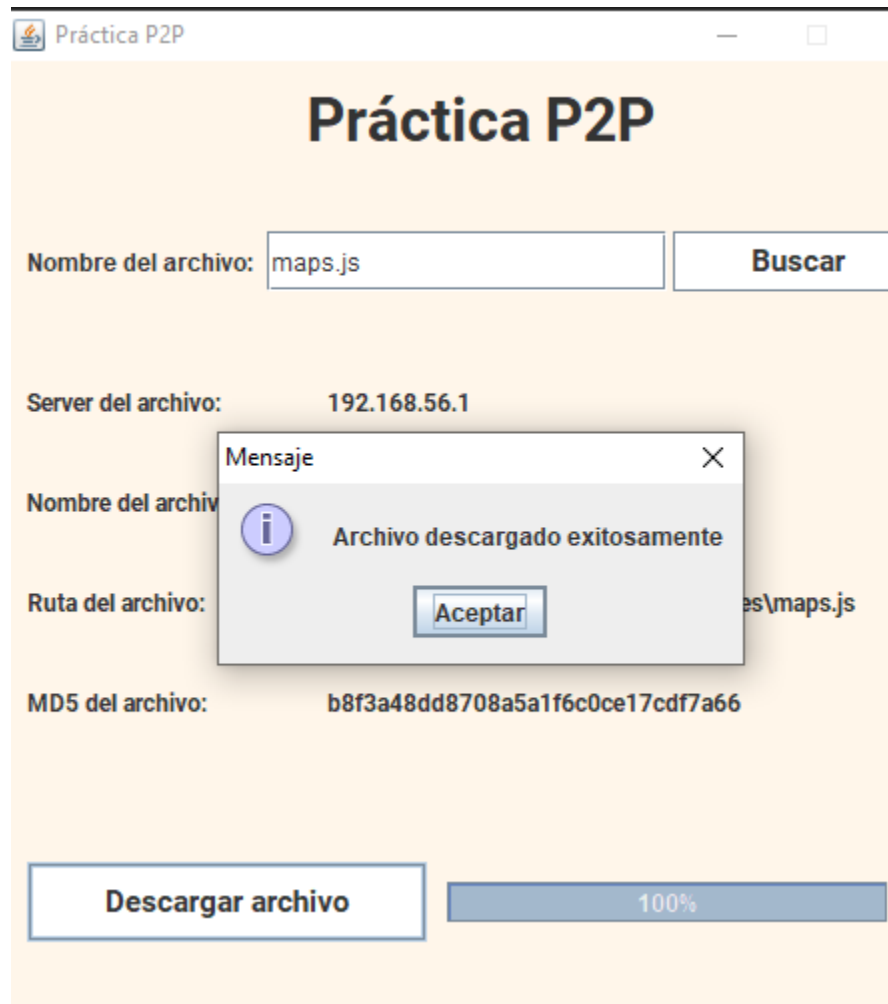


Ilustración 19. Archivo descargado.


| Almacenamiento (D:) > PC > Práctica7 > Práctica7 > Cliente > downloads | | | |
|---|------------------------|--------------------|--------|
| Nombre | Fecha de modificación | Tipo | Tamaño |
|  maps.js | 12/06/2022 02:16 a. m. | Archivo JavaScript | 2 KB |

Ilustración 20. Archivo guardado en carpeta correspondiente.

Conclusiones

La práctica, simuló un proyecto donde conjuntamos 3 conceptos importantes vistos durante el semestre. Primeramente, el concepto más significativo fue RMI que vimos nuevamente para búsqueda de los archivos. Como es un tema que vimos recientemente teníamos un poco de dificultad a elaborarlo. Luego usamos un multicast, que vimos en la práctica de aplicación de chat. Esta práctica se encarga de crear un hilo y se detectan de disponibilidad de servidor mediante el hilo. Últimamente implementamos unicast, que utilizamos en la práctica 1 para envío de archivos. Como la implementación del servidor de flujo o unicast fue igual que la anterior, no era necesario cambiar muchas cosas entonces fue más fácil implementarlo. Para algunos de nuestros integrantes todavía no quedó del todo claro con la implementación del MD5Checksum, ya que solo utilizamos el código ya hecho por el profesor. El funcionamiento fue satisfactorio, y logramos repasar lo que aprendimos en el transcurso del semestre en esta materia.

Bibliografía

1. Bonet, A. (2020). "Peer to Peer". Obtenido de :
<https://www.indracompany.com/es/blogneo/peer-peer-p2p-modelos-negocio-ciudadanos#:~:text=Una%20red%20peer%2Dto%2Dpeer,comportan%20como%20iguales%20entre%20s%C3%AD>.