E.T. Nº 36

ALMIRANTE GUILLERMO BROWN

Redes

TP N°14: Escáner de red: Documentación del Desarrollo

Año: 5º	División: 1º	Turno:	Tarde
Autor:		Sofía Power	
Docente: Oscar Obregón			
Fecha de entrega: 19/07/2025			

INTRODUCCIÓN DEL PROYECTO

El presente trabajo consiste en el desarrollo de una herramienta de red destinada al escaneo de direcciones IP, ya sea entre un rango predeterminado o por una dirección DNS.

Ante esto, la aplicación identificará los dispositivos de red encontrados y se los mostrará al usuario como información importante, brindando una solución accesible para el análisis básico de conectividad de una red local o externa.

OBJETIVOS Y FINALIDAD DEL SISTEMA

La aplicación permite hacer comunicaciones de red usando los protocolos **ICMP** (**ping**) y **DNS** con el fin de obtener información de la disponibilidad de los dispositivos. Para eso, el usuario deberá proporcionar el uso de los siguientes parámetros:

- **IP inicial:** Dirección que vamos a utilizar como el comienzo de un rango de equipos de red (ya sea si agregamos una IP final).

- **IP final:** Dirección que va a marcar el límite del rango de búsqueda. En el caso de que solo se quiera calcular una sola dirección, este campo puede dejarse vacío.

- **Dirección DNS:** Direcciones de dominio que sean compatibles. No está permitido el ingreso de direcciones IP, y si se requiere calcular solo este campo las IP inicial y final deben estar vacías. Se agrega este campo de texto para poder permitir dos direcciones de dominio distintas en la aplicación.

- **Tiempo de espera (timeout):** El usuario puede ingresar el tiempo de espera en milisegundos definido al ejecutar el comando ping. Está configurado de forma predeterminada con un tiempo de 1000 ms, aunque el usuario puede modificarlo según sus necesidades.

Cuando el usuario termine de proporcionar los parámetros requeridos sin ningún error de validación, podrá iniciar el proceso de escaneo presionando el botón “**Escanear**”.

Durante la ejecución se desplegará una ventana secundaria con una barra de progreso indicando el **estado de la operación**. Cuando llegue a su fin, todos los resultados encontrados se **mostrarán en la tabla principal** de la aplicación, junto con un cuadro de diálogo que indica la cantidad de equipos de red detectados por la solicitud.

Cada registro en la tabla incluye información como:

- **Dirección IP**
- El **nombre del equipo**
- Estado de **conectividad**
- **Tiempo de espera** calculado en milisegundos mediante el comando ping.

Además, la aplicación ofrece al usuario distintas funcionalidades para gestionar los resultados:

- Eliminar los registros que se muestren en pantalla presionando el botón “**Limpiar**”.

- Guardar los resultados, por lo que se permite exportarlos en un archivo, ya sea en formato **CSV** (**Excel**) o en archivo de **texto plano** (.**txt**) con solo presionar el botón “**Guardar**”.

- **Ordenar los registros** de menor a mayor utilizando cualquiera de las columnas disponibles.

- **Filtrar los datos** mostrando únicamente los dispositivos con conectividad, los que no respondieron, o todos los resultados disponibles.

De esta manera, la aplicación ofrece un conjunto de herramientas flexibles para el análisis y la organización de la información obtenida, ya sea mediante un rango de direcciones IP o por el uso de dominios específicos.

COMO ESTÁ ARMADO EL SISTEMA

El sistema fue organizado siguiendo el patrón de diseño Modelo-Vista-Controlador (MVC), por lo que permite una separación clara entre la lógica de información, la interfaz gráfica y la gestión de interacción con el usuario.

La organización del código se divide en diferentes paquetes:

- **Paquete "main":** Contiene la clase principal ***Main.java***, la cual se encarga de iniciar la aplicación y de que comience el controlador.

- **Paquete "controlador":** Incluye la clase ***Controlador.java***, que va a funcionar como un intermediario entre la interfaz gráfica (vistas) y la sección lógica de la aplicación (modelo). Administra las interacciones del usuario y coordina la ejecución de los procesos de red.

- **Paquete "modelo":** Aquí se encuentran todas las clases que van a servir para la lógica de datos del sistema:

    - ***ComienzoScanner.java***: Esta clase se encarga de conservar las clases principales que van a servir para la ejecución del programa (validar IP, ping, nslookup, escaneo en un rango y más).

    - ***listaEquiposRed.java***: Estructura que nos va a permitir guardar todos los resultados en una lista interna. Contiene funciones que se encargan de agregar y eliminar datos, y devolver los datos almacenados.

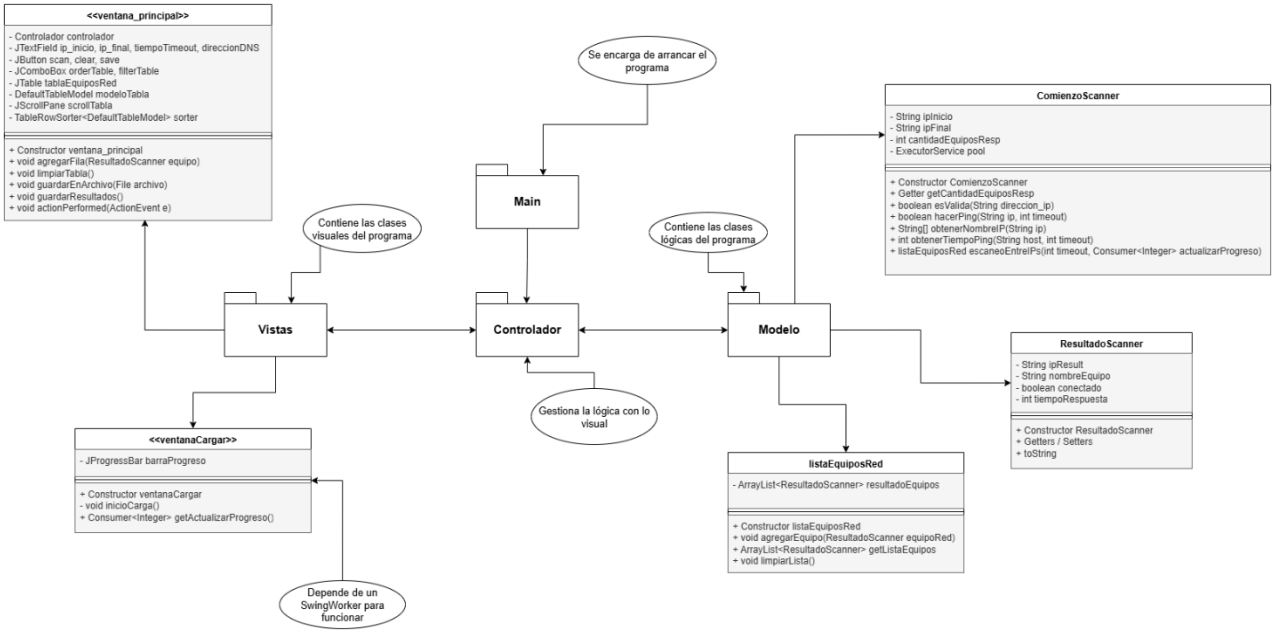
    - ***ResultadoScanner.java***: Se encarga de obtener una representación individual de cada dispositivo encontrado en la red. De esta forma se van a poder agregar a la lista de resultados de a poco.

- **Paquete “vistas”:** Contiene las clases que corresponden a la interfaz gráfica:

    - ***ventana_principal.java***: Es la vista principal que permite al usuario ingresar parámetros y visualizar los resultados.

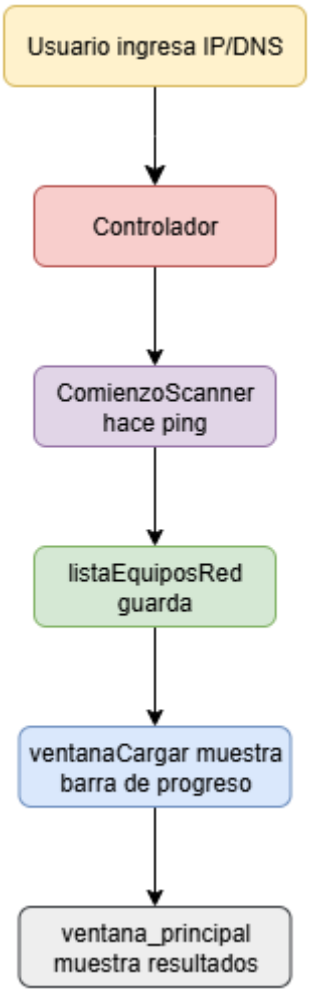
    - ***ventanaCargar.java***: Ventana secundaria que contiene la barra de progreso, indicando el estado del escaneo.

Diagrama UML de la organización del sistema:



En este diagrama UML están todos los paquetes ordenados uno por uno, cada uno con sus clases importantes. Por ejemplo, se muestran las dos clases visuales en el paquete Vistas (a la izquierda), y por la derecha están las clases lógicas en el paquete Modelo (a la derecha).

Procedimiento del programa:



MÉTODOS UTILIZADOS

En esta aplicación se utilizan distintos métodos que sirven para poder hacer una organización más ordenada de las distintas funciones del código. En este proyecto se hace el uso de varios métodos que se diferencian entre sí.

En esta sección se van a explicar todos los métodos utilizados en cada clase:

Clase “main”

```
public static void main(String[] args) {

        new Controlador();

}
```

Este método se utiliza principalmente en el sistema ya que se encarga de trabajar como un punto de entrada del programa. Su única tarea es instanciar la clase Controlador separando las responsabilidades y evitando que la clase Main tenga lógica adicional.

Clase “ResultadoScanner”

1- Constructor

```
public ResultadoScanner(String ipResult, String nombreEquipo, boolean
conectado, int tiempoRespuesta){

        this.ipResult = ipResult;
        this.nombreEquipo = nombreEquipo;
        this.conectado = conectado;
        this.tiempoRespuesta = tiempoRespuesta;
}
```

Este primer método es el constructor de la clase, y se encarga de inicializar los atributos o propiedades del objeto mientras se les asigna un valor inicial.

2- Getters & Setters

```
public String getIpResult(){ // Getter
        return ipResult;
}

    public void setIpResult(String ipResult){ // Setter
        this.ipResult = ipResult;
 }
```

```java
public String getNombreEquipo(){ // Getter
    return nombreEquipo;
}

public void setNombreEquipo(String nombreEquipo){ // Setter
    this.nombreEquipo = nombreEquipo;
}

public boolean isConectado(){ // Getter
    return conectado;
}

public int getTiempoRespuesta(){ // Getter
    return tiempoRespuesta;
}

public void setTiempoRespuesta(int tiempoRespuesta){ // Setter
    this.tiempoRespuesta = tiempoRespuesta;
}
```

Estos métodos van a tener un uso importante ya que de esta forma el resto de las clases van a acceder a los atributos privados de cualquier clase distinta, aplicando los principios de la **encapsulación**. Para eso sirven los Getter y Setter, y podemos diferenciar cada uno de la siguiente manera:

- **Getter:** Devuelve el valor de un atributo privado
- **Setter:** Modifica el valor de un atributo privado

En el caso de esta clase nos van a servir para modificar o devolver los datos que vayan a estar en los atributos IP, nombre de equipo, conectado y tiempo de espera.

3- Generar un toString para mostrar los atributos fácilmente

```
@Override
    public String toString() {
        return "IP: " + this.ipResult + ", Host: " + this.nombreEquipo
+ ", Responde: " + this.conectado + ", Timeout: " +
this.tiempoRespuesta;
    }
}
```

Es importante generarlo ya que devuelve una **representación legible del objeto en la lista**, mostrando cada uno de sus datos de importancia. De esta forma no van a haber errores legibles cuando se llame a la lista, ya que si no lo ponemos no nos va a devolver cada objeto, sino la referencia de la clase que creó la lista.

Clase “listaEquiposRed”

1- Constructor

```
public listaEquiposRed(){
        listaEquipos = new ArrayList<>();
    }
```

El primer método es el constructor de la clase, el cual se encarga de inicializar la lista de los equipos que se vayan a calcular en el escaneo.

2- AgregarEquipo(ResultadoScanner equipoRed)

```
public void agregarEquipo(ResultadoScanner equipoRed){
        listaEquipos.add(equipoRed);
}
```

En esta clase se identifica un método que agrega equipos a nuestra lista creada previamente. De esta forma se podrán guardar todos los resultados calculados en un corto período de tiempo.

3- Getter para obtener la lista

```
public ArrayList<ResultadoScanner> getListaEquipos(){
        return listaEquipos;
}
```

Su función principal se basa en obtener la lista creada y guardada en memoria, mientras se menciona la clase ResultadoScanner para devolver los equipos calculados que guardaron su información dentro de este.

4- limpiarLista()

```
public void limpiarLista(){
        listaEquipos.clear();
}
```

El último método de la clase es importante porque cada vez que tengamos que empezar otro escaneo, permite que los resultados calculados previamente queden eliminados para agregar los nuevos equipos de red que se calculen más tarde.

Clase “ComienzoScanner”

1- Constructor

```
public ComienzoScanner(String ipInicio, String ipFinal) {
      this.ipInicio = ipInicio;
      this.ipFinal = ipFinal;
}
```

Este constructor recibe los valores de la IP inicial y la IP final para luego inicializarlos y utilizarlos para el resto de los métodos en esta clase.

2- Obtiene la cantidad de equipos conectados

```
public int getCantidadEquiposRespuesta() {
        return cantidadEquiposRespuesta;
}
```

El segundo método de esta clase devuelve el total de equipos que responden a las solicitudes de conexión que hace el comando ping.

3- Verifica IP válida

```
public boolean esValida(String direccion_ip) {
        try {
            InetAddress address = InetAddress.getByName(direccion_ip);
            System.out.println(address);
            return address.isReachable(1000);
        }

        catch (Exception e) {
            return false; // IP o DNS inválido/fallido
        }
}
```

Este método se usa al inicio del escaneo ya que nos permite verificar si la IP ingresada está disponible para seguir con todo el cálculo siguiente. Para saber esto se obtiene la dirección ya calculada con el comando ping y con eso se verifica si hay errores de conexión.

4- Hace ping por consola

```
public boolean hacerPing(String ip, int timeout) {
        try {
            boolean responde =
InetAddress.getByName(ip).isReachable(timeout);
            if (responde == true) {
                cantidadEquiposRespuesta++;
            }

            return responde;
        }

        catch (IOException e) {
            System.err.println("Error al hacer ping a: " + ip);
            return false;
        }
    }
```

Este va a ser un método importante porque de esta forma vamos a poder identificar con el comando ping si la dirección responde a la solicitud. Si es así, se suma a la cantidad total de equipos que responden, y estará identificado como un dispositivo conectado.

5- Obtiene nombre de host y dirección IP

```
public String[] obtenerNombreIP(String ip) {
        try {
            InetAddress idHost = InetAddress.getByName(ip);

            String nombreHost = idHost.getCanonicalHostName();
            String direccionIP = idHost.getHostAddress();

            if (direccionIP.equals("127.0.0.1")) {
                nombreHost = "localhost";
            }

            return new String[] {
                    nombreHost,
                    direccionIP
            };
        }

        catch (UnknownHostException e) {
            System.err.println("Nombre no encontrado");
            return new String[] { "Nombre de Host desconocido",
"Dirección IP desconocida" };
        }
 }
```

Gracias a este método vamos a obtener el nombre de host del equipo, y si es necesario, la dirección IP. Todo esto se calcula gracias al comando ***nslookup***.

6- Obtiene el tiempo de respuesta de la solicitud

```
public int obtenerTiempoPing(String host, int timeout) {
        try {
            ProcessBuilder pb = new ProcessBuilder("ping", "-n", "1",
"-w", String.valueOf(timeout), host);
            Process process = pb.start();
```

```
            try (BufferedReader reader = new BufferedReader(new
InputStreamReader(process.getInputStream()))) {

                while ((line = reader.readLine()) != null) {
                    if (line.matches(".*\\d+ms.*")) {
                        String t = line.replaceAll(".*?(\\d+)ms.*",
"$1"); // Extrae solo el número

                        return Integer.parseInt(t.trim()); // Lo
convierte en int y luego lo devuelve
                    }
                }
            }
            process.waitFor(); // Espera a que el proceso ping
finalice antes de seguir
        }

        catch (IOException e) {
            e.printStackTrace();
        }

        catch (InterruptedException a){
            a.printStackTrace();
        }

        return timeout; // fallback si falla
    }
```

Este método sirve para obtener el tiempo de respuesta de la solicitud del ping, pero en tiempo real. Esto significa que toma como referencia el tiempo que ingresamos por pantalla y lo aplica al comando. Gracias a este método obtenemos el tiempo que tardó en responder cada equipo de red en su respectiva solicitud ping.

7- Proceso de escaneo

```
public listaEquiposRed escaneoEntreIPs(int timeout, Consumer<Integer>
actualizarProgreso) {
        cantidadEquiposRespuesta = 0;

        listaResultados.limpiarLista();

        // Caso en que haya IP única o dirección DNS
```

```
        if (ipFinal.isEmpty() ||
!ipFinal.matches("\\d+\\.\\d+\\.\\d+\\.\\d+")) {

            boolean responde = hacerPing(ipInicio, timeout);
            String[] datos = obtenerNombreIP(ipInicio);
            int tiempoRespuesta = obtenerTiempoPing(ipInicio,
timeout);

            listaResultados.agregarEquipo(new
ResultadoScanner(datos[1], datos[0], responde, tiempoRespuesta));
            actualizarProgreso.accept(100); // progreso completo

            return listaResultados;
        }

        String[] inicioPartes = ipInicio.split("\\.");
        String[] finPartes = ipFinal.split("\\.");

        int base1 = Integer.parseInt(inicioPartes[0]);
        int base2 = Integer.parseInt(inicioPartes[1]);
        int base3 = Integer.parseInt(inicioPartes[2]);
        int inicio = Integer.parseInt(inicioPartes[3]);
        int fin = Integer.parseInt(finPartes[3]);

        if (inicio > fin) {
            throw new IllegalArgumentException("La IP final no puede
ser menor que la inicial");
        }

        else{
            int totalIPs = fin - inicio + 1;
            AtomicInteger procesadas = new AtomicInteger(0);

            pool = Executors.newFixedThreadPool(20); // 20 hilos
simultáneos

            for (int i = inicio; i <= fin; i++) {

                String ip = base1 + "." + base2 + "." + base3 + "." +
i;


                pool.submit(() -> {
                    // No se va a interrumpir la interfaz gráfica

                    boolean responde = hacerPing(ip, timeout);

                    String[] datos = obtenerNombreIP(ip);
                    int tiempoRespuesta = obtenerTiempoPing(ip,
timeout);
```

```
                    synchronized (listaResultados) {
                        listaResultados.agregarEquipo(new
ResultadoScanner(
                                datos[1], datos[0], responde,
tiempoRespuesta));
                    }

                    int hechas = procesadas.incrementAndGet();
                    int porcentaje = (int) ((hechas / (double)
totalIPs) * 100);
                    actualizarProgreso.accept(porcentaje);
                });
            }

            pool.shutdown();

            try {
                pool.awaitTermination(2, TimeUnit.MINUTES);

            catch (InterruptedException e) {
                e.printStackTrace();
            }

            return listaResultados;
        }
    }
```

Este es el último método de la clase ComienzoScanner, y lo que hace es utilizar todos los métodos creados anteriormente para verificar el siguiente procedimiento:

- Si hay dos direcciones IP ingresadas, da comienzo al proceso en hilos del rango del último dígito de cada una, para calcular más equipos de red.

- Si solo hay una IP única, sólo se verifica la información de ese equipo de red.

Tal como está indicado, primero se verifica la dirección IP antes de llamar al método, y luego se hace el proceso de:

1- Hacer ping a cada dirección IP en el rango
2- Obtener el nombre de host del equipo (y la IP)

3- Sumar la cantidad de equipos conectados (si responden)
4- Agregar los equipos a la lista
5- Se actualiza el progreso de escaneo

Clase “Controlador”

1- Constructor de la clase

```
public Controlador() {
        listaResultados = new listaEquiposRed();

        ventanaPrinc = new ventana_principal(this);
        ventanaPrinc.setVisible(true);
    }
```

Sirve para inicializar y mostrar en pantalla la interfaz principal de la aplicación. También va a crear la lista de equipos para vincularla con todos los métodos lógicos de las clases anteriores.

2- Comenzar con el escaneo

```
public void startScan(String ipInicio, String ipFinal, int timeout) {

  // ComienzoScanner
        scanner = new ComienzoScanner(ipInicio, ipFinal);

        boolean responde1 = scanner.esValida(ipInicio);
        boolean responde2 = scanner.esValida(ipFinal);

        if (responde1 && responde2) {
            ventanaCargar ventanaLoad = new
ventanaCargar(ventanaPrinc);
            ventanaLoad.setModal(false);
            ventanaLoad.setVisible(true);

            // Ejecutar en hilo aparte para no bloquear la UI
            new Thread(() -> {
                try {
                    listaResultados = scanner.escaneoEntreIPs(timeout,
ventanaLoad.getActualizarProgreso());



                    SwingUtilities.invokeLater(() -> {
```

```
                        mostrarEquiposEnVista();
                        JOptionPane.showMessageDialog(ventanaPrinc,
                                "Escaneo completado. " +
scanner.getCantidadEquiposRespuesta()
                                        + " equipo(s)
encontrado(s).");
                    });


                }

                catch (IllegalArgumentException ex) {


                    SwingUtilities.invokeLater(() -> {
                        ventanaLoad.dispose();
                        JOptionPane.showMessageDialog(ventanaPrinc,
                                "Error en el rango de IPs ingresados",
"Error encontrado", JOptionPane.ERROR_MESSAGE);
                    });
                }
            }).start();

        }

        else {

            JOptionPane.showMessageDialog(ventanaPrinc, "Error. IP
inválida identificada, intente de nuevo.",
                    "Error encontrado", JOptionPane.ERROR_MESSAGE);
        }
    }
```

Su función principal es servir como un **intermedio** entre la parte lógica y la parte visual del programa. En este caso forma parte del cálculo entre **dos direcciones IP** distintas (inicial y final) para obtener los dispositivos encontrados entre ese rango, por lo que el proceso va a ser el siguiente:

1- Primero se verifica si las direcciones IP son correctas.

2- Si responden, se llama a una ventana secundaria que muestra el proceso de carga de la ejecución del escaneo.

3- Luego se ejecuta en hilos el método de la clase ComienzoScanner, para obtener el rango y las direcciones calculadas.

4- Por último, se ejecutan subprocesos múltiples para agregar cada elemento de la lista a la tabla principal de la aplicación.

De esta forma el usuario **recibirá todos los resultados posibles** y la cantidad de equipos que respondieron correctamente a la **solicitud** realizada con el comando **ping**.

3- Comenzar con el escaneo de una dirección DNS

```
public void startScanDNS(String dns, int timeout){
        scanner = new ComienzoScanner(dns, ""); // se pasa dns como
ipInicio y vacío en ipFinal

        if (scanner.esValida(dns)){
            ventanaCargar loading = new ventanaCargar(ventanaPrinc);
            loading.setModal(false);
            loading.setVisible(true);

            new Thread(() -> {
                try{
                    listaResultados = scanner.escaneoEntreIPs(timeout,
loading.getActualizarProgreso());

                    SwingUtilities.invokeLater(() -> {
                        mostrarEquiposEnVista();
                        JOptionPane.showMessageDialog(ventanaPrinc,
                            "Escaneo completado. " +
scanner.getCantidadEquiposRespuesta()
                                    + " equipo(s) encontrado(s).");
                    });
                }

                catch (IllegalArgumentException ex) {
                    SwingUtilities.invokeLater(() -> {
                        loading.dispose();
                        JOptionPane.showMessageDialog(ventanaPrinc,
                                "Error en el escaneo de la dirección
DNS", "Error encontrado", JOptionPane.ERROR_MESSAGE);
                    });
                }
            }).start();
        }
```

```
        else{
            JOptionPane.showMessageDialog(ventanaPrinc,
                "Error. DNS inválido o no responde, intente de
nuevo.",
                "Error encontrado", JOptionPane.ERROR_MESSAGE);
        }
    }
```

Este es otro posible caso de ejecución que sólo sucede si se ingresa **una dirección DNS** en su campo específico, y si no está ingresada **ninguna dirección IP**, ya sea inicial o final.

Este bloque de código tiene pasos similares al método anterior:

1- Primero se verifica si la dirección DNS es válida, y si es así llamamos a la función ***escaneoEntreIPs*** de la clase **ComienzoScanner**. Como mencionamos antes, esa función tiene una sección específica para calcular una sola dirección, ya sea IP o DNS.

2- Cuando finalice con ese bloque, volvemos a ejecutar en hilos la actualización de la tabla principal para agregar la información del equipo calculado.

De esta forma, si no se encuentra ningún fallo, el usuario va a poder recibir el equipo de red del que proviene la dirección DNS con sus datos importantes mediante nuestra interfaz gráfica.

4- Método privado para mostrar los equipos en la vista

```
private void mostrarEquiposEnVista() {

        ventanaPrinc.limpiarTabla();

        for (ResultadoScanner equipo :
listaResultados.getListaEquipos()) {
            ventanaPrinc.agregarFila(equipo);
        }
    }
```

En nuestra aplicación este método va a llamarse en las dos funciones anteriores del escaneo para vincularse con la ventana principal, agregando cada equipo de red que esté guardado en la lista a la tabla de información.

Antes de hacer eso se eliminan los datos existentes de la tabla para poder agregar nueva información requerida por el usuario, por eso se ejecuta una función específica de la vista, y luego se va agregando la información poco a poco.

Esta función privada de la clase Controlador nos sirve como una conexión con la vista principal para luego llamarla en las anteriores funciones lógicas, por lo que es importante para la organización interna que tiene la aplicación.

5- Limpiar la lista de la interfaz

```
public void clearList() {

        listaResultados.limpiarLista();

        ventanaPrinc.limpiarTabla();
}
```

Este va a ser el **segundo método conectado a la parte visual del programa**, y lo vamos a utilizar para eliminar todos los registros de la tabla si el usuario presiona el botón “Limpiar”. De esta forma se borran tanto los datos guardados en la lista como los mismos que se muestran por pantalla.

Clase “ventanaCargar”

1- Constructor

```
public ventanaCargar(JFrame ventanFrame){

        // Componentes importantes de la interfaz

        setTitle("Proceso de búsqueda");
        setSize(400, 150);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());

        // Estilo de la ventana (se agregan sus componentes, en este
caso una barra de progreso)

        JPanel laminaLoading = new JPanel(new GridBagLayout());
```

```
        GridBagConstraints gbc = new GridBagConstraints(); // El
GridBagConstraints sirve para ordenar cada componente de la interfaz
de forma customizada por el porgramador

        gbc.fill = GridBagConstraints.CENTER;
        gbc.anchor = GridBagConstraints.CENTER;

        gbc.insets = new Insets(10, 10, 10, 10);

        JLabel textLoading = new JLabel("Calculando los resultados.
Buscando si esta dirección existe...");
        gbc.gridx = 1;
        gbc.gridy = 0;
        gbc.gridwidth = 2;
        laminaLoading.add(textLoading, gbc);

        // Identificación de la JProgressBar

        barraProgreso = new JProgressBar(0, 100);
        barraProgreso.setPreferredSize(new Dimension(300, 25));
        barraProgreso.setStringPainted(true);
        gbc.gridy = 1;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.weightx = 1;
        laminaLoading.add(barraProgreso, gbc);

        add(laminaLoading);

        inicioCarga();

    }
```

Este método nos sirve mucho, ya que se encarga de **inicializar y crear la interfaz gráfica de nuestra ventana secundaria**. De esta forma se alinean sus componentes en un panel inicial (siendo un título y la barra de carga), y luego se identifica el inicio del proceso, comenzando del 0% hasta el 100% (esto se calcula en el próximo método privado de la clase).

2- Proceso de carga del escaneo

```
private void inicioCarga() {

        SwingWorker<Void, Integer> worker = new SwingWorker<>() {
```

```
            @Override
            protected Void doInBackground() throws Exception {
                for (int i = 0; i <= 100; i++) {
                    Thread.sleep(50); // Simula tiempo de carga
                    publish(i);              // Publica el progreso
                }
                return null;
            }

            @Override
            protected void process(java.util.List<Integer> chunks) {
                barraProgreso.setValue(chunks.get(chunks.size() - 1));
            }

            @Override
            protected void done() {
                dispose(); // cerrar ventana al terminar
            }
        };

        worker.execute();
    }
```

En este método vamos a hacer un procedimiento de carga que no congele la interfaz gráfica (UI), por lo que se ejecuta en segundo plano. Para esto vamos a utilizar un **SwingWorker**, que va a almacenar estas funciones secundarias:

1- La primera función Override (**doInBackground**) sirve para ir mostrando avances en segundo plano del progreso, sin ir trabando la UI

2- La segunda función interna (**process**) se utiliza para ejecutar los hilos de la interfaz gráfica para ir actualizando el progreso con el último valor que se reciba en chunks.

3- Y la última función (**done**) se va a ejecutar cuando terminemos con los avances en segundo plano, mientras se cierra la ventana secundaria.

3- Actualización de la barra de carga

```
public Consumer<Integer> getActualizarProgreso() {
        return (porcentaje) -> SwingUtilities.invokeLater(() -> {
            barraProgreso.setValue(porcentaje);
            if (porcentaje >= 100) {
                dispose();
            }
        });
    }
```

Este último método devuelve una función que recibe un Integer que utiliza desde otros hilos para **actualizar la barra de progreso** de forma segura. Además, asegura que los cambios se realicen en el UI, y cuando se llega al 100%, la barra se cierra automáticamente.

Clase “ventana_principal”

1- Constructor

```
public ventana_principal(Controlador controller) {

        setTitle("Escaner de Red");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(800, 650);
        setLocationRelativeTo(null);
        setLayout(new BorderLayout());

        this.controller = controller;

        JPanel lamina = new JPanel(new GridBagLayout());
        GridBagConstraints gbc = new GridBagConstraints(); /
        gbc.fill = GridBagConstraints.CENTER;

        gbc.insets = new Insets(8, 8, 8, 8);

        JLabel textoIntro = new JLabel("Ingrese los siguientes campos
de información para poder hacer el proceso");
        gbc.gridx = 0;
        gbc.gridy = 0;
        gbc.gridwidth = 4;
        lamina.add(textoIntro, gbc);

        ip_inicio.setInputVerifier(new InputVerifier() {
            @Override
            public boolean verify(JComponent input) {
```

```
                String texto = ((JTextField) input).getText().trim();

                if (texto.isEmpty()) {
                    return true;
                }

                if (texto.matches("^((25[0-5]|2[0-
4]\\d|[01]?\\d\\d?)\\.){3}(25[0-5]|2[0-4]\\d|[01]?\\d\\d?)$")) {
                    return true;
                }

                else {
                    JOptionPane.showMessageDialog(null, "IP inválida:
" + texto, "Error", JOptionPane.ERROR_MESSAGE);
                    return false;
                }
            }
        });

        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.gridwidth = 1;
        gbc.gridx = 0;
        gbc.gridy = 1;
        lamina.add(new JLabel("IP inicial:"), gbc);

        gbc.gridx = 2;
        lamina.add(ip_inicio, gbc);

        ip_final.setInputVerifier(new InputVerifier() {
            @Override
            public boolean verify(JComponent input) {
                String texto = ((JTextField) input).getText().trim();

                if (texto.isEmpty()) {
                    return true;
                }

                if (texto.matches("^((25[0-5]|2[0-
4]\\d|[01]?\\d\\d?)\\.){3}(25[0-5]|2[0-4]\\d|[01]?\\d\\d?)$")) {
                    return true;
                } else {
                    JOptionPane.showMessageDialog(null,
                            "IP inválida: " + texto,
                            "Error", JOptionPane.ERROR_MESSAGE);
                    return false;
                }
            }
        });
```

```
        gbc.gridy = 2;
        gbc.gridx = 0;
        lamina.add(new JLabel("IP final:"), gbc);
        gbc.gridx = 2;
        lamina.add(ip_final, gbc);

        direccionDNS.setInputVerifier(new InputVerifier() {
            @Override
            public boolean verify(JComponent input) {
                String texto = ((JTextField) input).getText().trim();
                if (texto.isEmpty()){
                    return true; // vacío = válido
                }

                if (texto.matches("^[a-zA-Z0-9.-]+$")) {
                    return true; // Si es DNS, sigue con el programa
                }

                else {
                    JOptionPane.showMessageDialog(null,
                            "DNS inválido: " + texto,
                            "Error", JOptionPane.ERROR_MESSAGE);
                    return false;
                }
            }
        });

        gbc.gridx = 0;
        gbc.gridy = 3;
        lamina.add(new JLabel("Dirección DNS (opcional): "), gbc);

        gbc.gridx = 2;

        lamina.add(direccionDNS, gbc);

        gbc.gridy = 4;
        gbc.gridx = 0;
        lamina.add(new JLabel("Tiempo de espera (ms)"), gbc);
        gbc.gridx = 2;
        lamina.add(tiempoTimeout, gbc);

        // Panel de botones

        JPanel panelBotones = new JPanel(new
FlowLayout(FlowLayout.CENTER, 50, 0)); // 50 px de espacio entre
botones
        scan.addActionListener(this);
        clean.addActionListener(this);
        save.addActionListener(this);
```

```
        panelBotones.add(scan);
        panelBotones.add(clean);
        panelBotones.add(save);

        gbc.anchor = GridBagConstraints.CENTER;
        gbc.fill = GridBagConstraints.CENTER;
        gbc.gridx = 0;
        gbc.gridy = 5;
        gbc.gridwidth = 3;

        lamina.add(panelBotones, gbc);

        JLabel presentarTabla = new JLabel("Tabla de información;
todos los resultados van a aparecer a continuación");

        gbc.fill = GridBagConstraints.CENTER;
        gbc.gridy = 6;

        lamina.add(presentarTabla, gbc);

        // Comienza la estructura de la JTable

        modeloTabla = new DefaultTableModel();
        modeloTabla.setColumnIdentifiers(
                new String[] { "Dirección de IP", "Nombre de equipo",
"Conectado", "Tiempo (ms)" });

        tablaEquiposRed = new JTable(modeloTabla);

        sorter = new TableRowSorter<>(modeloTabla);
        tablaEquiposRed.setRowSorter(sorter);

        // ---- Primera columna: Direcciones IP ----

        sorter.setComparator(0, (ip1, ip2) -> {
            String[] partes1 = ((String) ip1).split("\\.");
            String[] partes2 = ((String) ip2).split("\\.");
            for (int i = 0; i < 4; i++) {
                int n1 = Integer.parseInt(partes1[i]);
                int n2 = Integer.parseInt(partes2[i]);
                if (n1 != n2)
                    return Integer.compare(n1, n2);
            }
            return 0;
        });

        // ---- Segunda columna: Nombres de cada equipo ----
```

```
        sorter.setComparator(1, (n1, n2) -> {
            boolean esIp1 = ((String)
n1).matches("\\d+\\.\\d+\\.\\d+\\.\\d+");
            boolean esIp2 = ((String)
n2).matches("\\d+\\.\\d+\\.\\d+\\.\\d+");
            if (esIp1 && !esIp2)
                return 1; // nombres primero
            if (!esIp1 && esIp2)
                return -1;
            return ((String) n1).compareToIgnoreCase((String) n2);
        });

        // ---- Cuarta columna: Tiempo de espera ----

        sorter.setComparator(3, (t1, t2) -> {
            int n1 = Integer.parseInt(((String) t1).replace("ms",
""));
            int n2 = Integer.parseInt(((String) t2).replace("ms",
""));
            return Integer.compare(n1, n2);
        });



        scrollTabla = new JScrollPane(tablaEquiposRed);
        scrollTabla.setPreferredSize(new Dimension(680, 260));

        gbc.fill = GridBagConstraints.BOTH;
        gbc.gridy = 7;
        gbc.gridwidth = 3;

        // Se agrega la tabla con el JScrollPane implementado

        lamina.add(scrollTabla, gbc);

        // Organozación de los JComboBox de opciones

        // Primero se hace el que ordena los resultados

        orderTable.setModel(
                new DefaultComboBoxModel(
                        new String[] { "--- Ordenar por ---", "IP",
"Nombre", "Estado (conectado o no)", "Tiempo" }));

        orderTable.setBounds(10, 36, 191, 20);
        orderTable.setSelectedIndex(0);
        orderTable.addActionListener(this);

        gbc.fill = GridBagConstraints.EAST;
```

```
        gbc.gridy = 8;
        gbc.gridwidth = 1;

        lamina.add(orderTable, gbc);

        // Ahora se hace el que filtra los resultados

        filterTable.setModel(
                new DefaultComboBoxModel(
                        new String[] { "--- Filtrar por ---", "Todos",
"Solo conectados", "Solo desconectados" }));

        filterTable.setBounds(10, 36, 191, 20);
        filterTable.setSelectedIndex(0);
        filterTable.addActionListener(this);

        gbc.gridx = 2;
        gbc.fill = GridBagConstraints.EAST;

        lamina.add(filterTable, gbc);

        add(lamina);
    }
```

Este constructor nos sirve para poder identificar cada componente de la ventana principal, ya sean:

1- **Los campos de texto** (para ingresar IP inicial, IP final, dirección DNS y tiempo de espera).
2- **Botones (Escanear, Limpiar y Guardar)**, los cuales se les implementa un ActionListener para poder abrir ciertas funciones al presionar alguno de ellos.
3- **Tabla con una ScrollBar** para mostrar todos los resultados calculados.
4- **Opciones para ordenar la tabla y filtrar información por conectados o desconectados** (También se les implementa un ActionListener).

De esta misma forma en este constructor se utiliza una opción de ordenamiento de los componentes llamada **GridBagConstraints**, el cual se va a inicializar gracias al uso de un **GridBagLayout** en la organización del programa. Su uso en este sistema es ordenar mediante Grids de una forma customizada por el programador, por lo que puede elegir su ubicación vertical y horizontalmente, elegir su ancho de ocupación y más.

Hay otras funciones internas que se basan en el control de errores. Por ejemplo, se usan verificadores de Inputs que van a comprobar si las IP escritas son correctas, funciones para filtrar información y más.

2- Agregar filas en la tabla

```
public void agregarFila(ResultadoScanner equipo) {
        modeloTabla.addRow(new Object[] { equipo.getIpResult(),
equipo.getNombreEquipo(), equipo.isConectado(),
                equipo.getTiempoRespuesta() + "ms" });
    }
```

Nos va a servir para agregar **nuevas filas de registros** a la tabla, utilizando la función **addRow()**. Para hacer esto vamos a tener que mencionar directamente la clase ResultadoScanner, pero solo para obtener cada uno de los datos ingresados y separarlos por cada columna específica. De esta forma también nos van a servir los getters de esa clase.

Este método lo vamos a llamar en las funciones de la clase Controlador **"startScan"** y **"startScanDNS"**, así se agregan todos los resultados rápidamente mediante hilos.

3- Limpiar tabla

```
public void limpiarTabla() { // Elimina los registros de la tabla
        modeloTabla.setRowCount(0);
}
```

Nos va a servir para identificar que la cantidad de filas de la tabla **se vuelve cero**, ya que lo podemos utilizar cuando la llamemos en la función de la clase Controlador, para que luego podamos **identificar fácilmente los errores**. Como, por ejemplo, si queremos guardar los datos en un archivo, no vamos a tener ninguna fila, entonces esto nos puede ayudar para que nuestro sistema comprenda que no hay ninguna tabla.

Además, cuando la cantidad de filas de la tabla se vuelve cero, se **eliminan todos los registros directamente**, así que lo llamamos en Controlador para que todos los datos se eliminen, tanto en la lista como en la vista principal.

4- Guardar archivos (método lógico)

```
private void guardarEnArchivo(File archivo) {
        try (PrintWriter pw = new PrintWriter(new
FileWriter(archivo))) {
            // Encabezados
            for (int i = 0; i < modeloTabla.getColumnCount(); i++) {
                pw.print(modeloTabla.getColumnName(i));
                if (i < modeloTabla.getColumnCount() - 1)
                    pw.print(",");
            }
            pw.println();

            // Filas
            for (int fila = 0; fila < modeloTabla.getRowCount();
fila++) {
                for (int col = 0; col < modeloTabla.getColumnCount();
col++) {
                    pw.print(modeloTabla.getValueAt(fila, col));
                    if (col < modeloTabla.getColumnCount() - 1)
                        pw.print(",");
                }
                pw.println();
            }

            JOptionPane.showMessageDialog(this, "Resultados guardados
en: " + archivo.getAbsolutePath());
        } catch (IOException e) {
            JOptionPane.showMessageDialog(this, "Error al guardar
archivo: " + e.getMessage());
        }
    }
```

Vamos a utilizar este método para que se transcriba cada encabezado de las columnas, como cada fila en la tabla, copiándolos en un archivo de texto. Para que pase esto no debemos tener ningún error de entrada y salida, porque si pasa esto no nos va a funcionar.

Además, de esta forma tenemos que identificar la cantidad de columnas y tablas, e ir recorriéndolas poco a poco.

De esta forma, este método se va a llamar en la función que se explicará a continuación.

5- Guardar archivos (método visual)

```
private void guardarResultados() {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setDialogTitle("Guardar resultados");
        fileChooser.setSelectedFile(new File("resultados.csv"));

        int opcion = fileChooser.showSaveDialog(this);
        if (opcion == JFileChooser.APPROVE_OPTION) {
            File archivo = fileChooser.getSelectedFile();
            guardarEnArchivo(archivo);
        }
    }
```

En este caso este método nos va a mostrar una ventana secundaria que recolecta un conjunto de archivos del dispositivo para poder elegir un directorio, y de esta forma vamos a guardar los datos en un archivo.

Además, vamos a llamar a la función lógica para guardar toda la información, y también podemos elegir el tipo de archivo con el que vamos a guardar todo (Archivo de texto .txt, archivo CSV).

6- Función que indica lo que pasa al pulsar botones/opciones}

Esto lo vamos a dividir en dos partes importantes:

1- Botones:

```
@Override
    public void actionPerformed(ActionEvent e) {
        try {
            Object source = e.getSource();

            String ipInicioIngre = ip_inicio.getText();
            String ipFinalIngre = ip_final.getText();
            String direDNSingre = direccionDNS.getText();

            int tiempo_espera =
Integer.parseInt(tiempoTimeout.getText());

            if (source == scan) {

                if (!direDNSingre.isEmpty()) {
                    if (ipInicioIngre.isEmpty() &&
ipFinalIngre.isEmpty()) {
```

```
                        controller.startScanDNS(direDNSingre,
tiempo_espera);
                    } else {
                        JOptionPane.showMessageDialog(this,
                                "Si ingresa DNS, los campos de IP
deben estar vacíos",
                                "Error de validación",
JOptionPane.WARNING_MESSAGE);
                    }
                } else {
                    if (ipInicioIngre.isEmpty() &&
!ipFinalIngre.isEmpty()) {
                        JOptionPane.showMessageDialog(this, "Debe
ingresar más información", "Mensaje de alerta",
                                JOptionPane.WARNING_MESSAGE);
                    }

                    else {
                        try {
                            controller.startScan(ipInicioIngre,
ipFinalIngre, tiempo_espera);
                        }

                        catch (IllegalArgumentException a) {
                            JOptionPane.showMessageDialog(this, "Error
de cálculo con las IPs: " + a,
                                    "Error encontrado",
                                    JOptionPane.ERROR_MESSAGE);
                        }
                    }

                }

            }

            if (source == clean) {
                controller.clearList();
            }

            if (source == save) {
                if (modeloTabla.getRowCount() == 0){
                    JOptionPane.showMessageDialog(this, "No tiene
ningún registro para guardar en un archivo", "Error encontrado",
JOptionPane.ERROR_MESSAGE);
                }

                else{
                    guardarResultados();
                }
```

```
            }
        // Sigue el código…
}
```

En esta primera sección del código vamos a conseguir que objeto se presiona en nuestra aplicación, y de esta forma vamos a saber qué información o funciones se llamarán:

- **Si utilizamos el botón “Escanear”, se van a calcular los siguientes datos:**
    - Si se ingresó una **dirección DNS** y no se identifica **ninguna dirección IP inicial y final** escritas, el programa va a llamar a la función del controlador que sirve para escanear la dirección **DNS**. Si se encuentran direcciones de un rango, lanza una ventana de error.
    - Si se ingresaron **dos direcciones de rango inicial y final**, se llama al otro método del controlador que escanea los equipos en una **cantidad determinada**. Si no se ingresa una dirección final, lanza una ventana de error indicando que no se puede calcular.
- **Si presionamos el botón “Limpiar”, se van a eliminar los resultados tanto en la tabla como en la lista.**
- **Si presionamos el botón “Guardar”, va a saltar la ventana secundaria JFileChooser para guardar en nuestro dispositivo toda la información mostrada, si es que se encuentran dispositivos en nuestra tabla.**

2- Ordenar y filtrar información:

```
        // (Código actionPerformed de botones anteriormente…)
           String opcionOrdenar = (String)
orderTable.getSelectedItem();

           if (opcionOrdenar.equals("--- Ordenar por ---")) {
               sorter.setSortKeys(null);
           }
```

```
            else if (opcionOrdenar.equals("IP")) {
                sorter.setSortKeys(java.util.List.of(new
RowSorter.SortKey(0, SortOrder.ASCENDING)));
            }

            else if (opcionOrdenar.equals("Nombre")) {
                sorter.setSortKeys(java.util.List.of(new
RowSorter.SortKey(1, SortOrder.ASCENDING)));
            }

            else if (opcionOrdenar.equals("Estado (conectado o no)"))
{
                sorter.setSortKeys(java.util.List.of(new
RowSorter.SortKey(2, SortOrder.DESCENDING)));
            }

            else if (opcionOrdenar.equals("Tiempo")) {
                sorter.setSortKeys(java.util.List.of(new
RowSorter.SortKey(3, SortOrder.ASCENDING)));
            }

            String opcionFiltrar = (String)
filterTable.getSelectedItem();
            if (opcionFiltrar.equals("--- Filtrar por ---")) {
                return;
            }

            else if (opcionFiltrar.equals("Todos")) {
                sorter.setRowFilter(null); // Saca los filtros
            }

            else if (opcionFiltrar.equals("Solo conectados")) {
                sorter.setRowFilter(RowFilter.regexFilter("true", 2));
// Muestra solo conectados
            }

            else if (opcionFiltrar.equals("Solo desconectados")) {
                sorter.setRowFilter(RowFilter.regexFilter("false",
2)); // Muestra solo desconectados
            }
        }

        catch (Exception a) {
            JOptionPane.showMessageDialog(this, "Tuvo un error en su
programa.\n" + a, "Error encontrado",
                    JOptionPane.ERROR_MESSAGE);
        }
    }
```

Esta es otra opción del mismo método, el cual obtiene en un valor separado de la opción elegida en el JComboBox de ordenar los resultados, y del JComboBox que filtra los resultados.

De esta forma se indica si el usuario desea ordenar de forma ascendente o descendente mediante:

- Dirección IP
- Nombre de equipo
- Estado de conectividad
- Tiempo de espera (ms)

Además, una opción para el usuario puede ser filtrar la información por:

- Todos los equipos de red
- Mostrar sólo conectados
- Mostrar sólo desconectados

¿POR QUÉ SE ELIGIERON CIERTAS TECNOLOGÍAS?

En esta sección se van a explicar tanto el lenguaje de programación como las librerías elegidas para la creación de esta aplicación.

Esta aplicación fue creada con el lenguaje **Java**, ya que tiene una programación orientada a objetos que es más simple y entendible para crear una aplicación, o al menos ese es mi caso, ya que lo estuvimos utilizando últimamente.
Además, se eligió esta tecnología ya que otorga librerías de red para calcular los comandos requeridos para el objetivo principal, y porque se puede crear una interfaz gráfica rápida y completa.

Para comprender mejor las librerías utilizadas, se nombrarán a continuación:

- **Swing**: Es importante en este proyecto ya que otorga los componentes necesarios para crear una interfaz gráfica de usuario (**GUI**), que es en pocas palabras, la parte visual de la aplicación. También se utilizó **Swing.table** para desarrollar el modelo de la tabla principal. Además, fue de utilidad para mostrar recuadros de información **JOptionPane** para mostrar errores o si se cumplieron las operaciones, y hace posible ejecutar operaciones mediante hilos (**SwingUtilities**).

- **AWT**: Esta tecnología de Java otorga al código un conjunto de herramientas que van a servir para **configurar la GUI**, como botones, etiquetas, cuadros de texto y más. Gracias a esta también se implementó un sistema de eventos para manejar mejor las interacciones del usuario (pulsar botones o teclas y más), los cuales generan eventos procesados en “listeners”.

- **IO**: Este paquete es importante, ya que sirvió para realizar operaciones de entrada y de salida, que en este caso fue de utilidad para guardar los resultados mostrados en un archivo interno del dispositivo. Además, identifica los errores de entrada y salida que vayan a ocurrir.

- **Util**: Esta otra librería utilizada sirvió principalmente para proporcionar **un conjunto de clases e interfaces** que ofrecen funcionalidades esenciales para desarrollar aplicaciones en Java. De esta forma ofrece colecciones de datos (**ArrayList y List**), manejo de hilos con sincronización y ejecución de tareas (**Concurrent**), manejar números enteros de forma segura en hilos, y utilizar interfaces funcionales para operaciones comunes y expresiones lambda.

- **NET**: Este paquete proporciona clases para **implementar distintas aplicaciones de red**, dividido en dos secciones principales: una **API de bajo nivel** que maneja direcciones (como direcciones IP) y **sockets** para la comunicación bidireccional. Las clases importadas para hacer posible este trabajo fueron **InetAddress**, para usar el protocolo IP correctamente, y **UnknownHostException**, importante para indicar cuando una dirección IP del host no fue encontrada.

Gracias a todas estas librerías de Java, fue posible la implementación de comandos, operaciones de almacenamiento internas y más, para que de esa forma la aplicación pueda cumplir con sus objetivos tal como se mencionó anteriormente.

PROBLEMAS ENCONTRADOS Y SUS SOLUCIONES

En la organización de este proyecto, hubo varios errores e inconvenientes que fueron deteniendo el flujo de información y también al exportar la aplicación. En este apartado voy a explicar algunas de estas interrupciones que hubo al codificar la aplicación:

1- Diferenciar una dirección IP con una DNS

Este caso fue uno de los más rápidos de resolver, ya que una de sus soluciones fue crear dos funciones distintas en el controlador del programa, siendo una para comenzar el escaneo entre dos direcciones IP y una para escanear la dirección DNS. Pero el problema estuvo en que, como se podían ingresar ambas direcciones en el campo de IP inicial, hubo confusiones para saber si era necesario realizar esto. Por eso la segunda solución fue agregar otro campo de texto justamente indicado únicamente para ingresar por pantalla una dirección DNS.

Otra solución fue que, si no se ingresaba ninguna dirección en ningún campo, se mostrara por pantalla la información del equipo de red local, representando al equipo que utiliza el usuario (localhost).

2- Problemas de coordinación con la barra de progreso

Para esta sección tuve algunas dudas pequeñas que tienen que ver con el orden de procedimientos para agregar la información. Al principio pensé que, mientras se mostraba por pantalla la ventana secundaria (clase ventanaCargar), que se vayan cargando los datos a la tabla principal, mostrándole al usuario cada resultado calculado en tiempo

real, uno por uno. Pero no pude resolver esto porque, como se utilizaban muchos hilos simultáneos, la aplicación se fue trabando poco a poco para que al final no se mostraran los resultados después de un tiempo.

Este inconveniente se pudo resolver haciendo el proceso por partes: Primero se comprueban las direcciones IP, luego aparece la barra de proceso calculando el tiempo de espera para visualizar los resultados, y por último se muestran por pantalla la información de cada uno de los resultados calculados.

3- Problemas de comprobación de la ArrayList de resultados

Este pequeño error se trataba de calcular una de las primeras partes lógicas y probar mostrarlas por consola, antes de hacerlo en la tabla. Pero el tema estaba en que, aunque la lista estaba creada perfectamente, en consola sólo me aparecía una vinculación que mencionaba el directorio que estaba creada la ArrayList.

Para solucionar este problema se creó el toString en la clase ResultadoScanner, así se indicaba por consola la representación de un objeto de forma legible. De esta forma se obtuvo el IP, el nombre del equipo, el estado de conectividad y el tiempo de respuesta de ese objeto, sin ningún otro inconveniente.

4- Estilo de la aplicación principal

Este fue un inconveniente muy común para armar la estructura del proyecto, ya que para armar el estilo de la interfaz gráfica se fue diferenciando en columnas y tablas, y en cada una se fueron agregando los componentes. Mi problema fue que no era una interfaz amigable para el usuario al principio; Una de las primeras ideas fue mostrar la información en otra ventana secundaria que aparezca cuando la barra de progreso termine. Pero se terminó descartando porque no pude figurar la posibilidad para hacer eso.

Además tuve problemas mínimos de desorden al usar un GridBagLayout y GridBagConstraints, pero se logró solucionar. Ahora la aplicación cumple con una interfaz que puede resultar amigable para el usuario.

COSAS PARA MEJORAR EN UN FUTURO

En este último apartado de la documentación se explicarán cosas a fondo que no se lograron emplear en la aplicación, pero que pueden actualizarse en un futuro para que de esta forma sea más completo y seguro.

Una de las ideas principales sería poder tener menos tiempo de espera para mostrar algunos resultados en la tabla. Por ejemplo, si ponemos dos IP con un valor más largo para calcular un rango de dispositivos (como 127.0.0.1 y 127.0.0.35), cuando termine la barra del progreso no se van a mostrar los resultados al momento, sino después de 3 o 5 segundos. Esta posible mejora puede servir para evitar confusiones de cálculo en el usuario, y para que ese proceso se realice sin agregar un tiempo adicional.

Otra cosa para recalcar es que se espera mejorar la precisión del progreso de escaneo, ya que a veces la barra que va del 0% al 100% se traba y vuelve para atrás o para adelante, como si repitiera los números. Ante esto la mejora esperada es que, por la ejecución de hilos, no haya ningún inconveniente ni confusión.

Ante esto otra posible mejora es poder mencionar los paquetes enviados a dicha dirección específica, haciendo que suceda con alguna función que los indique. De esa forma, el usuario sabrá si todos los paquetes fueron enviados correctamente a su destinatario sin ninguna pérdida de información en el camino. Igualmente esto es una posible idea que se puede mostrar como información adicional, además de la cantidad de equipos que respondieron a solicitudes de ping.

Por último, una posible mejora en el futuro puede ser emplear un rango de direcciones IP que llegue a calcularse gracias a más dígitos. Por ejemplo, en este sistema se utilizan solo para el último dígito, como por ejemplo, si hacemos ping desde la dirección 8.8.8.8 hasta la dirección 127.0.0.11, el rango de dispositivos sería de 8 a 11 pero calculándolos mediante la IP inicial (o sea que se calculan 8.8.8.8, 8.8.8.9, etc.). La idea sería que se pueda pasar tanto al tercer dígito como al segundo, pero que eso sea a elección del usuario.

Ante esto puede haber una opción en la aplicación que sea directamente para el usuario, de que, si quiere calcular los equipos de red en un rango de números, puede decidir esto con una opción para elegir un dígito específico de ambas direcciones IP, para poner un principio y final. Pero esto es una idea extra para hacer más completa nuestra aplicación.

CONCLUSIÓN DEL PROYECTO

En este proyecto se utilizaron muchas prácticas, tanto visuales como lógicas del programa, pero se pudo implementar una buena herramienta práctica para poder realizar un escaneo de redes gracias a los protocolos ICMP y DNS. Este sistema en general cumple con su objetivo principal de detectar los dispositivos que estén disponibles y luego brindarle esta información al usuario en tiempo real. Además, se lograron aplicar varios conceptos de la programación orientada a objetos, ya sea el manejo de hilos y la separación de clases mediante el patrón MVC.

Este trabajo se abre a una gran posibilidad de mejoras a futuro respecto a rendimiento y utilidad del usuario, permitiendo que la aplicación tenga una base sólida para un análisis de redes.