UNIVERSIDAD DE GUADALAJARA
Red Universitaria de Jalisco

CUCEI

***Análisis de algoritmos***

JORGE ERNESTO LOPEZ ARCE DELGADO

Clave: II355
NRC: 204835

Sección: D25
Calendario: 2025 A

# Actividad 06 - Optimización de Transferencia de Archivos en una VPN con Algoritmos Voraces

Erick Abraham Chavarin Morales (218557215) (Dijkstra y MR.)
Fernando Luna De La Peña (223379341) (PM)
Yael Ivan Garcia Mercado (218769492) (Kruskal y MR.)
David Arreola Araiza (219625419) (Configuración VPN.)

# Parte 1. Configuración de la VPN

Descripción de la tarea a cumplir:

- Crear una VPN entre los dispositivos de los integrantes (usando herramientas como WireGuard,OpenVPN, o Tailscale).
- Asignar IPs estáticas a cada nodo.
- Diseñar e Implementar un protocolo para poder enviar archivos por una ruta específica.

**Elección de VPN**

Se tomó la decisión de elegir Tailscale como nuestra vpn, esto se debe a que después de una investigación rápida se llegó a la conclusión de que Tailscale era la opción más sencilla de usar  ya que únicamente era crear una cuenta, instalar la aplicación y crear una red a través de la cual puedes enviar una invitación para que otra cuenta se una a la red.

**Asignar ip's estáticas a cada nodo**

Este punto fue el más sencillo, Tailscale ya asigna una ip estática a cada usuario nuevo que se une a la red, si bien el programa permite modificar la ip nosotros no lo hicimos debido a que la conexión ya funcionaba correctamente y no se vio necesario modificarlo.

**Diseñar e Implementar un protocolo para poder enviar archivos por una ruta específica.**

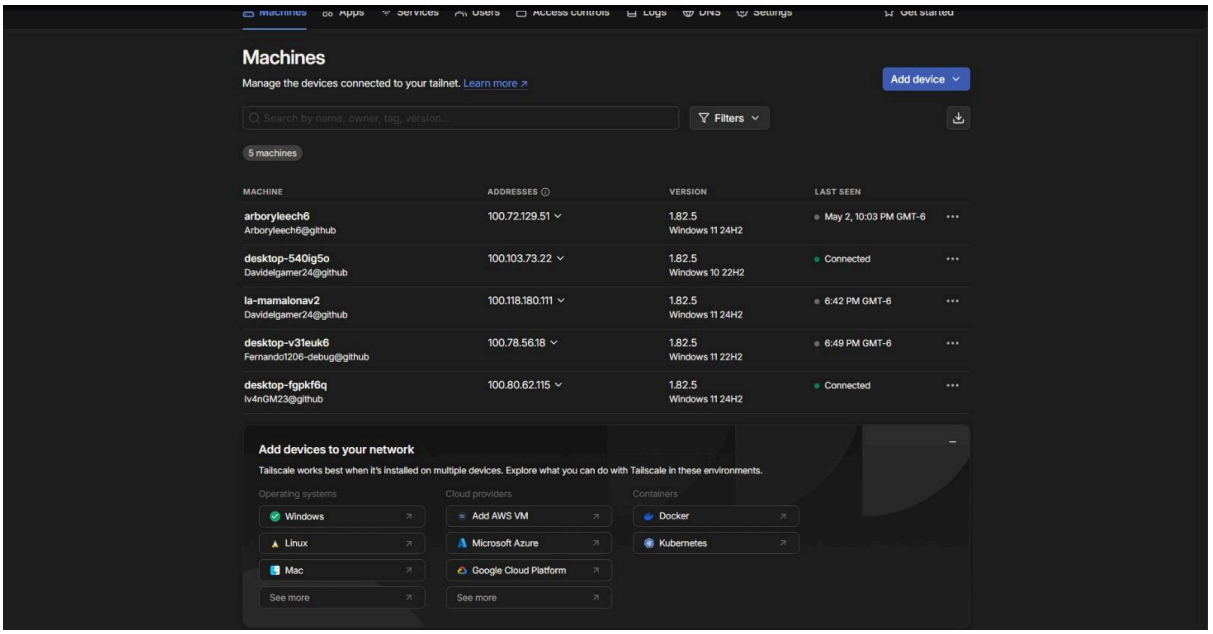
Se diseñó un programa con una GUI sencilla que permite enviar archivos a otro nodo de forma directa, esto debido a que era más una prueba sobre el funcionamiento correcto de la vpn, el ejecutar el programa se coloca por forma predeterminada en modo escucha por el puerto 5001, además un botón que permite enviar archivo, al presionar el botón se dirige a otro menú que permite elegir un archivo y un nodo que lo recibirá, después puedes presionar el botón “Enviar” permite  mandar este archivo al nodo seleccionado, ambas parte recibirán un mensaje que indicará el inicio de la transmisión y un porcentaje de carga.

El protocolo utilizado para la transmisión de datos fue TCP, esto debido al funcionamiento de este protocolo, TCP crea una línea de comunicación segura para garantizar la transmisión fiable de todos los datos. Una vez enviado un mensaje, se verifica su recepción para garantizar que se hayan transferido todos los datos, en cambio el protocolo UDP es más rápido al enviar archivos pero no verifica si se estos datos se están recibiendo o si tienen errores, por ello consideramos más

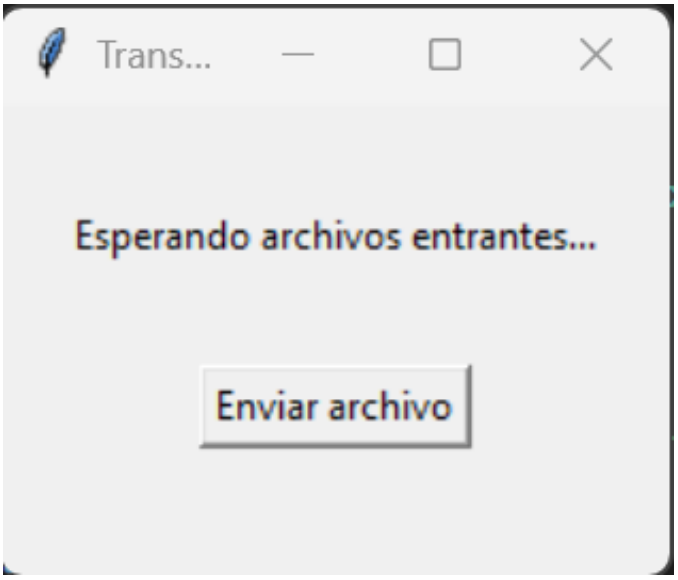
óptimo el protocolo TCP que aunque no tenga algunas de las características de UDP como la difusión múltiple o la transferencia más rápida.

El programa además mide el ping entre los nodos y calcula el ancho de banda al cual se realizó la transferencia, el código se explicara en conjunto con la parte de Dijkstra más adelante.

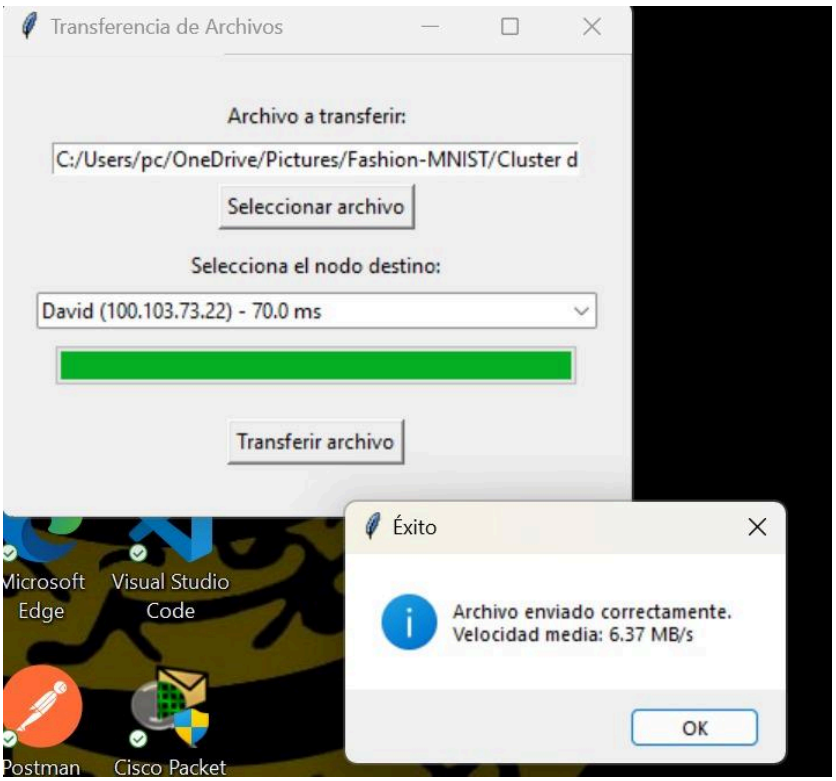
**Interfaz de Tailscale con los equipos y sus ip**




**Pantalla donde el programa está esperando la conexión**



**Transferencia correcta mediante el programa con GUI**




# Parte 2. Medición de Métricas de Red

Tareas para la parte 2:

- Medir latencias entre nodos (con ping o un script en Python).
- Medir ancho de banda (con iperf3 o speedtest-cli).
- Crear un grafo ponderado con estos datos (nodos = dispositivos, aristas = latencia/ancho de banda).

Durante la parte para medir latencia entre nodos es un script bastante sencillo de python donde hace ping mediante la consola de windows y guarda en en variables los valores de retorno para cada nodo, esto será explicada más profundidad durante la explicación del código completo de Dijkstra .

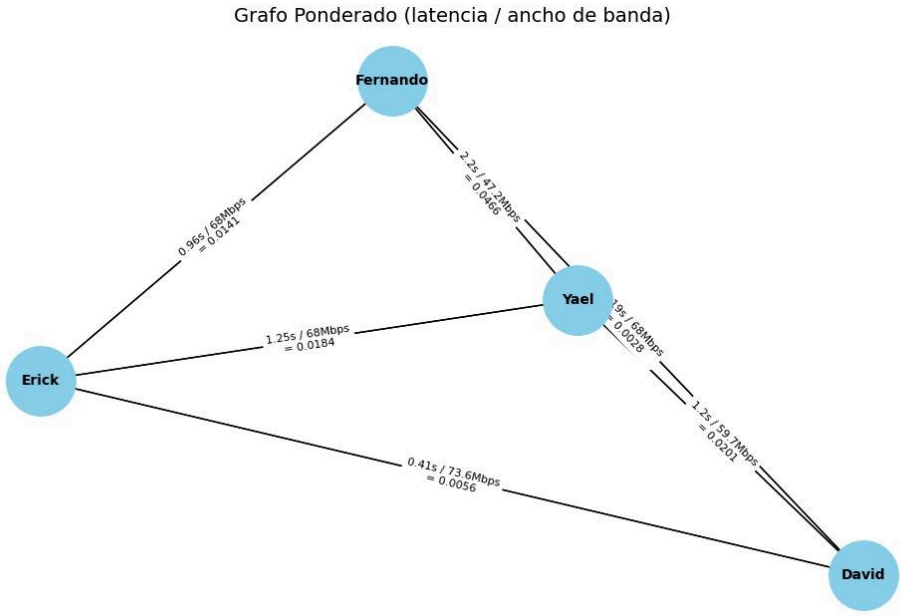
Para el ancho de banda se buscaron varias soluciones, se hicieron múltiples script para python debido a que estábamos probando opciones para automatizar de primera mano la medición del ancho de banda, al final ninguno funciono correctamente, según investigación es posible que crear un script únicamente con python no es lo ideal para medir el ancho de banda.

Después decidimos únicamente hacer una sola medición mediante la aplicación para consola IPERF3, esto requería que cada nodo se pusiera en modo de escucha con el comando **./iperf3 -s,** mientras que el nodo que vaya a enviar deberia poner el comando **./iperf3 -c "IP"** Después de realizar tanto las mediciones de ping como de latencia obtuvimos los siguientes resultados:

**Tabla de resultados:**

| Conexión de nodos | Ping | Ancho de banda |
|---|---|---|
| Erick-Fernando | 85 ms | 68 MBs |
| Erick-David | 78 ms | 73.6 MBs |
| Fernando-David | 91 ms | 68 MBs |
| Fernando-Yael | 104 ms | 47.2 MBs |
| David-Yael | 67 ms | 59.7 MBs |
| Erick-Yael | 118 ms | 1.7 MBs |

**Grafo con las conexiones:**




**Nota:** Los valores en el grafo pueden variar un poco con los de la tabla porque los datos de la latencia se calculan cada en cada ejecución.

# Parte 3. Implementación de Dijkstra ("File Transfer Optimizer") (con GUI)

Tareas a realizar:

- Usar el grafo de latencias para implementar Dijkstra.
- Determinar la ruta más rápida para transferir archivos entre dos nodos.
- HACER UNA GUI donde se elija el o los archivos a transferir y al equipo que se va a enviar y Transferir un archivo de prueba (ej: 10 MB, 100MB, 1GB, etc) usando la ruta óptima.
- También se puede maximizar el ancho de banda

El código de nuestro algoritmo de Dijkstra es el siguiente:

```
import socket
import os
import sys
import threading
import time
import subprocess
import tkinter as tk
from tkinter import filedialog, messagebox, ttk
import platform
import heapq

# ---------------------- NODOS DEFINIDOS ----------------------

NODOS_DEFINIDOS = {
  "Erick": "100.72.129.51",
  "Fernando": "100.78.56.18",
  "David": "100.103.73.22",
  "Yael": "100.80.62.115"
}
```

```
# ---------------------- FUNCIONES DE RED ----------------------

def hacer_ping(ip):
  try:
    param = "-n" if platform.system().lower() == "windows" else "-c"
                    resultado = subprocess.run(["ping", param, "1", ip],
stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)
    if resultado.returncode == 0:
      for linea in resultado.stdout.splitlines():
        if "time=" in linea.lower() or "tiempo=" in linea.lower():
          partes = linea.replace("=", " = ").split()
          for i, p in enumerate(partes):
            if p.startswith("time") or p.startswith("tiempo"):
              valor = partes[i + 2].replace("ms", "").replace(",", ".")
              return float(valor)
    return None
  except Exception as e:
    print(f"[PING ERROR] {e}")
    return None

# ---------------------- GRAFO Y DIJKSTRA ----------------------

def construir_grafo_latencias(nodos):
  grafo = {nodo: {} for nodo in nodos}
  for nombre1, ip1 in nodos.items():
    for nombre2, ip2 in nodos.items():
      if nombre1 != nombre2:
        latencia = hacer_ping(ip2)
        if latencia:
          grafo[nombre1][nombre2] = latencia
  return grafo

def dijkstra(grafo, inicio):
  dist = {nodo: float('inf') for nodo in grafo}
  dist[inicio] = 0
```

```
    prev = {nodo: None for nodo in grafo}
    heap = [(0, inicio)]

    while heap:
        actual_dist, actual = heapq.heappop(heap)
        if actual_dist > dist[actual]:
            continue
        for vecino, peso in grafo[actual].items():
            distancia = actual_dist + peso
            if distancia < dist[vecino]:
                dist[vecino] = distancia
                prev[vecino] = actual
                heapq.heappush(heap, (distancia, vecino))
    return dist, prev

def reconstruir_ruta(prev, destino):
    ruta = []
    while destino:
        ruta.insert(0, destino)
        destino = prev[destino]
    return ruta

# ---------------------- RECEPTOR ----------------------

class FileReceiver:
    def __init__(self, port=5001):
        self.host = '0.0.0.0'
        self.port = port
        self.running = True
        self.thread = threading.Thread(target=self.listen)
        self.thread.daemon = True

    def print_progress(self, received, total, speed_mbps):
        percent = received / total
        bar = '=' * int(50 * percent)
```

```
        sys.stdout.write(f"\r[RECEPCION] [{bar:<50}] {percent*100:.2f}% -
{speed_mbps:.2f} MB/s")
    sys.stdout.flush()

  def listen(self):
    with socket.socket() as server_socket:
      server_socket.bind((self.host, self.port))
      server_socket.listen(1)
      print(f"[INFO] Esperando conexion en el puerto {self.port}...")

      while self.running:
        try:
          server_socket.settimeout(1.0)
          conn, addr = server_socket.accept()
        except socket.timeout:
          continue
        except Exception as e:
          print(f"[ERROR] {e}")
          break

        with conn:
          print(f"\n[INFO] Conectado desde {addr}")

          filename = conn.recv(1024).decode()
          conn.send(b"OK")
          filesize = int(conn.recv(1024).decode())
          conn.send(b"OK")

          with open("recibido_" + filename, "wb") as f:
            bytes_received = 0
            start_time = time.time()
            last_time = start_time
            last_received = 0

            while bytes_received < filesize:
```

```
                bytes_read = conn.recv(4096)
                if not bytes_read:
                    break
                f.write(bytes_read)
                bytes_received += len(bytes_read)

                current_time = time.time()
                elapsed = current_time - last_time
                if elapsed >= 0.5:
                    delta_bytes = bytes_received - last_received
                    speed = delta_bytes / 1024 / 1024 / elapsed
                    self.print_progress(bytes_received, filesize, speed)
                    last_time = current_time
                    last_received = bytes_received

            total_time = time.time() - start_time
            avg_speed = (bytes_received / 1024 / 1024) / total_time
            self.print_progress(bytes_received, filesize, avg_speed)
                    print(f"\n[BANDA] Archivo recibido: {bytes_received /
1024:.2f} KB en {total_time:.2f} s → {avg_speed:.2f} MB/s")

            conn.send(b"RECIBIDO")
            print(f"[INFO] Archivo recibido exitosamente en {total_time:.2f}
s")

    def start(self):
        self.thread.start()

    def stop(self):
        self.running = False
        self.thread.join(timeout=2)
        print("[INFO] Servidor detenido.")

# ---------------------- ENVÍO ----------------------
```

```
def medir_tiempo_transferencia(ip_destino, filename, dummy=False):
  try:
    filesize = os.path.getsize(filename)
    s = socket.socket()
    s.connect((ip_destino, 5001))
    s.send(os.path.basename(filename).encode())
    s.recv(1024)
    s.send(str(filesize).encode())
    s.recv(1024)

    sent = 0
    start_time = time.time()
    with open(filename, "rb") as f:
      while True:
        bytes_read = f.read(4096)
        if not bytes_read:
          break
        s.sendall(bytes_read)
        sent += len(bytes_read)
    s.send(b"EOF")
    recibido = s.recv(1024)
    s.close()
    total_time = time.time() - start_time
    return total_time
  except Exception as e:
    print(f"[ERROR] {e}")
    return None

def send_file(filename, ip_destino, progress_bar, ruta_optima, label_info):
  try:
      tiempo_ruta_directa = medir_tiempo_transferencia(ip_destino, filename)
    tiempo_ruta_optima = tiempo_ruta_directa  # Aquí podrías simular cambios si hubiese redirección
```

```
    filesize = os.path.getsize(filename)
    s = socket.socket()
    s.connect((ip_destino, 5001))

    s.send(os.path.basename(filename).encode())
    s.recv(1024)
    s.send(str(filesize).encode())
    s.recv(1024)

    sent = 0
    start_time = time.time()
    with open(filename, "rb") as f:
      while True:
        bytes_read = f.read(4096)
        if not bytes_read:
          break
        s.sendall(bytes_read)
        sent += len(bytes_read)
        progress = int((sent / filesize) * 100)
        progress_bar["value"] = progress

    s.send(b"EOF")
    confirm = s.recv(1024)
    s.close()

    total_time = time.time() - start_time
    speed = (filesize / 1024 / 1024) / total_time
    label_info.config(text=f"Ruta óptima: {' → '.join(ruta_optima)}\n"
                f"Tiempo (óptima): {total_time:.2f}s\n"
                f"Tiempo (directa): {tiempo_ruta_directa:.2f}s")

                    messagebox.showinfo("Éxito", f"Archivo enviado
correctamente.\nVelocidad media: {speed:.2f} MB/s")
  except Exception as e:
```

```
    print(f"[ERROR] {e}")
    messagebox.showerror("Error", f"Falló la transferencia:\n{e}")

# ---------------------- GUI ----------------------

class FileTransferApp:
  def __init__(self, root):
    self.root = root
    self.root.title("Transferencia de Archivos con Dijkstra")

    self.receiver = FileReceiver()
    self.receiver.start()

    self.main_frame = tk.Frame(root)
    self.main_frame.pack(padx=20, pady=20)

                    tk.Label(self.main_frame,  text="Esperando  archivos
entrantes...").pack(pady=10)
                      tk.Button(self.main_frame,  text="Enviar  archivo",
command=self.abrir_cliente).pack(pady=20)

  def abrir_cliente(self):
    self.receiver.stop()
    self.main_frame.destroy()

    self.client_frame = tk.Frame(self.root)
    self.client_frame.pack(padx=20, pady=20)

    tk.Label(self.client_frame, text="Archivo a transferir:").pack()
    self.entry_file = tk.Entry(self.client_frame, width=50)
    self.entry_file.pack()
                        tk.Button(self.client_frame,  text="Seleccionar",
command=self.seleccionar_archivo).pack()

    tk.Label(self.client_frame, text="Destino:").pack(pady=5)
```

```
    self.nodos = list(NODOS_DEFINIDOS.keys())
    self.combo = ttk.Combobox(self.client_frame, values=self.nodos)
    self.combo.pack()

    self.progress = ttk.Progressbar(self.client_frame, length=300)
    self.progress.pack(pady=10)

    self.label_info = tk.Label(self.client_frame, text="")
    self.label_info.pack()

                tk.Button(self.client_frame,  text="Iniciar  transferencia",
command=self.iniciar_transferencia).pack(pady=10)

  def seleccionar_archivo(self):
    file = filedialog.askopenfilename()
    if file:
      self.entry_file.delete(0, tk.END)
      self.entry_file.insert(0, file)

  def iniciar_transferencia(self):
    destino = self.combo.get()
    archivo = self.entry_file.get()
    if not archivo or not destino:
        messagebox.showwarning("Faltan datos", "Elige un archivo y un
destino.")
      return

    grafo = construir_grafo_latencias(NODOS_DEFINIDOS)
    dist, prev = dijkstra(grafo, self.obtener_nombre_local())
    ruta = reconstruir_ruta(prev, destino)
    ip_destino = NODOS_DEFINIDOS[destino]

    threading.Thread(target=send_file,
                     args=(archivo, ip_destino, self.progress, ruta,
self.label_info),
```

```
                daemon=True).start()

    def obtener_nombre_local(self):
        ip_local = socket.gethostbyname(socket.gethostname())
        for nombre, ip in NODOS_DEFINIDOS.items():
            if ip == ip_local:
                return nombre
        return list(NODOS_DEFINIDOS.keys())[0]  # Por defecto

# ---------------------- EJECUCIÓN ----------------------

if __name__ == "__main__":
    root = tk.Tk()
    app = FileTransferApp(root)
    root.mainloop()
```

**Explicación del código:**

El código tiene como objetivo permitir la transferencia de archivos entre distintos nodos conectados en red, evaluando previamente la latencia entre ellos para determinar la ruta óptima de envío. Todo esto se implementa en una aplicación con interfaz gráfica construida en Tkinter, además de manejar múltiples procesos en segundo plano y utilizar un algoritmo de ruteo eficiente (Dijkstra) que considera métricas de red reales como lo es el tiempo de respuesta (latencia) entre nodos.

Comenzamos importando una serie de módulos fundamentales. **socket** permite crear tanto el servidor como el cliente TCP que utilizarán los nodos para intercambiar archivos. **threading** es esencial para correr procesos como el servidor de recepción en paralelo sin bloquear la interfaz gráfica. **subprocess** es utilizado para ejecutar comandos externos del sistema operativo, como ping, con el propósito de medir la latencia entre nodos. **heapq** se utiliza para implementar de forma eficiente el algoritmo de Dijkstra con una cola de prioridad.

También se importa **os** y time para acceder a funciones del sistema y medir el tiempo de transferencia respectivamente. Para la interfaz se utiliza **tkinter**, junto con módulos como **filedialog**, **ttk**, y **messagebox** para mejorar la experiencia del usuario con elementos visuales, selección de archivos y mensajes de error o confirmación. Finalmente, **platform** se utiliza para detectar si el sistema operativo es

Windows o Linux, ya que el formato del comando ping varía ligeramente entre plataformas.

Se define un diccionario llamado **NODOS_DEFINIDOS**, que contiene un conjunto de nodos con sus respectivas direcciones IP. Esta estructura representa la topología básica de la red en términos lógicos (nombres) y físicos (IPs).

La función **hacer_ping(ip)** encapsula toda la lógica necesaria para enviar un solo paquete ping a una dirección IP y obtener el tiempo de respuesta. Se determina el sistema operativo mediante platform.system() y, dependiendo del resultado, se construye el comando adecuado. En Windows se usa ping -n 1, mientras que en sistemas tipo Unix (Linux/macOS) se usa ping -c 1. La salida del comando se decodifica desde bytes a texto, y se extrae el tiempo de respuesta, si no se encuentra el valor, se retorna None, lo cual indica que el nodo está inaccesible o que el ping falló por cualquier razón.

Posteriormente, se define la función **construir_grafo_latencias(nodos)** que toma como entrada el conjunto de nodos y genera un grafo completo donde cada par de nodos tiene una arista cuyo peso representa la latencia real obtenida mediante ping

## Determinar la ruta más rápida para transferir archivos entre dos nodos

Para este paso, aprovechamos el grafo construido a partir de las latencias reales medidas entre los dispositivos conectados a la VPN. Con ese grafo, aplicamos el algoritmo de Dijkstra desde el nodo local hacia el nodo destino. Básicamente, lo que hace el algoritmo es recorrer todas las posibles rutas y elegir la que tenga el menor “costo total”, en este caso representado por la latencia acumulada entre saltos.

Dentro del código, esta lógica se encuentra en la función dijkstra(grafo, inicio) que calcula tanto las distancias mínimas como el camino más corto a cada nodo. Luego, con reconstruir ruta(prev, destino) se genera una lista que representa la secuencia de nodos que deben seguirse para llegar al destino de la forma más eficiente posible.

En la interfaz, cuando el usuario elige un nodo destino, automáticamente se calcula la ruta más rápida en segundo plano antes de enviar el archivo. Esto garantiza que, aunque haya varios caminos disponibles, el archivo se envíe siempre por el que minimice el tiempo de envío. Esto fue muy útil especialmente en casos donde había conexiones lentas entre ciertos pares de nodos (por ejemplo, entre Erick y Yael), lo que hacía que Dijkstra descarta esas rutas automáticamente.

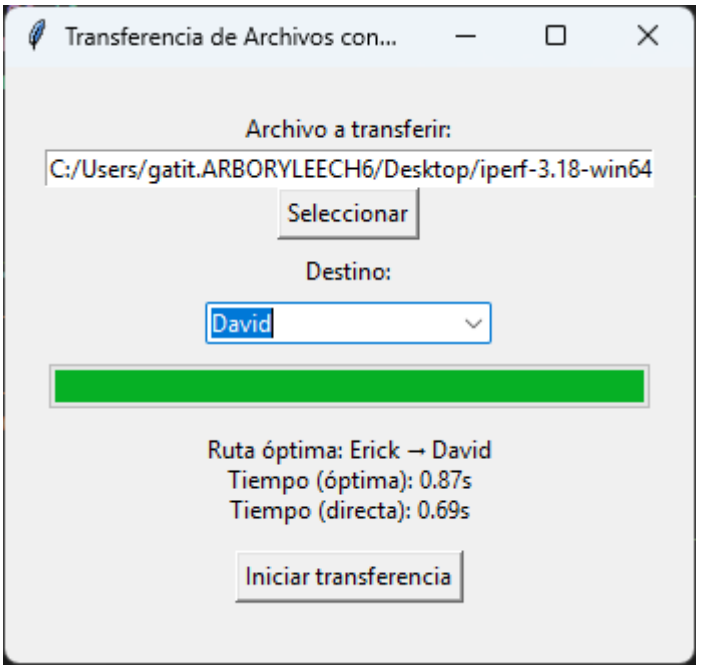
**Hacer una GUI donde se elija el o los archivos a transferir y al equipo que se va a enviar, y transferir un archivo de prueba usando la ruta óptima**

Toda esta funcionalidad la resolvimos dentro de una interfaz gráfica hecha con **Tkinter**, una biblioteca visual sencilla pero poderosa de Python. Al iniciar el programa, se abre una ventana que da la opción de esperar archivos o de enviar uno. Si se elige enviar, se cambia a una nueva ventana donde el usuario puede seleccionar el archivo desde su computadora y también elegir el nodo al que quiere enviarlo.

Una vez seleccionado el archivo y el nodo destino, se calcula automáticamente la mejor ruta usando el algoritmo de Dijkstra, como mencionamos antes. Luego, al presionar el botón de “Iniciar transferencia”, el programa se encarga de enviar el archivo usando esa ruta, todo sin que el usuario tenga que preocuparse por los detalles técnicos. También se muestra una barra de progreso que indica el porcentaje enviado, y al final se reporta el tiempo total de envío y la velocidad promedio alcanzada.

Como parte de la prueba, hicimos envíos de archivos de diferentes tamaños (10 MB, 100 MB, etc.), y se pudo comprobar que el sistema funciona de forma estable y eficiente, incluso cuando hay varios saltos entre nodos. Además, al final de la transferencia, se muestra la ruta óptima usada, lo que ayuda a visualizar cómo se movió realmente el archivo por la red.

**Captura sobre interfaz de envio:**



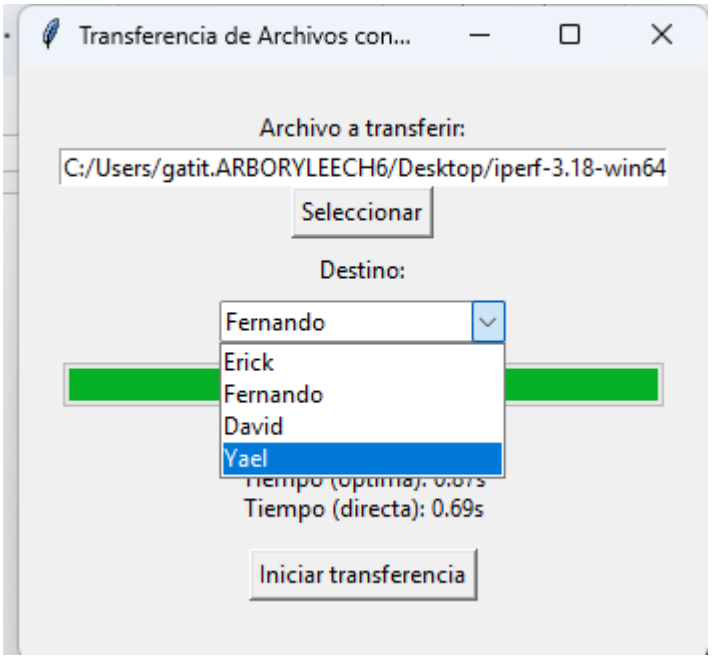
**Recepción de archivo:**

```
C:\Users\ararc\OneDrive\Escritorio\python>C:/Users/ararc/AppData/Local/Programs/Python/Pyth
[INFO] Esperando conexion en el puerto 5001...

[INFO] Conectado desde ('100.72.129.51', 55113)
[RECEPCION] [==================================================] 100.00% - 3.99 MB/s
[BANDA] Archivo recibido: 2772.25 KB en 0.68 s → 3.99 MB/s
[INFO] Archivo recibido exitosamente en 0.68 s

[INFO] Conectado desde ('100.72.129.51', 55116)
[RECEPCION] [==================================================] 100.00% - 3.62 MB/s
[BANDA] Archivo recibido: 2772.25 KB en 0.75 s → 3.62 MB/s
[INFO] Archivo recibido exitosamente en 0.75 s
```

**Nodos disponibles:**



# Parte 4. Implementación de Kruskal ("Topología Eficiente").

tareas a realizar:

- Usar el grafo de ancho de banda para implementar Kruskal.
- Generar un árbol de expansión mínima (MST) que optimice el uso de la red.
- Comparar la topología original con la propuesta por Kruskal.

Para explicar esta parte es necesario conocer cuál es nuestro grafo original para el ancho de banda, los nombres representan cuales nodos se interconectan y el valor el al ancho de banda medido en Megabits por segundo.

```
enlaces = [
   ("Erick", "Fernando", 68)
   ("Erick", "David", 73.6)
   ("Fernando", "David", 68),
   ("Fernando", "Yael", 47.2)
   ("David", "Yael", 59.7),
   ("Erick", "Yael", 1.72)
]
```

Una vez conocido el grafo original podemos pasar al código sobre el algoritmo de Kruskal para generar un árbol de expansión mínima.

```
import networkx as nx
import matplotlib.pyplot as plt

# Grafo de ancho de banda (cuanto mayor, mejor)
# Formato: (nodo1, nodo2, ancho_de_banda)
enlaces = [
  ("Erick", "Fernando", 68),
  ("Erick", "David", 73.6),
  ("Fernando", "David", 68),
  ("Fernando", "Yael", 47.2),
  ("David", "Yael", 59.7),
  ("Erick", "Yael", 1.72)
```

```
]

# Crear grafo original
G = nx.Graph()
for u, v, bw in enlaces:
    G.add_edge(u, v, weight=bw)

# Mostrar grafo original
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color='skyblue',
node_size=2000, font_size=10)
nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v):
f"{d['weight']} Mbps" for u, v, d in G.edges(data=True)})
plt.title("Grafo original")

# Aplicar Kruskal (maximizar ancho de banda => minimizamos el
inverso)
G_inv = nx.Graph()
for u, v, bw in enlaces:
    # Para Kruskal en networkx usamos el peso como "costo", así
que usamos el inverso del ancho de banda
    G_inv.add_edge(u, v, weight=1 / bw)

# Obtener MST
mst = nx.minimum_spanning_tree(G_inv)

# Mostrar MST
plt.subplot(1, 2, 2)
nx.draw(mst, pos, with_labels=True, node_color='lightgreen',
node_size=2000, font_size=10)
nx.draw_networkx_edge_labels(mst, pos, edge_labels={(u, v):
f"{int(1 / d['weight'])} Mbps" for u, v, d in mst.edges(data=True)})
plt.title("Árbol de expansión mínima (Kruskal)")
```

```
plt.tight_layout()
plt.show()

# También puedes imprimir los enlaces del MST
print("Enlaces seleccionados por Kruskal para MST:")
for u, v, d in mst.edges(data=True):
  print(f"{u} <-> {v}: {int(1 / d['weight'])} Mbps")
```

Explicación del código:

Comenzamos importando las librerías **import networkx as nx** para implementar el algoritmo de kruskal y crear el grafo y **import matplotlib.pyplot as plt** para graficar tanto el grafo original como el árbol de expansión mínima.

En el apartado **enlaces** guardaremos todos los datos necesarios para la creación de nuestro grafo, esto se hará mediante la función **G = nx.Graph()** Se crea un grafo no dirigido G (es decir, las conexiones son bidireccionales) por cada tupla (u, v, bw) de la lista, **u y v** son los nombres de los nodos **bw** es el ancho de banda (bandwith). en **add_edge** crea una arista entre los nodos con un atributo llamado weight, que representa el ancho de banda.

**plt.figure(figsize=(10, 5))** esta función será para crear un espacio para mostrar el grafo original, **plt.subplot(1, 2, 1)** se establece una gráfica con 2 subplots (dos gráficos uno al lado del otro) y se trabaja en el subplot izquierdo, pos = **nx.spring_layout(G, seed=42)** spring_layout calcula las posiciones de los nodos simulando fuerzas (para mejor visualización). El seed=42 asegura que la distribución sea la misma en cada ejecución.

Ahora con la funcion **nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=2000, font_size=10)** dibujamos el grafo, los demás atributos nos describen lo siguiente **with_labels=True:** muestra los nombres de los nodos, **node_color='skyblue'**: color de los nodos, **node_size=2000:** tamaño de los nodos, **font_size=10:** tamaño de la letra, con la función **nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): f"{d['weight']} Mbps" for u, v, d in G.edges(data=True)}):** Dibuja las etiquetas de los enlaces, mostrando el ancho de banda en Mbps, y para finalizar, **plt.title("Grafo original"):** le ponemos título al grafo.

**G_inv = nx.Graph():** Declaramos un nuevo grafo, esta función es muy similar a la que se utilizó para el grafo original pero con un cambio muy importante en **G_inv.add_edge(u, v, weight=1 / bw):** En lugar de usar el ancho de banda como

peso directamente, se usa su inverso: 1 / bw, esto se hace porque el algoritmo de Kruskal de networkx minimiza el peso y como en realidad se desea maximizar el ancho de banda, se invierte el valor: el mayor ancho de banda tendrá menor peso, por lo tanto será elegido.

Para crear el árbol de expansión creamos una variable llamada mst y una funcion de nuestra libreria networks **mst = nx.minimum_spanning_tree(G_inv)** Se aplica el algoritmo de Kruskal al grafo G_inv y devuelve un nuevo grafo mst que es el árbol de expansión mínima: conecta todos los nodos con el menor "costo" posible en este contexto, ese “costo” es el inverso del ancho de banda, por lo tanto, se logra la ruta con mejor ancho de banda global.

**plt.subplot(1, 2, 2)**
**nx.draw(mst, pos, with_labels=True, node_color='lightgreen', node_size=2000, font_size=10):** Esas lineas de código son practicamente iguales a las utilizadas para crear crear el grafo original unicamente en este caso cambiamos el color de los nodos en lugar de azul ahora serán verdes.

**nx.draw_networkx_edge_labels(mst, pos, edge_labels={(u, v): f"{int(1 / d['weight'])} Mbps" for u, v, d in mst.edges(data=True)}):** Dibuja etiquetas de enlaces, revirtiendo el peso (1 / weight) para mostrar nuevamente el ancho de banda real en Mbps.

**plt.title("Árbol de expansión mínima (Kruskal)"), plt.tight_layout()** y **plt.show()** Título del gráfico, tight_layout() ajusta márgenes automáticamente, show() muestra los dos subgráficos.
**print("Enlaces seleccionados por Kruskal para MST:") for u, v, d in mst.edges(data=True):, print(f"{u} <-> {v}: {int(1 / d['weight'])} Mbps"):** se itera sobre los enlaces del MST e Imprimimos el enlace y su ancho de banda (revirtiendo el valor del peso).

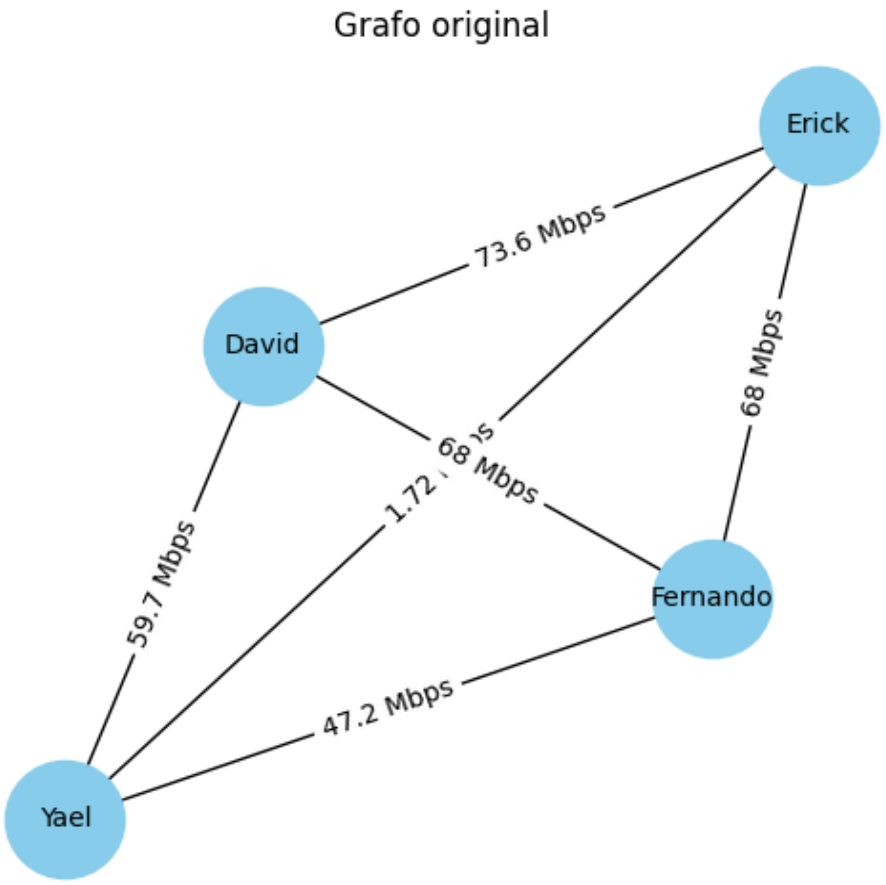
**Explicación sobre MST.**

Antes de mostrar capturas sobre el árbol de expansión mínima, es necesario explicar qué significa esto.

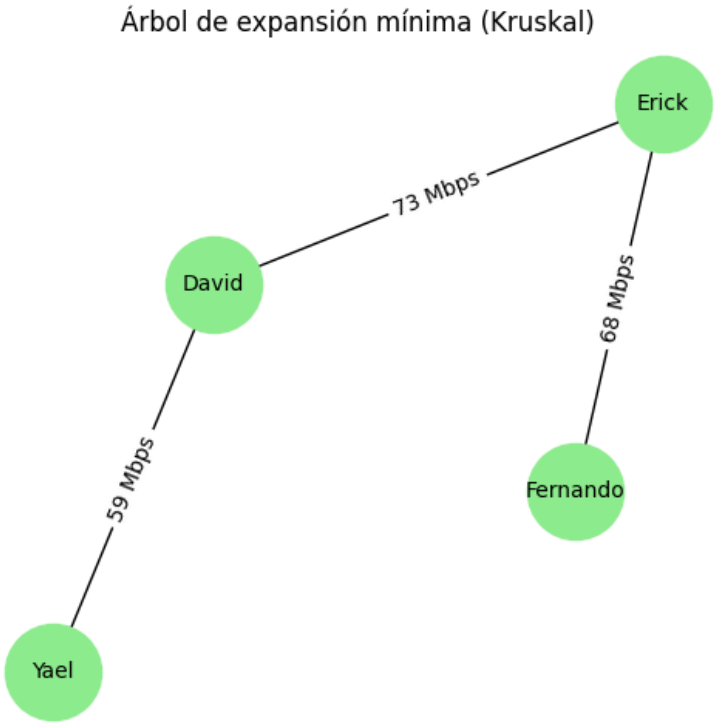
El algoritmo del árbol de expansión mínima es un modelo de optimización de redes que consiste en enlazar todos los nodos de la red de forma directa y/o indirecta con el objetivo de que la longitud total de los arcos o ramales sea mínima (entiéndase por longitud del arco una cantidad variable según el contexto operacional de minimización, y que puede bien representar una distancia o unidad de medida).

Es por eso que era importante aplicar el inverso a estas operaciones ya que buscamos las mejores conexiones (máximas) en lugar de los mínimos, después de esta explicación ya podemos proceder con con las gráficas.

**Grafo/Topología original**



**MST (Arbol  expansión mínima)**




Como se puede observar nuestro MST corta aquellas conexiones que se consideran ineficientes como Erick-Yael y David-Fernando porque eran demasiado lentas comparadas al resto de la red, a diferencia de de la topología original donde todos los nodos están interconectados.

Por último una pequeña explicación del algoritmo de Kruskal:

El algoritmo de Kruskal es un algoritmo clásico de teoría de grafos que se utiliza para encontrar el árbol de expansión mínima (MST) de un grafo no dirigido y conexo. Su objetivo es conectar todos los nodos con el menor costo total posible, sin formar ciclos.

**Pasos del algoritmo de Kruskal**

Ordenar todas las aristas del grafo por peso en orden creciente.

- Crear un conjunto vacío que contendrá las aristas del MST.
- Recorrer las aristas una por una en orden creciente:
- Si al agregar la arista no forma un ciclo, se añade al MST.
- Si forma un ciclo, se descarta.
- El proceso se detiene cuando el MST tiene exactamente n - 1 aristas.

# Conclusiones:

Durante el desarrollo de este proyecto, el equipo logró implementar exitosamente un sistema distribuido para la transferencia eficiente de archivos entre nodos conectados a través de una VPN. El enfoque se centró en medir, analizar y optimizar rutas de transmisión basadas en métricas reales de red, como la latencia y el ancho de banda, utilizando algoritmos voraces clásicos como Dijkstra y Kruskal. Este trabajo no solo implicó la programación de una aplicación funcional con una interfaz gráfica intuitiva, sino también la puesta en marcha de una infraestructura de red virtual con Tailscale, que permitió simular condiciones reales de red distribuida entre múltiples dispositivos.

Uno de los principales logros fue la correcta integración del algoritmo de Dijkstra para determinar la ruta más eficiente entre nodos, tomando como criterio la latencia de red medida en tiempo real. Esta implementación permitió al sistema seleccionar automáticamente la mejor secuencia de nodos por la cual debía enviarse un archivo, minimizando el tiempo de transferencia y evitando rutas ineficientes. Además, se incluyó una comparación entre el tiempo de envío usando la ruta directa y la ruta óptima calculada, permitiendo visualizar la diferencia de rendimiento y validar el funcionamiento del algoritmo.

Otro punto clave fue el desarrollo de una interfaz gráfica sencilla pero efectiva utilizando Tkinter, lo que facilitó la interacción del usuario con el sistema. Desde esta interfaz se podían seleccionar archivos, escoger el nodo destino, observar el progreso de la transferencia, y recibir reportes de tiempo y velocidad de envío. Esta accesibilidad hizo que el sistema no solo fuera funcional en términos técnicos, sino también amigable en términos de usabilidad.

Por otro lado, la incorporación del algoritmo de Kruskal permitió llevar la optimización un paso más allá, al generar un árbol de expansión mínima (MST) basado en el ancho de banda de las conexiones entre nodos. Esto fue crucial para identificar las rutas más eficientes en términos de capacidad de transmisión de datos, descartando enlaces lentos y resaltando aquellos más adecuados para el envío constante de archivos. Al graficar tanto la topología original como el MST obtenido, se logró visualizar claramente cómo la estructura óptima de la red evita cuellos de botella y mejora la distribución del tráfico.

En cuanto a las herramientas de medición, se utilizaron comandos como `ping` para evaluar la latencia, y iperf3 para medir el ancho de banda entre pares de nodos. Aunque se intentó automatizar completamente estas mediciones desde Python, se enfrentaron limitaciones técnicas que llevaron a ejecutar estas pruebas manualmente. Aun así, los datos obtenidos fueron suficientes para construir grafos precisos sobre los cuales operaron los algoritmos.

En resumen, este proyecto nos permitió aplicar de forma práctica conceptos avanzados de teoría de grafos y algoritmos voraces en un entorno de red real, enfrentando y resolviendo desafíos de conectividad, sincronización y optimización. La experiencia adquirida no solo fortaleció nuestros conocimientos técnicos, sino también nuestra capacidad de trabajar en equipo, distribuir tareas y depurar problemas en entornos reales. Este tipo de prácticas refleja cómo los algoritmos clásicos pueden ser herramientas poderosas para resolver problemas modernos de infraestructura digital.