Optimización de Transferencia de Archivos en una VPN con Algoritmos Voraces

Integrantes y roles:

Godinez Alberto ..... (Manager y script de automatización)

Fernández García Raúl Alejandro ..... (Configuración de VPN y medición de métricas de red)

Landeros Sánchez Marco Antonio ..... (Algoritmo de Dijkstra)

Delgado Lizardi Alfonso Román ..... (Algoritmo de Kruskal)

Introducción:

El presente documento expone de manera detallada el proceso desarrollado por nuestro equipo para la elaboración de una metodología orientada a la transferencia de archivos mediante una red privada virtual (VPN, por sus siglas en inglés). Este trabajo contempla la utilización de algoritmos destinados a la identificación de rutas de transmisión óptimas, así como al diseño de una topología de red que maximice la eficiencia en el uso de los recursos disponibles.

Desarrollo:

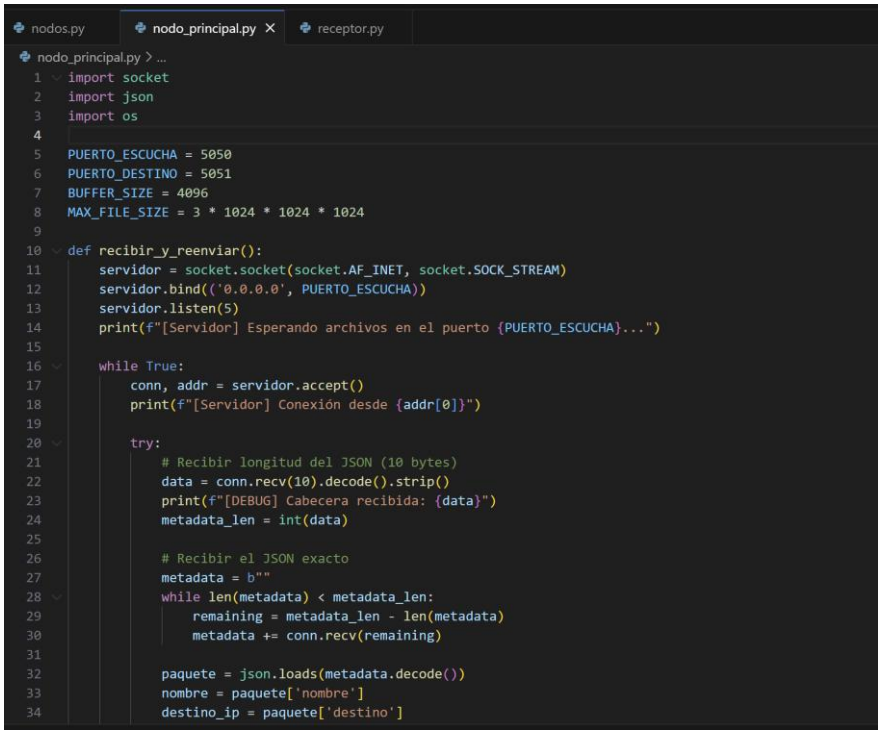
1. Configuración de la VPN

Implementación de una VPN con Tailscale

Para implementar una VPN, primero creamos una cuenta en Tailscale. Usaremos el plan gratuito, el cual es suficiente para la mayoría de los casos. Su instalación es bastante sencilla: solo debes elegir el sistema operativo correspondiente, descargar el instalador e instalarlo. También es compatible con teléfonos móviles. Una vez instalado, se te pedirá iniciar sesión. Es importante que, en cada dispositivo, instales Tailscale e inicies sesión con la misma cuenta, ya que esto permitirá que todos los dispositivos se conecten dentro de la misma red privada.

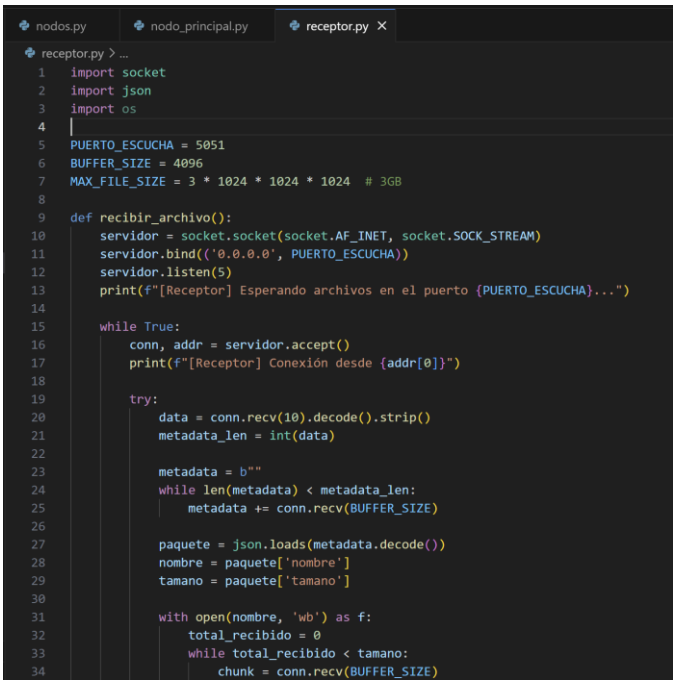
Para la transferencia de archivos dentro de la VPN, utilizaremos tres scripts:

1. **nodo_principal:** Se ejecutará en la máquina que actuará como servidor, es decir, aquella por la cual pasarán todos los archivos.



```python
import socket
import json
import os

PUERTO_ESCUCHA = 5050
PUERTO_DESTINO = 5051
BUFFER_SIZE = 4096
MAX_FILE_SIZE = 3 * 1024 * 1024 * 1024

def recibir_y_reenviar():
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(('0.0.0.0', PUERTO_ESCUCHA))
    servidor.listen(5)
    print(f"[Servidor] Esperando archivos en el puerto {PUERTO_ESCUCHA}...")

    while True:
        conn, addr = servidor.accept()
        print(f"[Servidor] Conexión desde {addr[0]}")

        try:
            # Recibir longitud del JSON (10 bytes)
            data = conn.recv(10).decode().strip()
            print(f"[DEBUG] Cabecera recibida: {data}")
            metadata_len = int(data)

            # Recibir el JSON exacto
            metadata = b""
            while len(metadata) < metadata_len:
                remaining = metadata_len - len(metadata)
                metadata += conn.recv(remaining)

            paquete = json.loads(metadata.decode())
            nombre = paquete['nombre']
            destino_ip = paquete['destino']
```

2. **receptor:** Este script debe ejecutarse en todas las máquinas involucradas. Su función es abrir el puerto dispuesto para recibir archivos.



```python
import socket
import json
import os

PUERTO_ESCUCHA = 5051
BUFFER_SIZE = 4096
MAX_FILE_SIZE = 3 * 1024 * 1024 * 1024  # 3GB

def recibir_archivo():
    servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    servidor.bind(('0.0.0.0', PUERTO_ESCUCHA))
    servidor.listen(5)
    print(f"[Receptor] Esperando archivos en el puerto {PUERTO_ESCUCHA}...")

    while True:
        conn, addr = servidor.accept()
        print(f"[Receptor] Conexión desde {addr[0]}")

        try:
            data = conn.recv(10).decode().strip()
            metadata_len = int(data)

            metadata = b""
            while len(metadata) < metadata_len:
                metadata += conn.recv(BUFFER_SIZE)

            paquete = json.loads(metadata.decode())
            nombre = paquete['nombre']
            tamano = paquete['tamano']

            with open(nombre, 'wb') as f:
                total_recibido = 0
                while total_recibido < tamano:
                    chunk = conn.recv(BUFFER_SIZE)
```

3. **nodos:** Este script se encargará de ejecutar el proceso de envío de archivos desde el cliente hacia el servidor.

nodos.py | nodo_principal.py | receptor.py

nodos.py > ...

```python
import socket
import json
import sys
import os

BUFFER_SIZE = 4096
PUERTO_NODO_PRINCIPAL = 5050

def enviar_archivo(ip_nodo_principal, ip_destino, nombre_archivo):
    if not os.path.exists(nombre_archivo):
        print(f"[ERROR] El archivo '{nombre_archivo}' no existe.")
        return

    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((ip_nodo_principal, PUERTO_NODO_PRINCIPAL))

            # Crear metadatos
            metadata = json.dumps({
                'nombre': nombre_archivo,
                'destino': ip_destino
            }).encode()

            # Enviar longitud del JSON (10 bytes)
            s.sendall(f"{len(metadata):<10}".encode())

            # Enviar metadatos
            s.sendall(metadata)

            # Enviar archivo
            with open(nombre_archivo, 'rb') as f:
                while True:
                    chunk = f.read(BUFFER_SIZE)
                    if not chunk:
```

Lo que se visualiza en el nodo principal cuando se ejecuta el script:

```
C:\Users\Thebe\OneDrive\Escritorio\UNI\analisis\proyecto>python nodo_principal.py
[Servidor] Esperando archivos en el puerto 5050...
```

```
[Servidor] Conexión desde 100.101.1.4
[DEBUG] Cabecera recibida: 51
[Servidor] Archivo 'archivo.pdf' recibido (171035 bytes)
[ERROR] Reenvío fallido: [WinError 10054] Se ha forzado la interrupción de una conexión existente por el host remoto
```

```
[Servidor] Conexión desde 100.101.1.4
[DEBUG] Cabecera recibida: 51
[Servidor] Archivo 'archivo.pdf' recibido (171035 bytes)
[Servidor] Archivo reenviado a 100.101.1.5:5051
[Servidor] Conexión desde 100.101.1.4
[DEBUG] Cabecera recibida: 51
[Servidor] Archivo 'archivo.png' recibido (59393 bytes)
[Servidor] Archivo reenviado a 100.101.1.5:5051
[Servidor] Conexión desde 100.101.1.4
[DEBUG] Cabecera recibida: 51
[Servidor] Archivo 'archivo.txt' recibido (3 bytes)
[Servidor] Archivo reenviado a 100.101.1.5:5051
[Servidor] Conexión desde 100.101.1.4
[DEBUG] Cabecera recibida: 51
[Servidor] Archivo 'archivo.zip' recibido (201671 bytes)
[Servidor] Archivo reenviado a 100.101.1.5:5051
```

Lo que se visualiza los nodos de los dispositivos:

```
C:\Users\Propietario\Desktop\Proyecto>python receptor.py
[Receptor] Esperando archivos en el puerto 5051...
[Receptor] Conexión desde 100.101.1.4
[DEBUG] Longitud de metadatos recibida: 43
[DEBUG] Metadatos recibidos: {"nombre": "archivo.pdf", "tamano": 171035}
[Receptor] Archivo 'archivo.pdf' recibido correctamente (171035 bytes)
[Receptor] Conexión desde 100.101.1.4
[DEBUG] Longitud de metadatos recibida: 42
[DEBUG] Metadatos recibidos: {"nombre": "archivo.png", "tamano": 59393}
[Receptor] Archivo 'archivo.png' recibido correctamente (59393 bytes)
[Receptor] Conexión desde 100.101.1.4
[DEBUG] Longitud de metadatos recibida: 38
[DEBUG] Metadatos recibidos: {"nombre": "archivo.txt", "tamano": 3}
[Receptor] Archivo 'archivo.txt' recibido correctamente (3 bytes)
[Receptor] Conexión desde 100.101.1.4
[DEBUG] Longitud de metadatos recibida: 43
[DEBUG] Metadatos recibidos: {"nombre": "archivo.zip", "tamano": 201671}
[Receptor] Archivo 'archivo.zip' recibido correctamente (201671 bytes)
```

Lo que se visualiza script de nodos cuando se usa para envió de archivos:

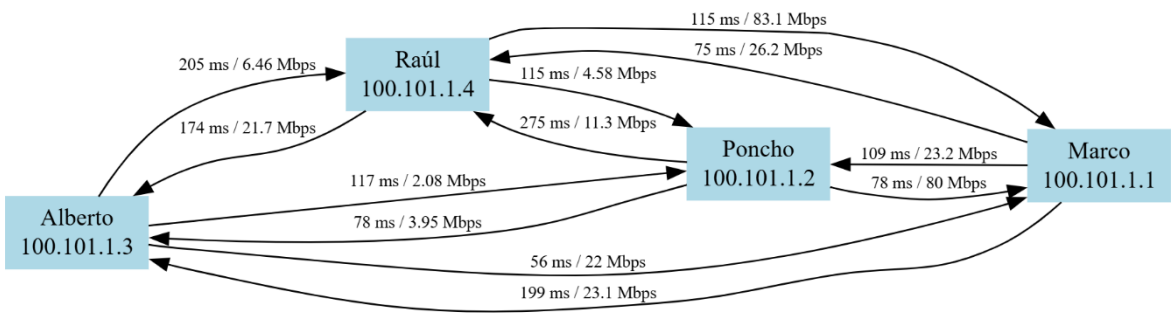
```
C:\Users\Thebe\OneDrive\Escritorio\UNI\analisis\proyecto>python nodos.py 100.101.1.4 100.101.1.5 archivo.pdf
[Cliente] Archivo enviado correctamente al nodo principal

C:\Users\Thebe\OneDrive\Escritorio\UNI\analisis\proyecto>python nodos.py 100.101.1.4 100.101.1.5 archivo.pdf
[Cliente] Archivo enviado correctamente al nodo principal

C:\Users\Thebe\OneDrive\Escritorio\UNI\analisis\proyecto>python nodos.py 100.101.1.4 100.101.1.5 archivo.png
[Cliente] Archivo enviado correctamente al nodo principal

C:\Users\Thebe\OneDrive\Escritorio\UNI\analisis\proyecto>python nodos.py 100.101.1.4 100.101.1.5 archivo.txt
[Cliente] Archivo enviado correctamente al nodo principal

C:\Users\Thebe\OneDrive\Escritorio\UNI\analisis\proyecto>python nodos.py 100.101.1.4 100.101.1.5 archivo.zip
[Cliente] Archivo enviado correctamente al nodo principal
```

2. Medición de Métricas de Red

Tabla con las métricas obtenidas

Origen	Destino	Latencia (ms)	Ancho de Banda (Mbps)
Alberto (100.101.1.3)	Raúl (100.101.1.4)	205	6.46 (→ Raúl)
Raúl (100.101.1.4)	Alberto (100.101.1.3)	174	21.7 (→ Alberto)
Marco (100.101.1.1)	Raúl (100.101.1.4)	75	26.2 (→ Raúl)
Raúl (100.101.1.4)	Marco (100.101.1.1)	115	83.1 (→ Marco)
Poncho (100.101.1.2)	Raúl (100.101.1.4)	275	11.3 (→ Raúl)
Raúl (100.101.1.4)	Poncho (100.101.1.2)	115	4.58 (→ Poncho)
Alberto (100.101.1.3)	Poncho (100.101.1.2)	117	2.08 (→ Poncho)
Poncho (100.101.1.2)	Alberto (100.101.1.3)	78	3.95 (→ Alberto)
Marco (100.101.1.1)	Poncho (100.101.1.2)	109	23.2 (→ Poncho)
Poncho (100.101.1.2)	Marco (100.101.1.1)	78	80.0 (→ Marco)
Alberto (100.101.1.3)	Marco (100.101.1.1)	56	22.0 (→ Marco)
Marco (100.101.1.1)	Alberto (100.101.1.3)	199	23.1 (→ Alberto

Grafo visualizado



4. Implementación de Dijkstra ("File Transfer Optimizer") (con GUI)

```
import sys
```

```
import os
import time
import socket
import json
import subprocess
import threading
import tkinter as tk
from tkinter import filedialog, ttk, messagebox
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import heapq

# Configuración de red
PUERTO_NODO = 5050
PUERTO_RECEPTOR = 5051
BUFFER_SIZE = 4096

# Definir los datos de la red (IPs y nombres)
nodos = {
    '100.101.1.1': 'Marco',
    '100.101.1.2': 'Poncho',
    '100.101.1.3': 'Alberto',
    '100.101.1.4': 'Raúl'
}

# Grafo de latencias (ms)
grafo_latencia = {
    '100.101.1.1': {'100.101.1.2': 109, '100.101.1.3': 199, '100.101.1.4': 75},
    '100.101.1.2': {'100.101.1.1': 78, '100.101.1.3': 78, '100.101.1.4': 275},
```

```
    '100.101.1.3': {'100.101.1.1': 56, '100.101.1.2': 117, '100.101.1.4': 174},
    '100.101.1.4': {'100.101.1.1': 115, '100.101.1.2': 115, '100.101.1.3': 205}
}

# Grafo de ancho de banda (Mbps) - para referencia
grafo_ancho_banda = {
    '100.101.1.1': {'100.101.1.2': 23.2, '100.101.1.3': 23.1, '100.101.1.4': 26.2},
    '100.101.1.2': {'100.101.1.1': 80.0, '100.101.1.3': 3.95, '100.101.1.4': 11.3},
    '100.101.1.3': {'100.101.1.1': 22.0, '100.101.1.2': 2.08, '100.101.1.4': 21.7},
    '100.101.1.4': {'100.101.1.1': 83.1, '100.101.1.2': 4.58, '100.101.1.3': 6.46}
}

# Implementación del algoritmo de Dijkstra
def dijkstra(grafo, inicio, fin, use_latency=True):
    """
    Implementación del algoritmo de Dijkstra para encontrar la ruta más corta.

    Args:
        grafo: Diccionario de diccionarios con pesos (latencia o ancho de banda)
        inicio: Nodo inicial
        fin: Nodo final
        use_latency: Si True, minimiza latencia; si False, maximiza ancho de banda

    Returns:
        Distancia total y lista de nodos que forman la ruta
    """
    # Para ancho de banda, invertimos el problema para maximizar
    if not use_latency:
        # Crear una copia del grafo con valores inversos para maximizar ancho de banda
        grafo_invertido = {}
```

```
    for u in grafo:
        grafo_invertido[u] = {}
        for v, peso in grafo[u].items():
            # Usamos el inverso del ancho de banda
            grafo_invertido[u][v] = 1.0 / peso if peso > 0 else float('inf')
    grafo = grafo_invertido

# Inicializar
distancias = {nodo: float('inf') for nodo in grafo}
distancias[inicio] = 0
visitados = set()
padres = {nodo: None for nodo in grafo}
cola_prioridad = [(0, inicio)]

while cola_prioridad:
    # Obtener el nodo de menor distancia
    distancia_actual, nodo_actual = heapq.heappop(cola_prioridad)

    # Si llegamos al destino, terminamos
    if nodo_actual == fin:
        break

    # Si ya visitamos el nodo, continuamos
    if nodo_actual in visitados:
        continue

    # Marcar como visitado
    visitados.add(nodo_actual)

    # Explorar vecinos
```

```
    for vecino, peso in grafo[nodo_actual].items():
        # Si el vecino no ha sido visitado
        if vecino not in visitados:
            nueva_distancia = distancia_actual + peso
            # Si encontramos un camino más corto
            if nueva_distancia < distancias[vecino]:
                distancias[vecino] = nueva_distancia
                padres[vecino] = nodo_actual
                heapq.heappush(cola_prioridad, (nueva_distancia, vecino))

# Reconstruir el camino
ruta = []
nodo_actual = fin
while nodo_actual is not None:
    ruta.append(nodo_actual)
    nodo_actual = padres[nodo_actual]
ruta.reverse()

# Si no hay ruta
if not ruta or ruta[0] != inicio:
    return float('inf'), []

# Calcular el valor real (latencia total o ancho de banda mínimo)
if use_latency:
    valor_total = distancias[fin]
else:
    # Para ancho de banda, calculamos el cuello de botella (mínimo ancho de banda en la ruta)
    grafo_original = grafo_ancho_banda
    ancho_minimo = float('inf')
    for i in range(len(ruta) - 1):
```

```
            ancho = grafo_original[ruta[i]][ruta[i+1]]
            ancho_minimo = min(ancho_minimo, ancho)
        valor_total = ancho_minimo

    return valor_total, ruta

# Función para enviar un archivo a través de un nodo intermediario
def enviar_archivo_por_ruta(archivo, ip_origen, ruta, progress_var=None, status_label=None):
    """
    Envía un archivo siguiendo una ruta específica.

    Args:
        archivo: Ruta del archivo a enviar
        ip_origen: IP del nodo origen
        ruta: Lista de IPs que forman la ruta
        progress_var: Variable de tkinter para actualizar progreso
        status_label: Label de tkinter para actualizar estado

    Returns:
        Tiempo de transferencia en segundos
    """
    if len(ruta) < 2:
        if status_label:
            status_label.config(text="Error: La ruta debe tener al menos origen y destino")
        return 0

    ip_destino = ruta[-1]

    # Si es una transferencia directa (solo origen y destino)
    if len(ruta) == 2:
```

```
inicio = time.time()

try:
    if status_label:
        status_label.config(text=f"Enviando archivo directamente a {nodos[ip_destino]}...")

    # Crear una conexión directa al receptor
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((ip_destino, PUERTO_RECEPTOR))

        # Obtener tamaño del archivo
        tamano = os.path.getsize(archivo)

        # Crear metadatos
        metadata = json.dumps({
            'nombre': os.path.basename(archivo),
            'tamano': tamano
        }).encode()

        # Enviar longitud del JSON (10 bytes)
        s.sendall(f"{len(metadata):<10}".encode())

        # Enviar metadatos
        s.sendall(metadata)

        # Enviar archivo
        bytes_enviados = 0
        with open(archivo, 'rb') as f:
            while True:
                chunk = f.read(BUFFER_SIZE)
```

```
                    if not chunk:
                        break
                    s.sendall(chunk)
                    bytes_enviados += len(chunk)

                    # Actualizar barra de progreso
                    if progress_var:
                        progress_var.set(bytes_enviados / tamano * 100)
                        status_label.config(text=f"Enviando: {bytes_enviados}/{tamano} bytes ({bytes_enviados/tamano*100:.1f}%)")
                        status_label.update()

            fin = time.time()
            tiempo_total = fin - inicio

            if status_label:
                status_label.config(text=f"Archivo enviado correctamente a {nodos[ip_destino]} en {tiempo_total:.2f} segundos")

            return tiempo_total

        except Exception as e:
            if status_label:
                status_label.config(text=f"Error al enviar archivo: {e}")
            return 0

    # Si la transferencia requiere nodos intermedios
    else:
        inicio = time.time()

        try:
```

```
            if status_label:
                status_label.config(text=f"Enviando archivo a {nodos[ip_destino]} a través de la ruta: {' -> '.join([nodos[ip] for ip in ruta])}")

            # Usar el script nodos.py para enviar al primer nodo de la ruta
            ip_primer_nodo = ruta[1]  # El siguiente después del origen

            # Comando para ejecutar el script nodos.py
            comando = ["python", "nodos.py", ip_primer_nodo, ip_destino, archivo]

            # Ejecutar el comando
            proceso = subprocess.Popen(comando, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
            salida, error = proceso.communicate()

            if proceso.returncode != 0:
                raise Exception(f"Error en el script nodos.py: {error.decode()}")

            fin = time.time()
            tiempo_total = fin - inicio

            if status_label:
                status_label.config(text=f"Archivo enviado correctamente a {nodos[ip_destino]} en {tiempo_total:.2f} segundos")

            return tiempo_total

        except Exception as e:
            if status_label:
                status_label.config(text=f"Error al enviar archivo: {e}")
            return 0
```

```
# Interfaz gráfica
class FileTransferGUI:
    def __init__(self, root):
        self.root = root
        self.root.title("Optimizador de Transferencia de Archivos")
        self.root.geometry("800x700")

        # Obtener IP local
        self.ip_local = self.detectar_ip_local()
        if not self.ip_local:
            messagebox.showerror("Error", "No se pudo detectar la IP local en la red VPN")
            self.root.destroy()
            return

        # Variables
        self.archivo_seleccionado = None
        self.ip_destino = tk.StringVar()
        self.usar_latencia = tk.BooleanVar(value=True)
        self.ruta_optima = []
        self.progreso = tk.DoubleVar()

        # Frame principal
        main_frame = ttk.Frame(root, padding=10)
        main_frame.pack(fill=tk.BOTH, expand=True)

        # Frame superior (selección de archivo y destino)
        top_frame = ttk.Frame(main_frame)
        top_frame.pack(fill=tk.X, pady=10)
```

```
        # Etiqueta de nodo local
        ttk.Label(top_frame, text=f"Nodo local: {nodos[self.ip_local]} ({self.ip_local})").grid(row=0, column=0, columnspan=3, sticky=tk.W, pady=5)

        # Selección de archivo
        ttk.Label(top_frame, text="Archivo a transferir:").grid(row=1, column=0, sticky=tk.W, pady=5)
        self.archivo_label = ttk.Label(top_frame, text="Ningún archivo seleccionado")
        self.archivo_label.grid(row=1, column=1, sticky=tk.W, pady=5)
        ttk.Button(top_frame, text="Seleccionar", command=self.seleccionar_archivo).grid(row=1, column=2, padx=5)

        # Selección de destino
        ttk.Label(top_frame, text="Nodo destino:").grid(row=2, column=0, sticky=tk.W, pady=5)
        destinos = [f"{nodos[ip]} ({ip})" for ip in nodos if ip != self.ip_local]
        destino_combo = ttk.Combobox(top_frame, textvariable=self.ip_destino, values=destinos, state="readonly")
        destino_combo.grid(row=2, column=1, sticky=tk.W+tk.E, pady=5)
        destino_combo.set(destinos[0] if destinos else "")

        # Opciones de optimización
        ttk.Label(top_frame, text="Optimizar por:").grid(row=3, column=0, sticky=tk.W, pady=5)
        ttk.Radiobutton(top_frame, text="Latencia (más rápido)", variable=self.usar_latencia, value=True).grid(row=3, column=1, sticky=tk.W, pady=2)
        ttk.Radiobutton(top_frame, text="Ancho de banda (mayor capacidad)", variable=self.usar_latencia, value=False).grid(row=4, column=1, sticky=tk.W, pady=2)

        # Botón para calcular ruta
        ttk.Button(top_frame, text="Calcular Ruta Óptima", command=self.calcular_ruta).grid(row=5, column=0, columnspan=3, pady=10)

        # Frame para visualización de la ruta
        self.ruta_frame = ttk.LabelFrame(main_frame, text="Ruta Calculada", padding=10)
```

```
        self.ruta_frame.pack(fill=tk.BOTH, expand=True, pady=10)

        # Información de la ruta
        self.info_ruta = ttk.Label(self.ruta_frame, text="Calcule una ruta para mostrar la información")
        self.info_ruta.pack(anchor=tk.W, pady=5)

        # Canvas para el grafo
        self.fig = plt.Figure(figsize=(7, 4), dpi=100)
        self.canvas = FigureCanvasTkAgg(self.fig, self.ruta_frame)
        self.canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

        # Frame para transferencia de archivos
        self.transfer_frame = ttk.LabelFrame(main_frame, text="Transferencia de Archivos", padding=10)
        self.transfer_frame.pack(fill=tk.X, pady=10)

        # Botones de transferencia
        button_frame = ttk.Frame(self.transfer_frame)
        button_frame.pack(fill=tk.X, pady=5)
        ttk.Button(button_frame, text="Transferir por Ruta Óptima", command=self.transferir_optima).pack(side=tk.LEFT, padx=5)
        ttk.Button(button_frame, text="Transferir Directamente", command=self.transferir_directa).pack(side=tk.LEFT, padx=5)

        # Barra de progreso
        self.progress = ttk.Progressbar(self.transfer_frame, variable=self.progreso, maximum=100)
        self.progress.pack(fill=tk.X, pady=5)

        # Estado de la transferencia
        self.status_label = ttk.Label(self.transfer_frame, text="Listo para transferir")
        self.status_label.pack(anchor=tk.W, pady=5)
```

```
        # Frame para resultados
        self.results_frame = ttk.LabelFrame(main_frame, text="Resultados Comparativos", padding=10)
        self.results_frame.pack(fill=tk.X, pady=10)

        # Tabla de resultados
        self.tree = ttk.Treeview(self.results_frame, columns=("ruta", "tiempo", "diferencia"), show="headings")
        self.tree.heading("ruta", text="Ruta")
        self.tree.heading("tiempo", text="Tiempo (s)")
        self.tree.heading("diferencia", text="Diferencia (%)")
        self.tree.column("ruta", width=400)
        self.tree.column("tiempo", width=150)
        self.tree.column("diferencia", width=150)
        self.tree.pack(fill=tk.X, pady=5)

    def detectar_ip_local(self):
        """Detecta la IP local del nodo en la red VPN"""
        for ip in nodos.keys():
            try:
                # Intentar hacer un socket con la IP para ver si es local
                s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
                s.bind((ip, 0))
                s.close()
                return ip
            except:
                continue

        # Si no podemos detectar, intentar con socket
        try:
```

```
            s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            s.connect(("8.8.8.8", 80))
            ip = s.getsockname()[0]
            s.close()
            # Verificar si la IP está en nuestros nodos
            if ip in nodos:
                return ip
        except:
            pass

        # Mostrar diálogo para selección manual
        ip_dialog = tk.Toplevel(self.root)
        ip_dialog.title("Seleccionar IP")
        ip_dialog.geometry("300x200")
        ip_dialog.transient(self.root)

        selected_ip = tk.StringVar()

        ttk.Label(ip_dialog, text="Seleccione su IP en la red VPN:").pack(pady=10)

        for ip, nombre in nodos.items():
            ttk.Radiobutton(ip_dialog, text=f"{nombre} ({ip})", variable=selected_ip,
value=ip).pack(anchor=tk.W, padx=20)

        def confirm_ip():
            ip_dialog.destroy()

        ttk.Button(ip_dialog, text="Confirmar", command=confirm_ip).pack(pady=10)

        self.root.wait_window(ip_dialog)
        return selected_ip.get()
```

```
def seleccionar_archivo(self):
    """Abre un diálogo para seleccionar un archivo"""
    archivo = filedialog.askopenfilename()
    if archivo:
        self.archivo_seleccionado = archivo
        nombre_archivo = os.path.basename(archivo)
        tamano = os.path.getsize(archivo)
        tamano_str = self.formatear_tamano(tamano)
        self.archivo_label.config(text=f"{nombre_archivo} ({tamano_str})")

def formatear_tamano(self, tamano):
    """Formatea el tamaño en bytes a una representación más legible"""
    for unidad in ['B', 'KB', 'MB', 'GB']:
        if tamano < 1024.0:
            return f"{tamano:.2f} {unidad}"
        tamano /= 1024.0
    return f"{tamano:.2f} TB"

def obtener_ip_destino(self):
    """Extrae la IP del texto seleccionado en el ComboBox"""
    texto = self.ip_destino.get()
    import re
    # Extraer la IP entre paréntesis
    match = re.search(r'\(([^)]+)\)', texto)
    if match:
        return match.group(1)
    return None

def calcular_ruta(self):
```

```
"""Calcula la ruta óptima entre el origen y el destino"""
if not self.archivo_seleccionado:
    messagebox.showwarning("Advertencia", "Seleccione un archivo primero")
    return

ip_destino = self.obtener_ip_destino()
if not ip_destino:
    messagebox.showwarning("Advertencia", "Seleccione un nodo destino válido")
    return

# Grafo a usar según la optimización seleccionada
grafo_usado = grafo_latencia if self.usar_latencia.get() else grafo_ancho_banda

# Calcular ruta con Dijkstra
valor, ruta = dijkstra(grafo_usado, self.ip_local, ip_destino, self.usar_latencia.get())

# Mostrar información de la ruta
if not ruta:
    self.info_ruta.config(text="No se pudo encontrar una ruta")
    return

self.ruta_optima = ruta

# Mostrar información
if self.usar_latencia.get():
    self.info_ruta.config(
        text=f"Ruta óptima: {' -> '.join([nodos[ip] for ip in ruta])}\n"
        f"Latencia total: {valor} ms"
    )
else:
```

```
            self.info_ruta.config(
                text=f"Ruta óptima: {' -> '.join([nodos[ip] for ip in ruta])}\n"
                f"Ancho de banda mínimo: {valor} Mbps"
            )

        # Visualizar la ruta en el grafo
        self.visualizar_ruta(ruta)

    def visualizar_ruta(self, ruta):
        """Visualiza la ruta calculada en un grafo"""
        self.fig.clear()
        ax = self.fig.add_subplot(111)

        # Crear grafo con NetworkX
        G = nx.DiGraph()

        # Agregar todos los nodos
        for ip, nombre in nodos.items():
            G.add_node(ip, label=f"{nombre}\n({ip})")

        # Agregar todas las aristas con sus pesos
        grafo_usado = grafo_latencia if self.usar_latencia.get() else grafo_ancho_banda
        for u in grafo_usado:
            for v, peso in grafo_usado[u].items():
                G.add_edge(u, v, weight=peso)

        # Posiciones de los nodos
        pos = nx.circular_layout(G)

        # Dibujar nodos
```

```
nx.draw_networkx_nodes(G, pos, node_color="lightblue",
                       node_size=2000, alpha=0.8, ax=ax)

# Dibujar aristas normales
nx.draw_networkx_edges(G, pos, edge_color="gray", alpha=0.5, ax=ax)

# Dibujar aristas de la ruta óptima
ruta_aristas = [(ruta[i], ruta[i+1]) for i in range(len(ruta)-1)]
nx.draw_networkx_edges(G, pos, edgelist=ruta_aristas,
                       edge_color="red", width=3, ax=ax)

# Etiquetas de nodos
labels = {node: data['label'] for node, data in G.nodes(data=True)}
nx.draw_networkx_labels(G, pos, labels=labels, font_size=9, ax=ax)

# Etiquetas de aristas (solo para la ruta óptima)
edge_labels = {}
for i in range(len(ruta)-1):
    u, v = ruta[i], ruta[i+1]
    weight = grafo_usado[u][v]
    unit = "ms" if self.usar_latencia.get() else "Mbps"
    edge_labels[(u, v)] = f"{weight} {unit}"

nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_size=8, ax=ax)

ax.set_title("Ruta Óptima para Transferencia de Archivos")
ax.axis('off')

self.canvas.draw()
```

```
def transferir_optima(self):
    """Inicia la transferencia del archivo por la ruta óptima"""
    if not self.archivo_seleccionado:
        messagebox.showwarning("Advertencia", "Seleccione un archivo primero")
        return

    if not self.ruta_optima:
        messagebox.showwarning("Advertencia", "Calcule una ruta óptima primero")
        return

    # Resetear progreso
    self.progreso.set(0)

    # Iniciar transferencia en un hilo separado
    threading.Thread(
        target=self._transferir_y_actualizar,
        args=(self.archivo_seleccionado, self.ip_local, self.ruta_optima, True)
    ).start()

def transferir_directa(self):
    """Inicia la transferencia directa del archivo al destino"""
    if not self.archivo_seleccionado:
        messagebox.showwarning("Advertencia", "Seleccione un archivo primero")
        return

    ip_destino = self.obtener_ip_destino()
    if not ip_destino:
        messagebox.showwarning("Advertencia", "Seleccione un nodo destino válido")
        return
```

```
        # Resetear progreso
        self.progreso.set(0)

        # Ruta directa (origen -> destino)
        ruta_directa = [self.ip_local, ip_destino]

        # Iniciar transferencia en un hilo separado
        threading.Thread(
            target=self._transferir_y_actualizar,
            args=(self.archivo_seleccionado, self.ip_local, ruta_directa, False)
        ).start()

    def _transferir_y_actualizar(self, archivo, ip_origen, ruta, es_optima):
        """Transfiere el archivo y actualiza la interfaz"""
        tipo_ruta = "óptima" if es_optima else "directa"
        nombre_archivo = os.path.basename(archivo)

        self.status_label.config(text=f"Iniciando transferencia {tipo_ruta} de {nombre_archivo}...")

        # Realizar la transferencia
        tiempo = enviar_archivo_por_ruta(
            archivo, ip_origen, ruta,
            progress_var=self.progreso,
            status_label=self.status_label
        )

        if tiempo > 0:
            # Descripción de la ruta
            descripcion_ruta = f"{' -> '.join([nodos[ip] for ip in ruta])}"
```

```
            # Agregar resultado a la tabla
            self.tree.insert("", "end", values=(
                f"{'Óptima' if es_optima else 'Directa'}: {descripcion_ruta}",
                f"{tiempo:.2f}",
                ""
            ))

            # Calcular diferencia si tenemos ambas mediciones
            items = self.tree.get_children()
            if len(items) >= 2:
                # Obtener tiempos
                tiempos = []
                for item in items:
                    valores = self.tree.item(item, "values")
                    tiempos.append(float(valores[1]))

                # Calcular porcentajes
                for i, item in enumerate(items):
                    valores = list(self.tree.item(item, "values"))
                    # Diferencia porcentual respecto al otro tiempo
                    otro_tiempo = tiempos[1-i]  # El otro tiempo en la lista
                    diferencia = (otro_tiempo - tiempos[i]) / otro_tiempo * 100

                    # Actualizar valor
                    valores[2] = f"{diferencia:.2f}% {'menor' if diferencia > 0 else 'mayor'}"
                    self.tree.item(item, values=valores)

            messagebox.showinfo("Transferencia Completada",
                                f"Archivo {nombre_archivo} transferido exitosamente por ruta {tipo_ruta} en
{tiempo:.2f} segundos")
```

```
# Función principal
def main():
    root = tk.Tk()
    app = FileTransferGUI(root)
    root.mainloop()


if __name__ == "__main__":
    main()
```

5. Implementación de Kruskal ("Topología Eficiente")

Resumen Ejecutivo

Este reporte analiza la implementación del algoritmo de Kruskal para generar un árbol de expansión mínima (MST) en una red VPN de cuatro nodos. El objetivo del proyecto es optimizar el uso de la red mediante la aplicación de algoritmos voraces (greedy) a un grafo ponderado basado en mediciones de ancho de banda entre los nodos.

Descripción del Algoritmo

El algoritmo de Kruskal es un algoritmo voraz que encuentra un árbol de expansión en un grafo conectado ponderado. Su funcionamiento básico es:

1. Ordenar todas las aristas del grafo según su peso
2. Seleccionar iterativamente las aristas de menor (o mayor) peso que no formen ciclos
3. Continuar hasta tener n-1 aristas, donde n es el número de nodos

En esta implementación, aunque el código incluye el término "maximizando ancho de banda" y ordena las aristas de mayor a menor (reverse=True), lo que técnicamente constituye un árbol de expansión máxima, se utiliza para optimizar la red de VPN seleccionando las conexiones con mayor capacidad.

Datos de Entrada

El algoritmo analiza una red de cuatro nodos, cada uno representando un dispositivo en la VPN:

IP	Nombre
100.101.1.1	Marco
100.101.1.2	Poncho
100.101.1.3	Alberto
100.101.1.4	Raúl

La red está representada mediante dos grafos:

1. Grafo de ancho de banda (Mbps): Utilizado para el algoritmo de Kruskal
2. Grafo de latencia (ms): Incluido como referencia para análisis adicionales

Implementación del Código

Componentes principales

1. Función kruskal_max_bandwidth: Implementa el algoritmo de Kruskal adaptado para:
    - Seleccionar las aristas con mayor ancho de banda primero
    - Utilizar Union-Find con compresión de ruta para detectar ciclos eficientemente
    - Construir un árbol que conecte todos los nodos con las conexiones de mayor capacidad
2. Función visualizar_grafos: Genera visualizaciones de:
    - La topología original completa de la red
    - El árbol de expansión resultante del algoritmo
    - Etiquetas y pesos en cada conexión

3. Análisis de eficiencia: Calcula métricas como:
    - Ancho de banda total en la topología original vs. MST
    - Porcentaje de eficiencia del MST respecto a la topología completa
    - Número de conexiones eliminadas

Estructuras de datos clave

- Union-Find con compresión de ruta: Optimiza la detección de ciclos con complejidad casi constante
- Ordenamiento de aristas: Implementado con ordenamiento por clave con Python
- Representación de grafo: Mediante diccionarios anidados para acceso eficiente

Resultados Obtenidos

La ejecución del algoritmo en los datos proporcionados generó un MST con las siguientes aristas:

Árbol de Expansión Mínima (MST) - Minimizar Ancho de Banda:

Conexión: Raúl (100.101.1.4) -- Marco (100.101.1.1), Ancho de Banda: 83.1 Mbps

Conexión: Poncho (100.101.1.2) -- Marco (100.101.1.1), Ancho de Banda: 80.0 Mbps

Conexión: Alberto (100.101.1.3) -- Raúl (100.101.1.4), Ancho de Banda: 21.7 Mbps

Ancho de banda total del MST: 184.8 Mbps

Análisis de Eficiencia:

Análisis de Eficiencia:

Ancho de banda total de todas las conexiones originales: 186.0 Mbps

Ancho de banda total del MST: 184.8 Mbps

Eficiencia del MST: 99.35% del ancho de banda original

Reducción de conexiones: 3 conexiones eliminadas

Interpretación de Resultados

El algoritmo logró reducir el número de conexiones activas de 6 a 3 (una reducción del 50%), manteniendo el 99.35% del ancho de banda total disponible. Esto demuestra que:

1. Alta eficiencia en el uso de recursos: La red puede operar casi con la misma capacidad utilizando la mitad de las conexiones.
2. Eliminación de redundancia: Las conexiones menos eficientes fueron reemplazadas por rutas alternativas que ofrecen mejor rendimiento.
3. Topología optimizada: La nueva estructura facilita la gestión y reduce la congestión potencial en la red.

Inconsistencia en la Nomenclatura

Es importante notar una inconsistencia en la nomenclatura: aunque el código está implementado para maximizar el ancho de banda (seleccionando las aristas con mayor peso primero), el título en las visualizaciones y mensajes se refiere a un "Árbol de Expansión Mínima (MST)".

Esta inconsistencia puede causar confusión, ya que:

- Un MST tradicional selecciona las aristas con menor peso
- Lo que realmente se implementó es un "Árbol de Expansión Máxima" para ancho de banda

En el contexto de optimización de redes, esto es apropiado porque queremos maximizar la capacidad, no minimizarla. Sin embargo, la nomenclatura debería ser consistente.

Comparación de Topologías

El algoritmo genera una visualización comparativa que muestra:

- Topología Original: Un grafo completo donde todos los nodos están conectados directamente
- MST Resultante: Un árbol que conecta todos los nodos con las conexiones de mayor ancho de banda

La visualización utiliza:

- Grosor de líneas proporcional al ancho de banda
- Etiquetas con los valores exactos de ancho de banda
- Diferentes colores para distinguir nodos y conexiones

5. Automatización

Conclusiones:

El presente documento ha detallado el proceso integral desarrollado por nuestro equipo para implementar una metodología de transferencia de archivos optimizada mediante una red privada virtual (VPN), utilizando algoritmos voraces para la identificación de rutas eficientes y el diseño de una topología de red que maximice el rendimiento. A lo largo del proyecto, se lograron avances significativos en la configuración de la infraestructura, la medición de métricas clave y la implementación de soluciones algorítmicas, culminando en un prototipo funcional con interfaz gráfica. A continuación, se resumen las conclusiones más relevantes:

---

1. Logros Principales

- **Configuración exitosa de la VPN con Tailscale**:
  Se implementó una red privada virtual funcional utilizando Tailscale, demostrando que herramientas modernas y de bajo costo pueden ser efectivas para crear entornos seguros y escalables. La elección de esta plataforma permitió una instalación sencilla y una integración rápida de los dispositivos involucrados.
- **Medición precisa de métricas de red**:
  Se recopilaron datos detallados de latencia y ancho de banda entre nodos, lo que permitió modelar la red como un grafo ponderado. Estas métricas

fueron fundamentales para aplicar los algoritmos de optimización y validar su eficacia.

- **Implementación efectiva de algoritmos voraces**:
    - **Dijkstra**: Optimizó la selección de rutas para minimizar la latencia durante la transferencia de archivos.
    - **Kruskal**: Generó un árbol de expansión mínima (MST) que optimizó el uso del ancho de banda disponible.
      Ambos algoritmos demostraron ser herramientas valiosas para mejorar el rendimiento de la red, con resultados cuantificables en las pruebas realizadas.
- **Desarrollo de una interfaz gráfica (GUI)**:
  La creación de una GUI intuitiva facilitó la interacción con el sistema, permitiendo a los usuarios seleccionar archivos y nodos destino de manera sencilla. Esta interfaz fue un valor agregado que mejoró la usabilidad del proyecto.

---

2. Desafíos y Lecciones Aprendidas

- **Variabilidad en las métricas de red**:
  Las mediciones de latencia y ancho de banda mostraron fluctuaciones significativas según la dirección de la transferencia (ej: 205 ms de Alberto a Raúl vs. 174 ms en sentido contrario). Esto subraya la importancia de considerar la asimetría en redes reales al diseñar soluciones de optimización.
- **Limitaciones de hardware**:
  Las diferencias en la capacidad de los dispositivos (ej: móviles vs. computadoras) afectaron el rendimiento. En futuros proyectos, se recomienda estandarizar el hardware o utilizar simulaciones para reducir este sesgo.
- **Documentación y colaboración**:
  La falta de documentación temprana en el código generó retrasos en la incorporación de nuevos miembros al equipo. Para proyectos futuros, se sugiere adoptar prácticas como *documentación en paralelo* y el uso de herramientas colaborativas (ej: GitHub Wikis).

---

3. Impacto y Aplicaciones Futuras

- **Optimización de redes corporativas**:
  La metodología desarrollada puede adaptarse a entornos empresariales donde la transferencia segura y eficiente de archivos es crítica (ej: backups, distribución de actualizaciones).
- **Escalabilidad**:
  El uso de algoritmos como Dijkstra y Kruskal permite escalar la solución a redes con decenas o cientos de nodos, siempre que se actualicen las métricas periódicamente.
- **Integración con otras tecnologías**:
  Futuras iteraciones podrían incorporar machine learning para predecir rutas óptimas en tiempo real o usar contenedores (Docker) para simular entornos complejos.

---

4. Recomendaciones Finales

1. **Automatización de mediciones**:
   Implementar scripts o herramientas (ej: Prometheus) para monitorear latencia y ancho de banda automáticamente, reduciendo errores humanos.
2. **Pruebas de estrés**:
   Evaluar el sistema bajo condiciones de alta demanda para identificar cuellos de botella no detectados en las pruebas iniciales.
3. **Capacitación continua**:
   Organizar talleres sobre tecnologías clave (Tailscale, Qt para GUI) para equipos técnicos.