Practical Assessment Task

Phase 4.1 – Technical Document

Igor Karbowy

Table of Contents

4.1.1 — Externally Sourced Code

There is one instance of externally sourced code in *PingPal*. The code which has been externally sourced is the code for the loop which is responsible for the subnet scan functionality—namely, scanning a number of IP addresses concurrently. However, the code has been considerably edited from the original code to fit the needs of *PingPal*. The aforementioned code is found on the following pages.

The code was sourced from the *The Software Developer* website—namely, an article titled *Create a Multi Threaded IP Scanner with Java*. The reference for the article is found below—

Raz, D. (2021). *Create a Multi Threaded IP Scanner with Java.* [online] The SW Developer. Available at: https://www.theswdeveloper.com/post/scan-for-devices-in-the-network-with-java [Accessed 25 Feb. 2025].

The original code from which I wrote the adaptation for *PingPal* is found on the following pages, after the code from *PingPal*.

It is also important to mention that the *PortScan* class uses a similar loop to scan the ports of a device concurrently. However, that code was even further adopted from the code from the *SubnetScan* class. Nevertheless, for the purpose of certainty, I have also attached the code from the *PortScan* class and declared it as externally sourced code. That code is found after the original code from the *The Software Developer* website.

The code in the *SubnetScan* class in *PingPal* being referred to—

```java
    /**
     * Starts the subnet scan.
     * <p>
     * Clears the table and progress bar, then concurrently scans all IPs in the
     * subnet. Uses an ExecutorService to run scan tasks concurrently and
     * updates the progress bar.
     * </p>
     * <p>
     * The method executes until either all tasks complete, the timeout is
     * reached, or a stop is requested.
     * </p>
     */
    public void start() {
        // Clear current data in the table on the EDT.
        invokeLater(() -> model.setRowCount(0));
        // Reset the progress bar on the EDT.
        invokeLater(() -> prgSubnetScan.setValue(0));

        // Counter variable for IPs to scan.
        AtomicInteger ips = new AtomicInteger(0);
        // Counter variable for already scanned IPs.
        AtomicInteger scannedIps = new AtomicInteger(0);

        // Loop through each IP index in the range.
        while (ips.get() <= numOfIPs) {
            // Generate the IP to scan.
            String ip = generateIP(ips.getAndIncrement());

            // Submit a task to the executor to scan the IP.
            executorService.submit(() -> {
                InetAddress inAddress;
                try {
                    // Resolve the IP address.
                    inAddress = InetAddress.getByName(ip);
                    // Scan the IP.
                    scanIP(inAddress);

                    // Update the progress bar after scanning each IP.
                    updateProgressBar(scannedIps.getAndIncrement());
                } catch (UnknownHostException e) {
                    // If IP is unknown, ignore and continue.
                }
            });
        }

        // Initiate shutdown and wait for tasks to finish, with a maximum wait time.
        executorService.shutdown();
        try {
            // Wait until all tasks have finished, or timeout after 10 minutes.
            if (!executorService.awaitTermination(10, TimeUnit.MINUTES)) {
                // If tasks are not finished in 10 minutes, force shutdown.
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            // If the thread is interrupted, reset the interrupt flag and exit the loop.
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
```

```
    /**
     * Attempts to ping the single given InetAddress. If reachable, adds the IP
     * address to the results list and updates the table model.
     *
     * @param inAddress the InetAddress representing the IP to scan
     */
    private void scanIP(InetAddress inAddress) {
        try {
            // If the IP is reachable within the given timeout, consider it active.
            if (inAddress.isReachable(timeout)) {
                // Add result to the subnet scan results list.
                subnetScanResults.add(new SubnetScanResult(inAddress.getHostAddress()));
                // Update the table model on the EDT using invokeLater.
                invokeLater(() -> model.addRow(new Object[]{inAddress.getHostAddress()}));
            }
        } catch (IOException e) {
            // If an exception occurs (e.g., timeout), consider it inactive.
        }
    }
```

The original code from which I wrote the adaptation for *PingPal*—

```
package ipScanner;

import java.io.IOException;
import java.net.InetAddress;
import java.util.concurrent.ConcurrentSkipListSet;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicInteger;

public class IpScanner {

    public static void main(String[] args) {
        ConcurrentSkipListSet networkIps = scan("192.168.1.0", 254);
        System.out.println("Devices connected to the network:");
        networkIps.forEach(ip -> System.out.println(ip));
    }

    /**
     *
     * @param firstIpInTheNetwork e.g: 192.168.1.0
     * @param numOfIps e.g: 254
     * @return
     */
    public static ConcurrentSkipListSet scan(String firstIpInTheNetwork, int numOfIps) {
        ExecutorService executorService = Executors.newFixedThreadPool(20);
        final String networkId = firstIpInTheNetwork.substring(0,
firstIpInTheNetwork.length() - 1);
         ConcurrentSkipListSet ipsSet = new ConcurrentSkipListSet();

         AtomicInteger ips = new AtomicInteger(0);
         while (ips.get() <= numOfIps) {
             String ip = networkId + ips.getAndIncrement();
             executorService.submit(() -> {
                 try {
                     InetAddress inAddress = InetAddress.getByName(ip);
                     if (inAddress.isReachable(500)) {
                         System.out.println("found ip: " + ip);
                         ipsSet.add(ip);
                     }
                 }
                 catch (IOException e) {

                 }
             });
         }
         executorService.shutdown();
         try {
             executorService.awaitTermination(1, TimeUnit.MINUTES);
         }
         catch (InterruptedException e) {
             System.out.println(e.getMessage());
         }

         return ipsSet;
     }

 }
```

The code in the *PortScan* class in *PingPal* being referred to—

```
    /**
     * Starts the port scan.
     * <p>
     * This method clears any existing data from the UI components, then
     * iterates over the port range, submitting tasks to scan each port
     * concurrently. The progress bar is updated as ports are scanned.
     * </p>
     * <p>
     * The method executes until either all tasks complete, the timeout is
     * reached, or a stop is requested.
     * </p>
     */
    public void start() {
        // Clear current data being displayed in the UI table via EDT.
        invokeLater(() -> model.setRowCount(0));

        // Reset progress bar via EDT.
        invokeLater(() -> prgPortScan.setValue(0));

        // Atomic counter to generate port numbers from bottomRangePort to topRangePort.
        AtomicInteger ports = new AtomicInteger(bottomRangePort);
        // Counter to track how many ports have been scanned so far.
        AtomicInteger scannedPorts = new AtomicInteger(0);

        // Loop through each port in the range.
        while (ports.get() <= topRangePort) {
            // Generate the IP to scan.
            int port = ports.getAndIncrement();

            // Submit a task to the executor to scan the port.
            executorService.submit(() -> {
                // Scan the port
                scanPort(port);

                // Update the progress bar after scanning each port.
                updateProgressBar(scannedPorts.getAndIncrement());
            });
        }

        // Initiate shutdown and wait for tasks to finish, with a maximum wait time.
        executorService.shutdown();
        try {
            // Wait until all tasks have finished, or timeout after 10 minutes.
            if (!executorService.awaitTermination(10, TimeUnit.MINUTES)) {
                // If tasks are not finished in 10 minutes, force shutdown.
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            // If the thread is interrupted, reset the interrupt flag and exit the loop.
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
```

4.1.2 — Explanation of Critical Algorithms

PingPal has the following 5 main critical algorithms—

A. **Subnet Scan Loop—**

The purpose of this algorithm is to discover which IPv4 addresses in a specified network range respond to an ICMP-like reachability check. Its role is to produce the list of active IP addresses in a subnet, update the UI table with discovered hosts, and show progress to the user.

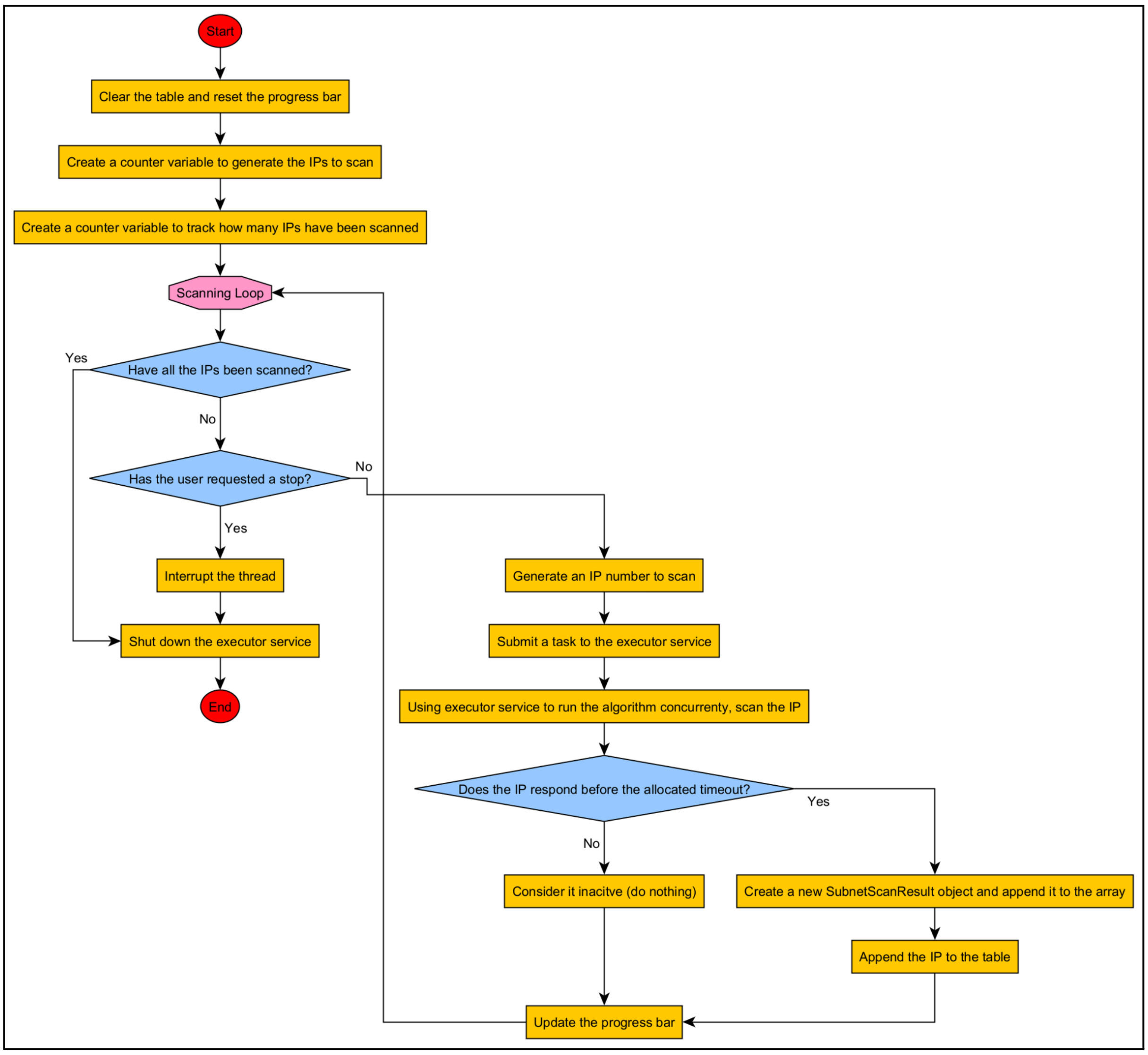
The algorithm is critical as it serves part of core *PingPal* functionality. Discovering live hosts on a subnet is a primary feature of *PingPal*. Therefore, without it, the app cannot perform subnet discovery or feed subsequent operations (e.g., targeted port scan, ping) with meaningful targets.

Additionally, the use of threads in this algorithm is critical. The algorithm must be fast and responsive (non-blocking UI) because network scans touch many addresses and can take significant time when done without threads.

Furthermore, many other program actions depend on accurate and complete results (import/export, UI state), so correctness and robustness—which are achieved by this specific algorithm—are essential.

Lastly, the algorithm ensures performance and scalability. The algorithm scales to large subnets while making efficient use of CPU and network resources.

The flow diagram explaining how the algorithm works is as follows—



Start
Clear the table and reset the progress bar
Create a counter variable to generate the IPs to scan
Create a counter variable to track how many IPs have been scanned
Scanning Loop
Have all the IPs been scanned?
Yes
No
Has the user requested a stop?
No
Yes
Interrupt the thread
Shut down the executor service
End
Generate an IP number to scan
Submit a task to the executor service
Using executor service to run the algorithm concurrently, scan the IP
Does the IP respond before the allocated timeout?
Yes
No
Consider it inacitve (do nothing)
Create a new SubnetScanResult object and append it to the array
Append the IP to the table
Update the progress bar

B. **Device Ping Loop—**

This algorithm repeatedly measures reachability and response latency for a configured IP address at a set interval, accumulates per-ping results, updates the UI tables, and computes summary statistics (min/max/avg round trip time and packet loss) for reporting and export.

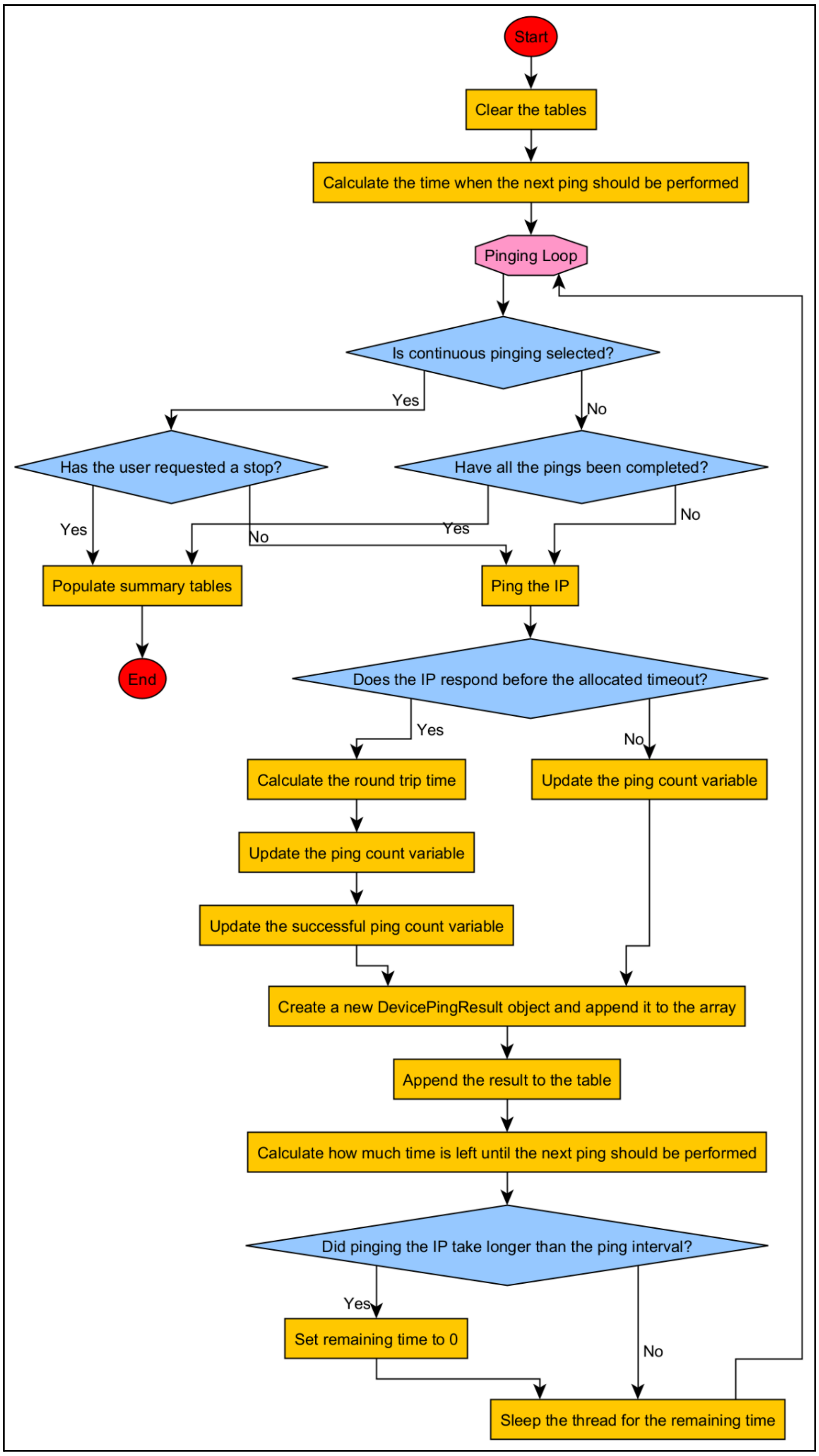
The algorithm is critical as it produces a primary measurement dataset. Any networking that the user aims to do that depends on live network state—the per-ping table, the summary statistics, and exports—uses the results this algorithm generates, so incorrect or missing results break functionality.

Additionally, the use of scheduling logic in this algorithm is critical. The scheduling logic keeps pings at precisely the configured interval, which is essential for meaningful latency comparisons and reliable time-based statistics.

Furthermore, this algorithm ensures robustness and continuity of measurement. By handling per-ping errors locally (treating them as failed pings) and restoring the interrupt flag correctly, the algorithm prevents a single network hiccup from aborting the whole session and supports user cancellation.

Lastly, it preserves UI correctness and usability. Results are appended in the background and UI updates are dispatched to the Event Dispatch Thread, which preserves Swing threading rules and guarantees the UI displays consistent, timely measurement updates for the user.

The flow diagram explaining how the algorithm works is as follows—


Start
Clear the tables
Calculate the time when the next ping should be performed
Pinging Loop
Is continuous pinging selected?
Yes
No
Has the user requested a stop?
Have all the pings been completed?
Yes
No
Yes
No
Populate summary tables
Ping the IP
End
Does the IP respond before the allocated timeout?
Yes
No
Calculate the round trip time
Update the ping count variable
Update the ping count variable
Update the successful ping count variable
Create a new DevicePingResult object and append it to the array
Append the result to the table
Calculate how much time is left until the next ping should be performed
Did pinging the IP take longer than the ping interval?
Yes
No
Set remaining time to 0
Sleep the thread for the remaining time

C. **Port Scan Loop—**

This algorithm concurrently attempts TCP connections across a configured port range on a target IP, records open ports and their associated protocols, updates the UI with per-port results and progress, and produces a result set for export or further processing.

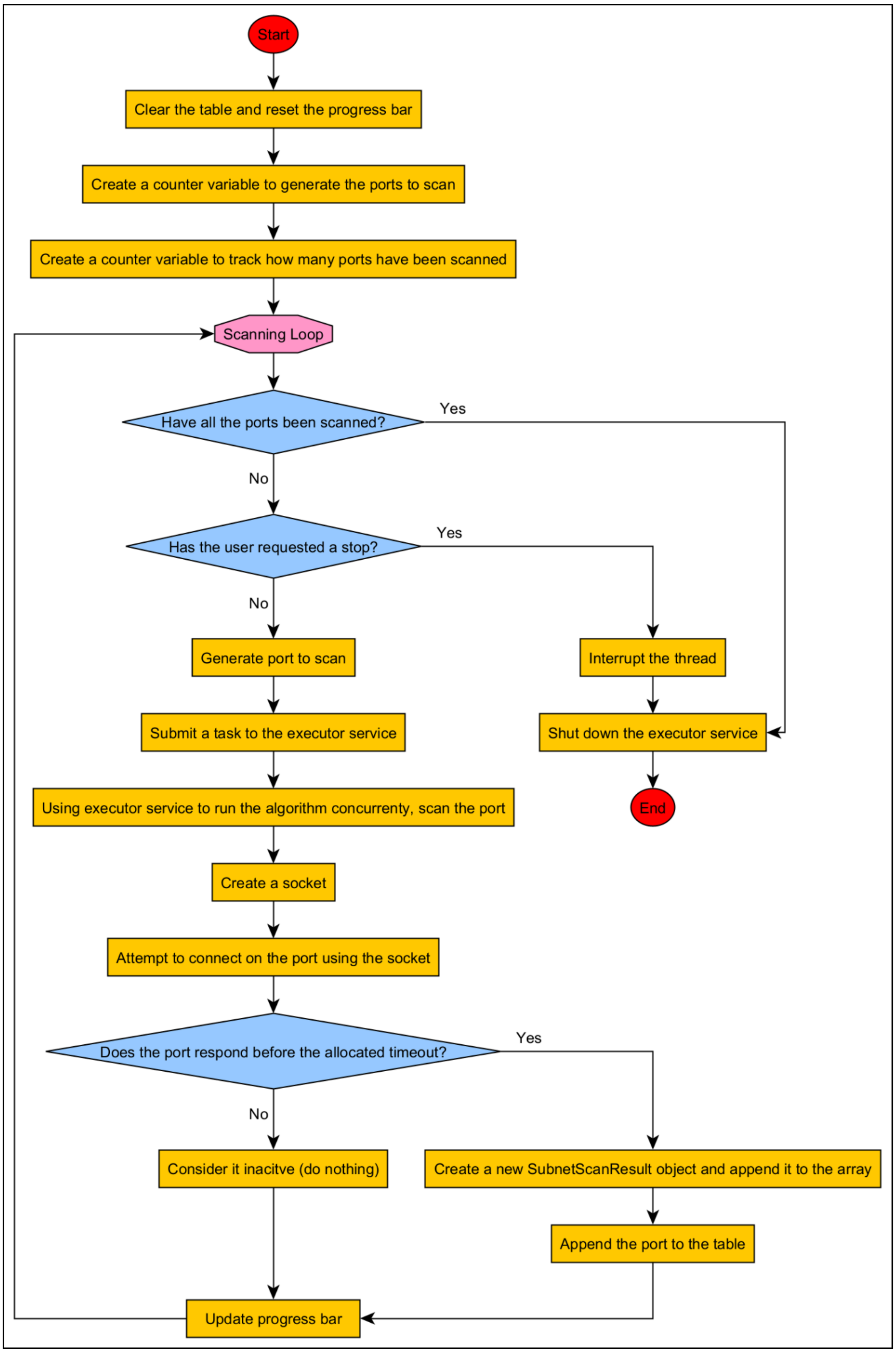
This algorithm is critical as it discovers live service endpoints. By attempting real TCP connections and recording successful ones, the algorithm produces the definitive list of open ports and associated protocols that the rest of the program reports, exports, and acts upon, so incorrect scanning would yield useless or misleading results.

Additionally, the use of threads in this algorithm is critical. The algorithm must be fast and responsive (non-blocking UI) because port scans touch many ports and can take significant time when done without threads.

Furthermore, this specific algorithm is crucial as it balances performance and resource use safely. By submitting independent tasks to an *ExecutorService* and then shutting it down cleanly, the algorithm achieves high throughput (many ports scanned in parallel) while still enforcing a bounded lifetime and managing thread termination to avoid runaway resource consumption.

Lastly, this algorithm ensures robustness and continuity of measurement. By handling network exceptions locally (treating them as closed ports) and restoring the interrupt flag correctly, the algorithm prevents a single network hiccup from aborting the whole session and supports user cancellation.

The flow diagram explaining how the algorithm works is as follows—



Start
Clear the table and reset the progress bar
Create a counter variable to generate the ports to scan
Create a counter variable to track how many ports have been scanned
Scanning Loop
Have all the ports been scanned?
Yes
No
Has the user requested a stop?
Yes
No
Generate port to scan
Interrupt the thread
Submit a task to the executor service
Shut down the executor service
Using executor service to run the algorithm concurrenty, scan the port
End
Create a socket
Attempt to connect on the port using the socket
Does the port respond before the allocated timeout?
Yes
No
Consider it inacitve (do nothing)
Create a new SubnetScanResult object and append it to the array
Append the port to the table
Update progress bar

D. **TCP Message Listen Loop—**

This algorithm enables real-time communication between *PingPal* and an external client also using *PingPal* over a TCP connection. It opens a server socket on a chosen port, waits for a client to connect, and then continuously receives, formats, and displays messages while also supporting sending messages back to the client. This allows the program to act as a live message listener and relay tool within the network.

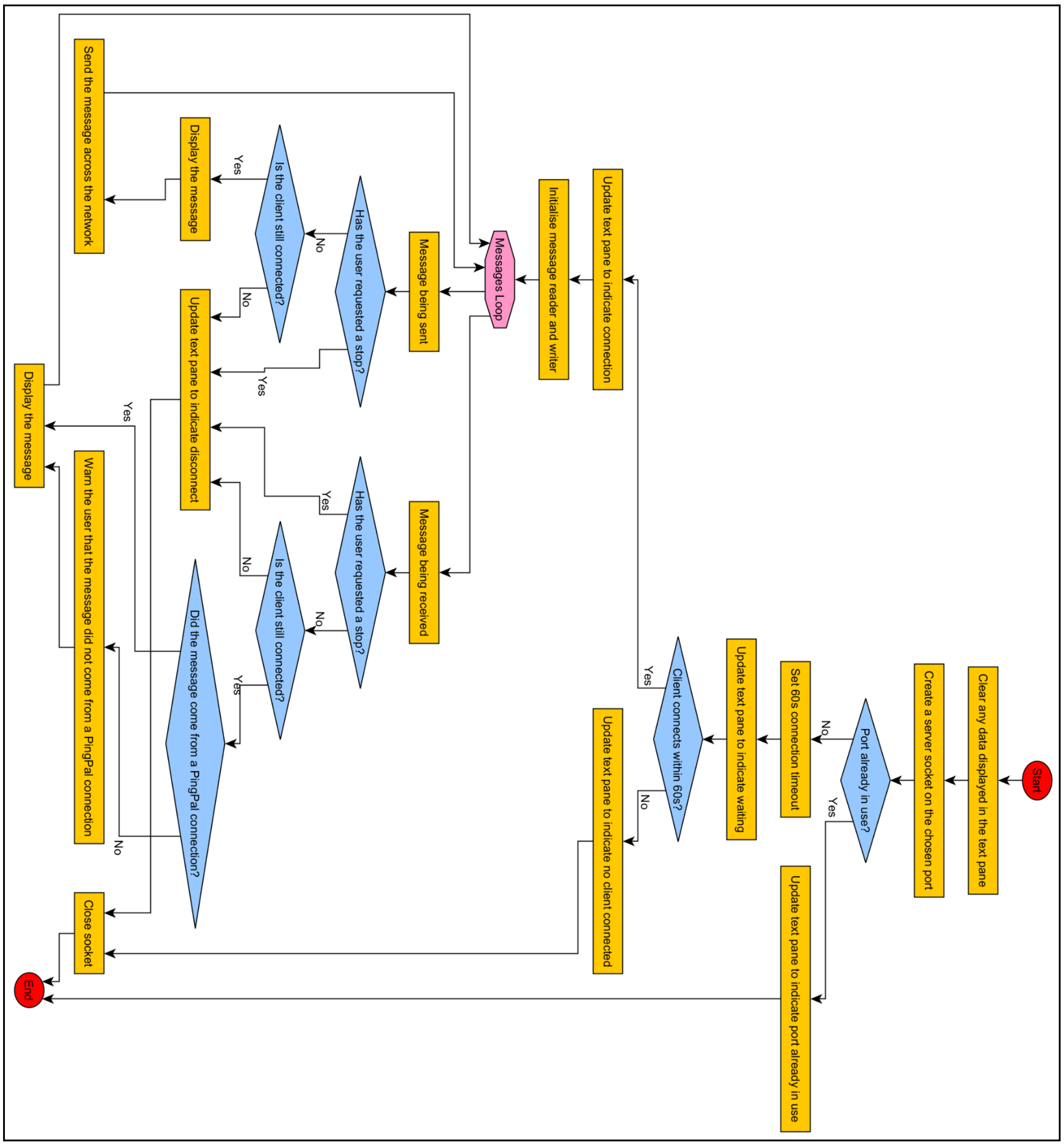
This algorithm is critical as it provides interactive communication functionality. Without this algorithm, *PingPal* would only perform scanning tasks. The message listener enables the program to have an interactive communication tool, enabling two-way TCP messaging that is essential for its full feature set.

Additionally, the use of threads in this algorithm is critical. The algorithm must be responsive (non-blocking UI) because, without it, the program would appear frozen whenever a connection is established.

Furthermore, this specific algorithm ensures proper connection handling and resilience. The algorithm includes timeout handling, error reporting, and clean shutdown when sockets cannot be established or are interrupted. This ensures the program remains robust, avoids freezing, and gives clear feedback when network conditions fail.

Lastly, this algorithm also ensures that messages are formatted and presented consistently. Incoming and outgoing messages are timestamped, tagged with hostnames, and styled in the UI, giving clarity to communication. This formatting step is essential when it comes to distinguishing messages from multiple devices and providing context for network communication.

The flow diagram explaining how the algorithm works is as follows—


Start
Clear any data displayed in the text pane
Create a server socket on the chosen port
Port already in use?
Yes
No
Update text pane to indicate port already in use
Set 60s connection timeout
Update text pane to indicate waiting
Client connects within 60s?
Yes
No
Update text pane to indicate no client connected
Update text pane to indicate connection
Initialise message reader and writer
Messages Loop
Message being sent
Has the user requested a stop?
Yes
No
Is the client still connected?
Yes
No
Display the message
Send the message across the network
Message being received
Has the user requested a stop?
Yes
No
Is the client still connected?
Yes
No
Update text pane to indicate disconnect
Did the message come from a PingPal connection?
Yes
No
Display the message
Warn the user that the message did not come from a PingPal connection
Close socket
End

E. **TCP Message Connect Loop—**

This algorithm enables PingPal to act as a client that connects to a server socket over TCP. It attempts to establish a connection to a specified IP address and port, then continuously exchanges messages with the server. Messages are formatted with timestamps and hostnames, displayed in the user interface, and relayed back to the connected server. This allows PingPal to join and participate in live TCP-based communication sessions.

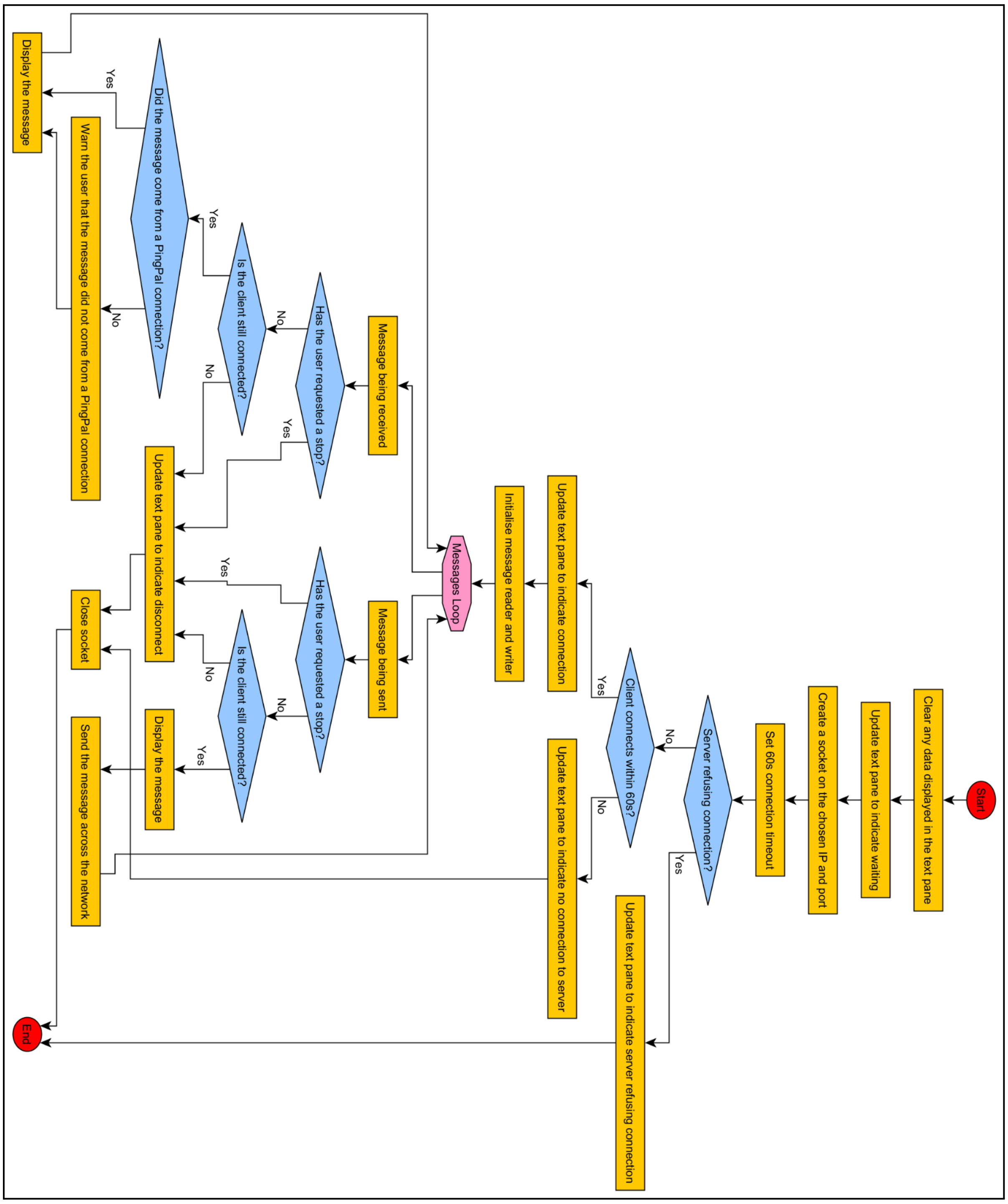
This algorithm is critical as it provides interactive communication functionality. Without this algorithm, *PingPal* would only perform scanning tasks. The message listener enables the program to have an interactive communication tool, enabling two-way TCP messaging that is essential for its full feature set.

Additionally, the use of threads in this algorithm is critical. The algorithm must be responsive (non-blocking UI) because, without it, the program would appear frozen whenever a connection is established.

Furthermore, this specific algorithm ensures proper connection handling and resilience. The algorithm includes timeout handling, error reporting, and clean shutdown when sockets cannot be established or are interrupted. This ensures the program remains robust, avoids freezing, and gives clear feedback when network conditions fail.

Lastly, this algorithm also ensures that messages are formatted and presented consistently. Incoming and outgoing messages are timestamped, tagged with hostnames, and styled in the UI, giving clarity to communication. This formatting step is essential when it comes to distinguishing messages from multiple devices and providing context for network communication.

The flow diagram explaining how the algorithm works is as follows—



Start
Clear any data displayed in the text pane
Update text pane to indicate waiting
Create a socket on the chosen IP and port
Set 60s connection timeout
Server refusing connection?
Yes
Update text pane to indicate server refusing connection
No
Client connects within 60s?
No
Update text pane to indicate no connection to server
Yes
Update text pane to indicate connection
Initialise message reader and writer
Messages Loop
Message being received
Has the user requested a stop?
Yes
No
Is the client still connected?
No
Yes
Did the message come from a PingPal connection?
Yes
Display the message
No
Warn the user that the message did not come from a PingPal connection
Message being sent
Has the user requested a stop?
Yes
No
Is the client still connected?
No
Yes
Display the message
Send the message across the network
Update text pane to indicate disconnect
Close socket
End

4.1.3 — Advanced Techniques

PingPal makes use of a variety of advanced techniques. However, given the limitation of 4 marks for this section, I have chosen to explain the following two advanced techniques—

A. **Networking**—

   Networking is a programming technique that enables applications to communicate across devices by transmitting and receiving data over a computer network. Networking requires an understanding of sockets, IP addressing, ports, and communication protocols—all of which are concepts outside the normal curriculum. In *PingPal*, networking is central to almost every major feature, and it is implemented in multiple ways to support different types of communication.

   Firstly, the subnet scanning program function relies on networking to systematically probe all possible IP addresses in a given subnet. The program generates valid IP addresses within a range and attempts to establish communication with each device using the Java *InetAddress* class and *isReachable()* method. If a device responds, it is marked as active and displayed to the user. This functionality demonstrates networking by directly interacting with network infrastructure, using low-level reachability checks to discover devices connected to the local network.

   Secondly, the device pinging functionality makes use of networking to send repeated echo requests to a specific IP address, measuring whether the target is reachable and calculating the round-trip time. This uses Java's networking libraries (*InetAddress*) to simulate the behaviour of the standard network utility "ping." By implementing this functionality, PingPal demonstrates how to measure network performance and device availability—tasks that require advanced understanding of packet transmission, timeouts, and host resolution.

Thirdly, port scanning is another networking-based feature. This feature checks whether specific ports on a device are open and listening for connections. This is implemented by creating *Socket* objects that attempt to connect to a target device's IP and port within a given timeout. Successful connections indicate that a service is running on that port, and *PingPal* retrieves the associated protocol from a lookup table. This demonstrates networking in a more advanced way because it involves not only IP addresses but also ports, which represent communication endpoints for different services.

Fourthly, the messaging functionality where *PingPal* acts as a server is another example of networking in *PingPal*. Acting as a server requires *PingPal* to create a *ServerSocket*, bind it to a port, and listen for incoming client connections. Once a client connects, the server continuously exchanges messages using input and output streams. This is advanced networking because it involves server-side socket programming, persistent connections, and message handling. The program must also deal with issues such as timeouts, binding errors (when a port is already in use), and client disconnections— all of which require networking to implement properly.

Lastly, the messaging functionality where *PingPal* acts as a client is another example of networking in *PingPal*. The client-side implementation of networking involves creating a *Socket* to connect to a server's IP address and port, then maintaining a live connection to send and receive messages. In *PingPal*, this allows one device to join a messaging session hosted by another. This demonstrates client-side socket programming, message formatting, and handling asynchronous communication. The implementation includes exception handling for common networking errors, such as server refusal, connection timeouts, and unexpected disconnections.

Overall, networking is a critical advanced technique in *PingPal* because it enables the core functionality of discovering devices, testing connectivity, scanning for services, and exchanging messages across a network. Without networking, PingPal would not serve its intended purpose as a communication and diagnostic tool.

The code related to the networking functionality in PingPal is found on the following pages—

Subnet Scan Networking Technique Code

Start Subnet Scan Method

```
    /**
     * Starts the subnet scan.
     * <p>
     * Clears the table and progress bar, then concurrently scans all IPs in the
     * subnet. Uses an ExecutorService to run scan tasks concurrently and
     * updates the progress bar.
     * </p>
     * <p>
     * The method executes until either all tasks complete, the timeout is
     * reached, or a stop is requested.
     * </p>
     */
    public void start() {
        // Clear current data in the table on the EDT.
        invokeLater(() -> model.setRowCount(0));
        // Reset the progress bar on the EDT.
        invokeLater(() -> prgSubnetScan.setValue(0));

        // Counter variable for IPs to scan.
        AtomicInteger ips = new AtomicInteger(0);
        // Counter variable for already scanned IPs.
        AtomicInteger scannedIps = new AtomicInteger(0);

        // Loop through each IP index in the range.
        while (ips.get() <= numOfIPs) {
            // Generate the IP to scan.
            String ip = generateIP(ips.getAndIncrement());

            // Submit a task to the executor to scan the IP.
            executorService.submit(() -> {
                InetAddress inAddress;
                try {
                    // Resolve the IP address.
                    inAddress = InetAddress.getByName(ip);
                    // Scan the IP.
                    scanIP(inAddress);

                    // Update the progress bar after scanning each IP.
                    updateProgressBar(scannedIps.getAndIncrement());
                } catch (UnknownHostException e) {
                    // If IP is unknown, ignore and continue.
                }
            });
        }

        // Initiate shutdown and wait for tasks to finish, with a maximum wait time.
        executorService.shutdown();
        try {
            // Wait until all tasks have finished, or timeout after 10 minutes.
            if (!executorService.awaitTermination(10, TimeUnit.MINUTES)) {
                // If tasks are not finished in 10 minutes, force shutdown.
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            // If the thread is interrupted, reset the interrupt flag and exit the loop.
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
```

Scan IP Method

```
    /**
     * Attempts to ping the single given InetAddress. If reachable, adds the IP
     * address to the results list and updates the table model.
     *
     * @param inAddress the InetAddress representing the IP to scan
     */
    private void scanIP(InetAddress inAddress) {
        try {
            // If the IP is reachable within the given timeout, consider it active.
            if (inAddress.isReachable(timeout)) {
                // Add result to the subnet scan results list.
                subnetScanResults.add(new SubnetScanResult(inAddress.getHostAddress()));
                // Update the table model on the EDT using invokeLater.
                invokeLater(() -> model.addRow(new Object[]{inAddress.getHostAddress()}));
            }
        } catch (IOException e) {
            // If an exception occurs (e.g., timeout), consider it inactive.
        }
    }
```

Device Ping Networking Technique Code

Start Device Ping Method

```
    /**
     * Starts the pinging process. This method clears the UI tables, then
     * repeatedly pings the target IP address according to the ping interval,
     * until the specified number of pings has been reached (or indefinitely if
     * continuous). After pinging is complete, it populates summary result
     * tables.
     * <p>
     * Note: This method runs on the calling thread. It is expected that you run
     * it in a background thread to avoid blocking the UI.
     * </p>
     */
    public void start() {
        // Clear current data in the result tables on the EDT
        invokeLater(() -> {
            devicePingTableModel.setRowCount(0);
            devicePingResponseResultsTableModel.setRowCount(0);
            devicePingPacketResultsTableModel.setRowCount(0);
        });

        // Calculate the time (in nanoseconds) when the next ping should be performed.
        double nextPingTime = System.nanoTime() + (pingInterval * 1_000_000);

        // Loop until the program reaches the specified number of pings or continuous mode,
and no stop has been requested.
        while ((pingCount < numOfPings || continuousPinging) && !stopRequested) {
            // Call the ping IP method
            pingIP();

            try {
                // Calculate remaining time (in milliseconds) before the next ping.
                double remainingTime = (nextPingTime - System.nanoTime()) / 1_000_000;

                // If pinging the IP took longer than the ping interval, reset the remaining
time to 0 to ensure no negative time.
                if (remainingTime < 0) {
                    remainingTime = 0;
                }

                // Sleep until it's time for the next ping.
                Thread.sleep((long) remainingTime);

                // Update the time when the next ping should be performed.
                nextPingTime += (pingInterval * 1_000_000);

            } catch (InterruptedException e) {
                // If the thread is interrupted, reset the interrupt flag and exit the loop.
                Thread.currentThread().interrupt();
            }
        }

        // After pinging, update the UI with the summary results.
        populateResultsTables();
    }
```

Ping IP Method

```
    /**
     * Performs a single ping to the target IP address.
     * <p>
     * This method attempts to ping the specified IP address using
     * {@code InetAddress.isReachable()}. If the IP is reachable, it records the
     * round-trip time and considers the ping successful. If not, it records a
     * failed ping with a round-trip time equal to the ping interval. The result
     * is stored in the {@code devicePingResults} list, and the UI table is
     * updated.
     * </p>
     */
    private void pingIP() {
        try {
            // Resolve the IP address.
            InetAddress inAddress = InetAddress.getByName(ipAddress);

            // Record the time before pinging.
            long timeBeforePing = System.nanoTime();

            // Check if the IP is reachable within the ping interval.
            if (inAddress.isReachable(pingInterval)) {
                // Calculate round-trip time in milliseconds.
                int roundTripTime = (int) (System.nanoTime() - timeBeforePing) / 1_000_000;

                // Clamp roundTripTime to the pingInterval if necessary.
                roundTripTime = Math.min(roundTripTime, pingInterval);

                // Update counter variables
                pingCount++;
                successfulPings++;

                // Add a new successful ping result.
                devicePingResults.add(new DevicePingResult(roundTripTime, true,
getPacketLoss()));

                // Update the UI table with the successful ping result.
                invokeLater(() -> {
                    devicePingTableModel.addRow(new Object[]{
                        ipAddress,
                        devicePingResults.getLast().getRoundTripTime(),
                        true,
                        devicePingResults.getLast().getPacketLoss()
                    });
                });

                // Ping failed.
            } else {
                // Update counter variables.
                pingCount++;

                // Add a new unsuccessful ping result.
                devicePingResults.add(new DevicePingResult(pingInterval, false,
getPacketLoss()));

                // Update the UI table with the failed ping result.
                invokeLater(() -> {
                    devicePingTableModel.addRow(new Object[]{
                        ipAddress,
                        pingInterval,
                        false,
                        devicePingResults.getLast().getPacketLoss()
                    });
                });

            }
        } catch (IOException e) {
            // In case of an I/O error, the exception is ignored as it does not occurr given 
the IP address is guaranteed to be in correct format.
        }
    }
```

Port Scan Networking Technique Code

Start Port Scan Method

```
    /**
     * Starts the port scan.
     * <p>
     * This method clears any existing data from the UI components, then
     * iterates over the port range, submitting tasks to scan each port
     * concurrently. The progress bar is updated as ports are scanned.
     * </p>
     * <p>
     * The method executes until either all tasks complete, the timeout is
     * reached, or a stop is requested.
     * </p>
     */
    public void start() {
        // Clear current data being displayed in the UI table via EDT.
        invokeLater(() -> model.setRowCount(0));

        // Reset progress bar via EDT.
        invokeLater(() -> prgPortScan.setValue(0));

        // Atomic counter to generate port numbers from bottomRangePort to topRangePort.
        AtomicInteger ports = new AtomicInteger(bottomRangePort);
        // Counter to track how many ports have been scanned so far.
        AtomicInteger scannedPorts = new AtomicInteger(0);

        // Loop through each port in the range.
        while (ports.get() <= topRangePort) {
            // Generate the port to scan.
            int port = ports.getAndIncrement();

            // Submit a task to the executor to scan the port.
            executorService.submit(() -> {
                // Scan the port.
                scanPort(port);

                // Update the progress bar after scanning each port.
                updateProgressBar(scannedPorts.getAndIncrement());
            });
        }

        // Initiate shutdown and wait for tasks to finish, with a maximum wait time.
        executorService.shutdown();
        try {
            // Wait until all tasks have finished, or timeout after 10 minutes.
            if (!executorService.awaitTermination(10, TimeUnit.MINUTES)) {
                // If tasks are not finished in 10 minutes, force shutdown.
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            // If the thread is interrupted, reset the interrupt flag and exit the loop.
            executorService.shutdownNow();
            Thread.currentThread().interrupt();
        }
    }
```

Scan Port Method

```
    /**
     * Attempts to scan a single port on the target IP address.
     * <p>
     * The method creates a new socket, attempts to connect to the given port
     * with the specified timeout, and then closes the socket. If the connection
     * is successful, it retrieves the associated protocol and updates the scan
     * results list and the UI table.
     * </p>
     *
     * @param port the port number to scan
     */
    public void scanPort(int port) {
        try {
            // Create new socket object.
            Socket socket = new Socket();
            // Attempt to connect within a given timeout.
            socket.connect(new InetSocketAddress(ipAddress, port), timeout);
            socket.close();

            // Retrieve protocol for the given port.
            String protocol = Protocols.getProtocolForPort(port);

            // Record the scan result in the port scan results list.
            portScanResults.add(new PortScanResult(port, protocol));

            // Update UI results table on the EDT
            invokeLater(() -> model.addRow(new Object[]{port, protocol}));

        } catch (IOException e) {
             // Exception is ignored if the port is not reachable, and the port is considered
closed.
        }
    }
```

TCP Message Listen Networking Technique Code

Start TCP Message Listen Method

```
    /**
     * Starts the server socket, waits (up to 60s) for a client, and then
     * continuously receives and displays messages.
     * <p>
     * If no client connects within 60 seconds, a timeout occurs, the socket is
     * closed, and an error message is displayed.
     * </p>
     */
    public void start() {
        try {
            // Clear any data that is currently displayed in the text pane.
            doc.remove(0, doc.getLength());

            // Initialise the server socket, and begin listening on the specified port.
            serverSocket = new ServerSocket(port);
            // Set 60000ms (60s) accept timeout.
            serverSocket.setSoTimeout(60_000);
            // Write to the text pane to indicate the program is waitng for a client
connection.
            updateTextPane("Waiting for client connection on " +
InetAddress.getLocalHost().getHostAddress() + ":" + port + ".\n", messageStyle);

            // Accept device to connect and initialise the client socket.
            clientSocket = serverSocket.accept();

            // Get and format client IP address.
            String clientIP = clientSocket.getInetAddress().getHostAddress();
            // Write to the text pane to indicate a client has connected.
            updateTextPane("Client connected: " + clientIP + "\n", successStyle);

            // Initialise the persistent reader and writer.
            in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
            out = new PrintWriter(clientSocket.getOutputStream(), true);

            // Enter receive loop.
            receiveMessages();

            // Throw an error if no clinet connected within 60s.
        } catch (SocketTimeoutException e) {
            // Stop the program.
            requestStop();
            // Write to the text pane to indicate no client connected within 60s.
            updateTextPane("No client connected within 60 seconds. Closing server socket.\n",
errorStyle);

            // Throw an error if the specified port is already open and being used.
        } catch (BindException e) {
            // Write to the text pane to indicate that the program is already used.
            updateTextPane("Port alread in use.\n", errorStyle);

            // General I/O error, but its most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the connection closed.
            updateTextPane("Connection closed.\n", errorStyle);

            // Catch exceptions that may occur when trying to append a message to the text
pane.
        } catch (BadLocationException ex) {
            // Display a JOptionPane to indicate such.
            JOptionPane.showMessageDialog(txpTCPMessageListen.getParent(), "Error occurred
during text pane update process.", "Text Pane Write Error", JOptionPane.ERROR_MESSAGE);
        }
    }
```

Send Message Method

```
    /**
     * Sends a message to the connected client and echoes it locally with
     * timestamp and hostname formatting.
     *
     * @param message the text to send
     */
    public void sendMessage(String message) {
        try {
            // Get and format the current date and time.
            LocalDateTime now = LocalDateTime.now();
            String formattedDateTime = now.format(DateTimeFormatter.ofPattern("dd-MM-yy
HH:mm:ss"));

            // Get the host name of the local device.
            String hostname = InetAddress.getLocalHost().getHostName();

            // Write the formatted message to the text pane.
            updateTextPane(formattedDateTime + " ", dateTimeStyle);
            updateTextPane("[" + hostname + "] ", hostnameStyle);
            updateTextPane("> " + message + "\n", messageStyle);

            // Print the message for the connected device.
            out.println(formattedDateTime + " [" + hostname + "] > " + message);

            // General I/O error, but its most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the client disconnected.
            updateTextPane("Client disconnected.", errorStyle);
        }

    }
```

Receive Message Method

```
    /**
     * Continuously receives incoming messages from the client, formats them,
     * and appends them to the text pane.
     */
    private void receiveMessages() {
        // Loop until the client disconnects or the user chooses to stop listening for
messages and disconnect themself.
        try {
            while (!stopRequested) {
                // Declare the variable to hold the message the client sends.
                String message;
                // Loop to keep checking whether the client has sent a message.
                while ((message = in.readLine()) != null) {
                    // Check whether the message came via a PingPal connection, by checking
format.
                    if (message.contains("[") && message.contains("]")
                            && message.contains(">")) {
                        // Write the formatted message to the text pane.
                        updateTextPane(message.substring(0,
                                message.indexOf("[")), dateTimeStyle);
                        updateTextPane(message.substring(message.indexOf("["),
                                message.indexOf(">")), hostnameStyle);
                        updateTextPane(message.substring(message.indexOf(">"))
                                + "\n", messageStyle);
                    } else {
                        // Indicate that the message does not come from a PingPal connection,
however, still display it.
                        updateTextPane("Not connected to a device via PingPal. However, the
message reads:\n", errorStyle);
                        updateTextPane(message + "\n", messageStyle);
                    }
                }
            }

            // General I/O error, but it's most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the client disconnected.
            updateTextPane("Client disconnected.", errorStyle);
        }
    }
```

Request Stop Method

```
    /**
     * Requests that the pinging process stop.
     * <p>
     * This sets the {@code stopRequested} flag to true, so that the ping loop
     * in {@code start()} will terminate early, updates the text pane to
     * indicate the connection is being closed, and close both the client socket
     * and server socket.
     * </p>
     */
    public void requestStop() {
        // Set the stopRequested flag to true.
        stopRequested = true;

        // Write to the text pane to indicate that the sockets are being closed.
        updateTextPane("Exiting TCP Message and closing sockets.\n", errorStyle);
        try {
            // Close the client socket.
            if (clientSocket != null && !clientSocket.isClosed()) {
                clientSocket.close();
            }
            // Close the server socket.
            if (serverSocket != null && !serverSocket.isClosed()) {
                serverSocket.close();
            }

            // General I/O error, but it's most common use case is when the sockets are
already closed.
        } catch (IOException e) {
            updateTextPane("Sockets closed already.\n", errorStyle);
        }
    }
```

Is Device Connected Method

```
    /**
     * Checks whether a client is currently connected.
     *
     * @return {@code true} if the client socket is connected; otherwise
     * {@code false}
     */
    public boolean isDeviceConnected() {
        return clientSocket != null && clientSocket.isConnected();
    }
```

TCP Message Connect Networking Technique Code

Start TCP Message Connect Method

```
    /**
     * Starts the client socket, attempts to connect to a server socket, and
     * then continuously receives and displays messages.
     * <p>
     * If no client connects within 60 seconds, a timeout occurs, the socket is
     * closed, and an error message is displayed.
     * </p>
     */
    public void start() {
        try {
            // Clear any data that is currently displayed in the text pane.
            doc.remove(0, doc.getLength());

            // Write to the text pane to indicate the program is trying to establish a
connection.
            updateTextPane("Trying to establish a connection to " + ipAddress + ":" + port +
".\n", messageStyle);

            // Attempt to connect to a device on the specified IP and port.
            socket = new Socket(ipAddress, port);
            // Set 60000ms (60s) accept timeout.
            socket.setSoTimeout(60_000);
            // Write to the text pane to indicate that connection to server is successful.
            updateTextPane("Connected to chat server at " + ipAddress + ":" + port + ".\n",
successStyle);

            // Initialize the persistent reader and writer.
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);

            // Enter receive loop.
            receiveMessages();

            // Throw an error if the server refuses connection.
        } catch (ConnectException e) {
            // Stop the program.
            requestStop();
            // Write to the pane that the server has refused the connection request.
            updateTextPane("Server refused connection. Closing socket.\n", errorStyle);

            // Throw an error if no clinet connected within 60s.
        } catch (SocketTimeoutException e) {
            // Stop the program.
            requestStop();
            // Write to the text pane to indicate no client connected within 60s.
            updateTextPane("No client connected within 60 seconds. Closing socket.\n",
errorStyle);

            // General I/O error, but its most common use case is when the client
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the connection closed.
            updateTextPane("Connection closed.\n", errorStyle);
            // Catch exceptions that may occur when trying to append a message to the text
pane.
        } catch (BadLocationException ex) {
            // Display a JOptionPane to indicate such.
            JOptionPane.showMessageDialog(txpTCPMessageConnect.getParent(), "Error occurred
during text pane update process.", "Text Pane Write Error", JOptionPane.ERROR_MESSAGE);
        }
    }
```

Send Message Method

```
    /**
     * Sends a message to the connected client and echoes it locally with
     * timestamp and hostname formatting.
     *
     * @param message the text to send
     */
    public void sendMessage(String message) {
        try {
            // Get and format the current date and time.
            LocalDateTime now = LocalDateTime.now();
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yy HH:mm:ss");
            String formattedDateTime = now.format(formatter);

            // Get the host name of the local device.
            String hostname = InetAddress.getLocalHost().getHostName();

            // Write the formatted message to the text pane.
            updateTextPane(formattedDateTime + " ", dateTimeStyle);
            updateTextPane("[" + hostname + "] ", hostnameStyle);
            updateTextPane("> " + message + "\n", messageStyle);

            // Print the message for the connected device.
            out.println(formattedDateTime + " [" + hostname + "] > " + message);

            // General I/O error, but its most common use case is when the server
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the server disconnected.
            updateTextPane("Server disconnected.\n", errorStyle);
        }
    }
```

Receive Messages Method

```
/**
     * Continuously receives incoming messages from the client, formats them,
     * and appends them to the text pane.
     */
    private void receiveMessages() {
        try {
            // Loop until the server disconnects or the user chooses to stop listening for
messages and disconnect themself.
            while (!stopRequested) {
                // Declare the variable to hold the message the server sends.
                String message;
                // Loop to keep checking whether the server has sent a message.
                while ((message = in.readLine()) != null) {
                    // Check whether the message came via a PingPal connection, by checking
format.
                    if (message.contains("[") && message.contains("]")
                            && message.contains(">")) {
                        // Write the formatted message to the text pane.
                        updateTextPane(message.substring(0,
                                message.indexOf("[")), dateTimeStyle);
                        updateTextPane(message.substring(message.indexOf("["),
                                message.indexOf(">")), hostnameStyle);
                        updateTextPane(message.substring(message.indexOf(">"))
                                + "\n", messageStyle);
                    } else {
                        // Indicate that the message does not come from a PingPal connection,
however, still display it.
                        updateTextPane("Not connected to a device via PingPal. However, the
message reads:\n", errorStyle);
                        updateTextPane(message + "\n", messageStyle);
                    }
                }
            }

            // General I/O error, but it's most common use case is when the server
disconnects.
        } catch (IOException e) {
            // Write to the text pane to indicate the server disconnected.
            updateTextPane("Server disconnected.\n", errorStyle);
        }
    }
```

<u>Request Stop Method</u>

```
    /**
     * Requests that the pinging process stop.
     * <p>
     * This sets the {@code stopRequested} flag to true, so that the ping loop
     * in {@code start()} will terminate early, updates the text pane to
     * indicate the connection is being closed, and close both the client socket
     * and server socket.
     * </p>
     */
    public void requestStop() {
        // Set the stopRequested flag to true.
        stopRequested = true;

        // Write to the text pane to indicate that the sockets are being closed.
        updateTextPane("Exiting TCP Message and closing socket.\n", errorStyle);
        try {
            // Close the socket.
            if (socket != null && !socket.isClosed()) {
                socket.close();
            }

            // General I/O error, but it's most common use case is when the socket is already
closed.
        } catch (IOException e) {
            updateTextPane("Socket closed already.\n", errorStyle);
        }
    }
```

Is Device Connected Method

```
    /**
     * Checks whether a client is currently connected.
     *
     * @return {@code true} if the client socket is connected; otherwise
     * {@code false}
     */
    public boolean isDeviceConnected() {
        return socket != null && socket.isConnected();
    }
```

B. **Reading or writing to JSON Files with a complex data structure—e.g., an object with a field that is an array, or an array of objects or any combination—**

JSON (JavaScript Object Notation) is a widely used text-based format for storing and exchanging structured data. Working with JSON requires an understanding of hierarchical data structures, arrays, key-value mappings, and the use of libraries to parse and generate JSON data. In PingPal, JSON is a central technique for persisting results and enabling users to reimport data for later use. This goes beyond basic file input/output by allowing entire objects, arrays of objects, and nested fields to be serialised and reconstructed.

Firstly, *PingPal* allows the user to export scan and diagnostic results into JSON files. For example, after a subnet scan, device ping, and/or a port scan, the program gathers a collection of result objects (such as *SubnetScanResult*, *DevicePingResult*, and *PortScanResult*) into an array. Each object contains multiple fields (e.g., IP address, port number, protocol, round-trip time, etc.). These collections are then serialised into JSON format using the *org.json.** library. This results in a structured JSON file where arrays of objects capture the complete results in a format that is both human-readable and machine-parsable. This is more complex than writing to plain text, because the program must correctly map Java objects and their fields to JSON structures.

Secondly, *PingPal* supports importing previously exported JSON results back into the program. This involves reading a JSON file, parsing its contents, and reconstructing Java objects from the stored data. Arrays of JSON objects are initialised back into corresponding Java objects, ensuring that the results tables in the user interface are repopulated with all relevant details (IP addresses, ports, protocols, etc.). This is advanced because it requires deserialising complex nested data structures, handling arrays, and ensuring data integrity between the external file and the in-memory program structures.

Overall, JSON is essential in *PingPal* because it allows the program to preserve scan results across sessions, while supporting reusability and sharing of structured data. This ability to both export and import complex, hierarchical data makes JSON handling a clear example of an advanced programming technique.

The code of the classes related to the reading or writing to JSON files functionality in PingPal is found on the following pages—

Export Results JSON File Technique Code

<u>Write To JSON File Method</u>

```
    /**
     * Writes a {@link JSONObject} to a .json file in the chosen directory,
     * showing a success or error dialogue upon completion.
     *
     * @param output the JSON object to write
     */
    private void writeToJSONFile(JSONObject output) {
        // Construct full file path with .json extension.
        // Try-with-resources to ensure FileWriter is closed.
        try (FileWriter file = new FileWriter(exportResultsPath + "\\" + fileName + ".json"))
{
            // Use toString(4) for pretty printing with 4 space indentation.
            file.write(output.toString(4));

            // If successful, display a success message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Successfully exported results to \"" +
fileName + ".json\"", "Successful Export", JOptionPane.INFORMATION_MESSAGE);

            // General I/O error.
        } catch (IOException e) {
            // If unsuccessful, display an error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Error occured during file writing
process.", "File Writing Error", JOptionPane.ERROR_MESSAGE);
        }
    }
```

<u>Export Subnet Scan Results Method</u>

```
    /**
     * Exports the results of a {@link SubnetScan} to JSON.
     * <p>
     * This overload uses {@link SubnetScan} as the scan type.
     * </p>
     *
     * @param subnetScan the scan whose results to export
     */
    public void exportResults(SubnetScan subnetScan) {
        // Guard against a blank subnet scan, i.e. if no scan has been performed.
        if (subnetScan == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no subnet scan has
been performed.", "Null Subnet Scan Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Create a blank JSON array to hold all subnet scan results.
        JSONArray resultsArray = new JSONArray();

        // Loop through each subnet scan result.
        for (SubnetScanResult result : subnetScan.getSubnetScanResults()) {
            // Create a blank JSON object to hold the individual subnet scan results.
            JSONObject resultObj = new JSONObject();

            // Append the data from the subnet scan result to the JSON object.
            resultObj.put("ipAddress", result.getIPAddress());

            // Append the JSON object to the JSON array.
            resultsArray.put(resultObj);
        }

        // Create a top-level JSON object.
        JSONObject output = new JSONObject();

        // Wrap the metadata and array in the top-level JSON object.
        output.put("networkRange", subnetScan.getNetworkRange());
        output.put("timeout", subnetScan.getTimeout());
        output.put("subnetScanResults", resultsArray);

        // Write the data to the file.
        writeToJSONFile(output);
    }
```

Export Device Ping Results Method

```
    /**
     * Exports the results of a {@link DevicePing} to JSON.
     * <p>
     * This overload uses {@link DevicePing} as the scan type.
     * </p>
     *
     * @param devicePing the ping sessions whose results to export
     */
    public void exportResults(DevicePing devicePing) {
        // Guard against a blank device ping, i.e. if no ping has been performed.
        if (devicePing == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no device ping has
been performed.", "Null Device Ping Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Create a blank JSON array to hold all device ping results.
        JSONArray resultsArray = new JSONArray();

        // Loop through each device ping result.
        for (DevicePingResult result : devicePing.getDevicePingResults()) {
            // Create a blank JSON object to hold the individual device ping results.
            JSONObject resultObj = new JSONObject();

            // Append the data from the subnet scan result to the JSON object.
            resultObj.put("roundTripTime", result.getRoundTripTime());
            resultObj.put("successfulPing", result.isSuccessfulPing());
            resultObj.put("packetLoss", result.getPacketLoss());

            // Append the JSON object to the JSON array.
            resultsArray.put(resultObj);
        }

        // Create a top-level JSON object.
        JSONObject output = new JSONObject();

        // Wrap the metadata and array in the top-level JSON object.
        output.put("ipAddress", devicePing.getIpAddress());
        output.put("pingInterval", devicePing.getPingInterval());
        output.put("numOfPings", devicePing.getNumOfPings());
        output.put("continuousPinging", devicePing.isContinuousPinging());
        output.put("devicePingResults", resultsArray);

        // Write the data to the file.
        writeToJSONFile(output);
    }
```

Export Port Scan Results Method

```
    /**
     * Exports the results of a {@link PortScan} to JSON.
     * <p>
     * This overload uses {@link PortScan} as the scan type.
     * </p>
     *
     * @param portScan the port scan whose results to export
     */
    public void exportResults(PortScan portScan) {
        // Guard against a blank port scan, i.e. if no scan has been performed.
        if (portScan == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Cannot export results as no port scan has
been performed.", "Null Port Scan Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Guard against a blank file path, i.e. if the user has not selected a path.
        if (exportResultsPath == null) {
            // Display corresponding error message in a message dialogue.
            JOptionPane.showMessageDialog(panel, "Results not exported as no directory was
selected.", "File Path Error", JOptionPane.ERROR_MESSAGE);
            return;
        }

        // Create a JSON array to hold all port scan results.
        JSONArray resultsArray = new JSONArray();
        // Loop through each port scan result.
        for (PortScanResult result : portScan.getPortScanResults()) {
            // Create a blank JSON object to hold the individual port scan results.
            JSONObject resultObj = new JSONObject();

            // Append the data from the port scan result to the JSON object.
            resultObj.put("portNumber", result.getPortNumber());
            resultObj.put("protocol", result.getProtocol());

            // Append the JSON object to the JSON array.
            resultsArray.put(resultObj);
        }

        // Create a top-level JSON object.
        JSONObject output = new JSONObject();

        // Wrap the metadata and array in the top-level JSON object.
        output.put("ipAddress", portScan.getIpAddress());
        output.put("bottomRangePort", portScan.getBottomRangePort());
        output.put("topRangePort", portScan.getTopRangePort());
        output.put("timeout", portScan.getTimeout());
        output.put("portScanResults", resultsArray);

        // Write the data to the file.
        writeToJSONFile(output);
    }
```

Import Results JSON File Technique Code

Set Import Results Path Method

```
    /**
     * Opens a file chooser dialogue for the user to select a JSON file.
     * <p>
     * The file chooser is configured to only accept files with a ".json"
     * extension.
     * </p>
     */
    public void setImportResultsPath() {
        // Initialise the file chooser.
        fchFileChooser = new JFileChooser();

        // Disallow the selection of all files.
        fchFileChooser.setAcceptAllFileFilterUsed(false);
        // Restrict the accepted file type to only JSON files.
        fchFileChooser.setFileFilter(new FileNameExtensionFilter("JSON FILES", "json",
"json"));

        // Set the title of the file chooser.
        fchFileChooser.setDialogTitle("Select a JSON file");

        // If a file is selected, initialise the results file variable.
        int returnVal = fchFileChooser.showOpenDialog(panel);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            importResultsFile = fchFileChooser.getSelectedFile();
        }
    }
```

Read File Data Method

```
    /**
     * Reads and parses JSON data from the selected file.
     *
     * @return a JSONObject representing the contents of the file
     * @throws FileNotFoundException if the file does not exist
     * @throws JSONException if an error occurs during JSON parsing
     */
    private JSONObject readFileData() throws FileNotFoundException, JSONException {
        return new JSONObject(new JSONTokener(new FileReader(importResultsFile)));
    }
```

Parse Subnet Scan Array Method

```
    /**
     * Parses subnet scan results from the JSON array and converts them into a
     * list of {@code SubnetScanResult} objects.
     *
     * @return an {@code ArrayList} of {@code SubnetScanResult} objects parsed
     * from the JSON data
     */
    private ArrayList<SubnetScanResult> parseSubnetScanResultsArray() {
        // Create the ArrayList of SubnetScanResult objects.
        ArrayList<SubnetScanResult> subnetScanResults = new ArrayList<>();
        // Parse the data from the file to a JSONArray.
        JSONArray jsonSubnetScanResults = fileData.getJSONArray("subnetScanResults");

        // Loop through each JSONObject in the JSONArray.
        for (int i = 0; i < jsonSubnetScanResults.length(); i++) {
            JSONObject jsonSubnetScanResult = jsonSubnetScanResults.getJSONObject(i);

            // Parse the individual data values from the JSONObject, and use them to create a
new SubnetScanResult object.
            SubnetScanResult subnetScanResult = new SubnetScanResult(
                    jsonSubnetScanResult.getString("ipAddress")
            );
            // Append the SubnetScanResult to the list.
            subnetScanResults.add(subnetScanResult);
        }

        return subnetScanResults;
    }
```

Import Subnet Scan Data Method

```
    /**
     * Imports subnet scan results by extracting the network range, timeout, and
     * a list of subnet scan results from the JSON data, then passes the data to
     * the listener.
     */
    private void importSubnetScanData() {
        // Create the temporary holder variables.
        String networkRange = fileData.getString("networkRange");
        int timeout = fileData.getInt("timeout");
        ArrayList<SubnetScanResult> subnetScanResults = parseSubnetScanResultsArray();

        // Call the listener to indicate that a subnet scan has been imported.
        listener.onSubnetScanResultsImported(networkRange, timeout, subnetScanResults);
    }
```

Parse Device Ping Array Method

```
    /**
     * Parses device ping results from the JSON array and converts them into a
     * list of {@code DevicePingResult} objects.
     *
     * @return an {@code ArrayList} of {@code DevicePingResult} objects parsed
     * from the JSON data
     */
    private ArrayList<DevicePingResult> parseDevicePingResults() {
        // Create the ArrayList of DevicePingResult objects.
        ArrayList<DevicePingResult> devicePingResults = new ArrayList<>();

        // Parse the data from the file to a JSONArray.
        JSONArray jsonDevicePingResults = fileData.getJSONArray("devicePingResults");

        // Loop through each JSONObject in the JSONArray.
        for (int i = 0; i < jsonDevicePingResults.length(); i++) {
            JSONObject jsonDevicePingResult = jsonDevicePingResults.getJSONObject(i);

            // Parse the individual data values from the JSONObject, and use them to create a
```

```
new DevicePingResult object.
            DevicePingResult devicePingResult = new DevicePingResult(
                    jsonDevicePingResult.getInt("roundTripTime"),
                    jsonDevicePingResult.getBoolean("successfulPing"),
                    jsonDevicePingResult.getDouble("packetLoss")
            );
            // Append the SubnetScanResult to the list.
            devicePingResults.add(devicePingResult);
        }

        return devicePingResults;
    }
```

Import Device Ping Data Method

```
    /**
     * Imports device ping scan results by extracting the target IP address,
     * ping interval, number of pings, continuous pinging flag, and the list of
     * device ping results from the JSON data, then passes the data to the
     * listener.
     */
    private void importDevicePingData() {
        // Create the temporary holder variables.
        String ipAddress = fileData.getString("ipAddress");
        int pingInterval = fileData.getInt("pingInterval");
        int numOfPings = fileData.getInt("numOfPings");
        boolean continuousPinging = fileData.getBoolean("continuousPinging");
        ArrayList<DevicePingResult> devicePingResults = parseDevicePingResults();

        // Call the listener to indicate that a device ping has been imported.
        listener.onDevicePingResultsImported(ipAddress, pingInterval, numOfPings,
continuousPinging, devicePingResults);
    }
```

Parse Port Scan Array Method

```
    /**
     * Parses port scan results from the JSON array and converts them into a
     * list of {@code PortScanResult} objects.
     *
     * @return an {@code ArrayList} of {@code PortScanResult} objects parsed
     * from the JSON data
     */
    private ArrayList<PortScanResult> parsePortScanResults() {
        // Create the ArrayList of PortScanResult objects.
        ArrayList<PortScanResult> portScanResults = new ArrayList<>();

        // Parse the data from the file to a JSONArray.
        JSONArray jsonPortScanResults = fileData.getJSONArray("portScanResults");

        // Loop through each JSONObject in the JSONArray.
        for (int i = 0; i < jsonPortScanResults.length(); i++) {
            JSONObject jsonPortScanResult = jsonPortScanResults.getJSONObject(i);

            // Parse the individual data values from the JSONObject, and use them to create a
new PortScanResult object.
            PortScanResult portScanResult = new PortScanResult(
                    jsonPortScanResult.getInt("portNumber"),
                    jsonPortScanResult.getString("protocol")
            );
            // Append the SubnetScanResult to the list.
            portScanResults.add(portScanResult);
        }

        return portScanResults;
    }
```

Import Port Scan Data Method

```
    /**
     * Imports port scan scan results by extracting the target IP address, port
     * range, timeout, and the list of port scan results from the JSON data,
     * then passes the data to the listener.
     */
    private void importPortScanData() {
        // Create the temporary holder variables.
        String ipAddress = fileData.getString("ipAddress");
        int bottomRangePort = fileData.getInt("bottomRangePort");
        int topRangePort = fileData.getInt("topRangePort");
        int timeout = fileData.getInt("timeout");
        ArrayList<PortScanResult> portScanResults = parsePortScanResults();

        // Call the listener to indicate that a port scan has been imported.
        listener.onPortScanResultsImported(ipAddress, bottomRangePort, topRangePort, timeout,
portScanResults);
    }
```