

Proceso de solución implementado de los requerimientos

Como tecnologías para implementar nuestro sistema de transferencia de archivos TCP se utilizó principalmente Python 3.10 como lenguaje de programación de la aplicación, y Docker como herramienta para distribuir y desplegar fácilmente nuestro proyecto en las máquinas virtuales y otros sistemas. El uso de Docker nos ayuda a estandarizar los ambientes de ejecución para que, de esta manera, sean consistentes y universales. De esta manera, desde cualquier sistema operativo solo se necesitan 2 comandos (pull y run) para correr la aplicación sin necesidad de instalaciones adicionales de requerimientos, configuraciones, o dependencias.

Desarrollo del Servidor:

Nosotros desarrollamos el servidor como una aplicación concurrente que utiliza multithreading para el manejo de conexiones simultáneas. En principio, se crea un thread padre que administra y crea nuevos threads por cada conexión que llega al servidor, de esta manera se puede manejar más fácilmente la cantidad de conexiones entrantes.

```
def run(self):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind(ADDR)
    server.listen()
    log.info(f"[PROCESS] Server process ID: {os.getpid()}")
    log.info(f"[LISTENING] Server is listening on {IP}:{PORT}")
    current_clients = list()
    client_id = 1
    threading_condition = threading.Condition()
    while True:
        conn, addr = server.accept()
        client = Client(client_id, threading_condition)
        threaded_client = threading.Thread(target=client.handle_client, args=(conn, addr))
        threaded_client.daemon = True
        threaded_client.start()
        current_clients.append(client)
        client.setClientList(current_clients)
        log.info(f"[ACTIVE CONNECTIONS] {threading.active_count() - 2}")
        client_id += 1
```

En la imagen anterior se puede apreciar la configuración TCP del servidor al crear un socket de tipo SOCK_STREAM, luego se procede a escuchar en el socket y por cada conexión aceptada se crea un nuevo thread con su respectiva configuración.

En esencia, nosotros configuramos dos tipos de conexiones, una main que le pertenece al cliente principal y es quien le indica al servidor qué archivo y a cuántas conexiones debería de esperar y una conexión tipo cliente básico que es la conexión que espera la recepción de un archivo. Para simplificar las cosas, decidimos crear una clase que administre los clientes, comparta atributos y permita realizar diferentes acciones dependiendo del tipo de cliente que sea.

```
def handle_client(self, conn, addr):
    log.info(f"[NEW CONNECTION] {addr} connected.")
    self.connected = True
    self._connection = conn
    self.address = addr
    config_msg = conn.recv(1024).decode(FORMAT)
    if config_msg == "!MAIN_CONN":
        self.send(("Main connection received\n"+MENU))
        log.info(f"[MAIN CONNECTION] {addr} connected.")
        self.conn_type = "main"
    elif "!CONNECTION" in config_msg:
        self.conn_type = "client"
        self.send("OK")
    while self.connected:
        try:
            if self.conn_type == "main":
                self.main_client()
            elif self.conn_type == "client":
                self.normal_client()
        except Exception as e:
            try:
                self.send("Internal Server Error")
            except ConnectionResetError:
                pass
            exception_to_log(log, e)
    self._connection.close()
```

Figura 1. Manejo de Clientes.

Si el cliente es la conexión principal, se le mostrará un menú que le permitirá seleccionar la acción que deseé, ya sea listar los archivos disponibles en el servidor, configurar el archivo a transferir y al número de clientes secundarios, iniciar la transferencia, o desconectarse del servidor. En cuanto a la interfaz con el usuario, desde el cliente se le puede asignar un número de conexiones específico y qué archivo debe mandar para poder realizar el envío. Por tanto, para manejar la concurrencia de dichas conexiones, se decidió usar un “Threading Condition”, la cual nos permitía dejar una conexión (un thread) en espera mientras el resto de las conexiones asignadas al servidor llegaban. Una vez llega la cantidad de conexiones deseadas, el main thread le hace un notify all a todos los threads en espera y envían el archivo.

```

elif msg == "!START":
    if not self.selected_file or self.num_clients == 0:
        self.send("The settings have not been set. You need to configure the server in order to start")
    else:
        self.send("File Transfer Command Started", log_msg=True)
        while (len(self._clientList) <= self.num_clients):
            asyncio.run(asyncio.sleep(1))
        for client in self._clientList:
            if client.conn_type == "client":
                client.selected_file = self.selected_file
        self.condition.acquire()
        self.condition.notify_all()
        self.condition.release()
        self.connected = False # Disconnect the main client

```

Figura 2. Comando de !START, manejo de concurrencia.

Una vez que el servidor recibe el OK desde la conexión del subcliente, el thread se pone a esperar el llamado del thread principal. Después de ser notificado, se calcula el hash del archivo a enviar y se le manda el cliente para comprobar la integridad del mismo. Posteriormente, se le envía el archivo al cliente.

```

def normal_client(self):
    msg = self._connection.recv(1024).decode(FORMAT)
    try:
        self.server_path = "./server/files/"
        [f for f in os.listdir(self.server_path) if os.path.isfile(os.path.join(self.server_path, f))]
    except FileNotFoundError:
        self.server_path = "/home/server/files/"
    if msg == "IDISCONNECT":
        self.connected = False
        self.send("Disconnected from server")
        log.info(f"[DISCONNECTED] {self.address} disconnected.")
    elif msg == "OK":
        self.status = "OK"
        self.condition.acquire()
        self.condition.wait()
        self.condition.release()
        self.send(f"TRANSFER :{self.selected_file}:{os.path.getsize(self.server_path+self.selected_file)}")
        file_hash = hash_file(self.server_path+self.selected_file)
        with open(self.server_path+self.selected_file, "rb") as f:
            data = f.read()
        self.send(f"{file_hash}")
        try:
            t1 = time.time()
            self._connection.sendall(data)
            t2 = time.time()
            log.info(f"[FILE TRANSFER] - File {self.selected_file} sent to {self.address} [CLIENT {self.connection_id}] in {t2-t1} seconds")
            self.connected = False
            log.info("File successfully sent to Client " + str(self.connection_id))
        except (ConnectionResetError, ConnectionAbortedError, ConnectionRefusedError):
            log.error("Error sending file to Client " + str(self.connection_id))

```

Figura 3. Manejo de subcliente.

Durante una transferencia, los subclientes se conectarán al servidor para posteriormente mostrar una barra de progreso de descarga de archivos, finalizando con la muestra de información sobre la transferencia como estado, tiempo, y verificación de integridad por medio del cálculo del hash del archivo.

Desarrollo del Cliente:

Para el desarrollo del cliente se utilizó una arquitectura bastante similar a la del servidor, se crea una conexión principal y esta le indica al servidor qué archivo enviar y a cuántas conexiones. Una vez se configura y se le da la orden de empezar, el cliente genera X conexiones diferentes

que le avisan al servidor que están listas para recibir el archivo (donde X es la cantidad especificada por el cliente).

```

while connected:
    msg = input("[MENU] Enter a command: ")
    if not msg in COMMANDS and not "!CONFIG" in msg:
        log.info(f"[MENU] Invalid command {msg}")
        continue
    client.send(msg.encode(FORMAT))
    msg_rcv = client.recv(1024).decode(FORMAT)
    log.info(f"[SERVER] {msg_rcv}")
    if msg == "!DISCONNECT":
        connected = False
    elif "!CONFIG" in msg and "Config set" in msg_rcv:
        num_clients = int(msg.split(':')[2])
    if msg_rcv == "File Transfer Command Started":
        for i in range(num_clients):
            client = threading.Thread(target=connect_client, args=(i+1,))
            threads.append(client)
            client.start()
        connected = False

```

Figura 4. Estructura de Cliente Principal.

```

elif msg == "OK":
    log.info(f"[CLIENT #{client_num}] Connection established waiting for file transfer")
    client.send("OK".encode(FORMAT))
    msg = client.recv(1024).decode(FORMAT)
    if "TRANSFER" in msg:
        hash = client.recv(1024).decode(FORMAT)
        file_data = msg.split(":")
        filename = f"File {client_num} " + file_data[1].rstrip()
        filesize = int(file_data[2])
        log.info(f"[CLIENT #{client_num}] Receiving file {filename} with size {filesize}")
        bar = tqdm(range(filesize), f"Receiving {filename}", unit="B", unit_scale=True, unit_divisor=1024)
        try:
            os.makedirs('./client/ArchivosRecibidos')
        except FileExistsError:
            pass
    global num_clients

    with open(f"./client/ArchivosRecibidos/Cliente{client_num}-Prueba{num_clients}.mp4", "wb") as f:
        try:
            t1 = time.time()
            while True:
                data = client.recv(1024)
                if len(data) <= 0:
                    bar.close()
                    t2 = time.time()
                    log.info(f"[CLIENT #{client_num}] File transfer time: {t2-t1} seconds")
                    break
                bar.update(len(data))
                f.write(data)
        except (ConnectionResetError, ConnectionAbortedError, ConnectionRefusedError):
            log.error("ERROR: Connection lost")
            connected = False
    log.info(f"[CLIENT #{client_num}] File transfer complete")
    clientHash = hash_file(f"./client/ArchivosRecibidos/Cliente{client_num}-Prueba{num_clients}.mp4")
    if hash == clientHash:
        log.info(f"[CLIENT #{client_num}] File transfer successful, integrity check passed - Hashes are equal.")
        log.info(f"File Size: {filesize} bytes")
        log.info(f"Server Hash: {hash}")
        log.info(f"Client Hash: {clientHash}")
    else:
        log.info(f"[CLIENT #{client_num}] File transfer successful, integrity check failed - Hashes are not equal.")
        log.info(f"File Size: {filesize} bytes")
        log.info(f"Server Hash: {hash}")
        log.info(f"Client Hash: {clientHash}")

```

Figura 5. Recibida de OK del server, recepción del archivo y cálculo de integridad.

Puesta en marcha del sistema

```
→ docker run --platform linux/amd64 -i ghcr.io/communications-infrastructure/simple-file-transfer-system-client:main
2022-10-01 17:42:00 INFO    client [STARTING] Client is starting...
2022-10-01 17:42:00 INFO    client Local server is not running, please enter the new IP address of the server:
172.17.0.6
2022-10-01 17:42:07 INFO    client [CONNECTED] Main Client - Connected to 172.17.0.6:6969
2022-10-01 17:42:07 INFO    client [MENU] Main connection received
2022-10-01 17:42:07 INFO    client [MENU] Server Commands:
2022-10-01 17:42:07 INFO    client [MENU] !LIST - List all the available files
2022-10-01 17:42:07 INFO    client [MENU] !CONFIG - Set up the server file transfer configuration
2022-10-01 17:42:07 INFO    client [MENU] !START - Start the file transfer to all clients
2022-10-01 17:42:07 INFO    client [MENU] !DISCONNECT - Disconnect from the server
2022-10-01 17:42:07 INFO    client [MENU]
2022-10-01 17:42:07 INFO    client [MENU] Enter a command:
!LIST
2022-10-01 17:42:15 INFO    client [SERVER] ['100mb.mp4', '250mb.mp4']
2022-10-01 17:42:15 INFO    client [MENU] Enter a command:
!CONFIG :100mb.mp4 :1
2022-10-01 17:42:34 INFO    client [SERVER] Config set to 100mb.mp4 and 1 clients
2022-10-01 17:42:34 INFO    client [MENU] Enter a command:
!START
2022-10-01 17:42:43 INFO    client [SERVER] File Transfer Command Started
2022-10-01 17:42:43 INFO    client [CONNECTED] Client #1 - Connected to 172.17.0.6:6969
2022-10-01 17:42:43 INFO    client [SERVER] OK
2022-10-01 17:42:43 INFO    client [CLIENT #1] Connection established waiting for file transfer
2022-10-01 17:42:48 INFO    client [CLIENT #1] Receiving file File 1 100mb.mp4 with size 114283931
Receiving File 1 100mb.mp4: 100%[██████████] 109M/109M [00:01<00:00, 88.3MB/s]
2022-10-01 17:42:49 INFO    client [CLIENT #1] File transfer time: 1.2931432723999023 seconds
2022-10-01 17:42:49 INFO    client [CLIENT #1] File transfer complete
2022-10-01 17:42:52 INFO    client [CLIENT #1] File transfer successful, integrity check passed - Hashes are equal.
2022-10-01 17:42:52 INFO    client File Size: 114283931 bytes
2022-10-01 17:42:52 INFO    client Server Hash: 9710e9df8be653f9d9422f7412d7768433824670
2022-10-01 17:42:52 INFO    client Client Hash: 9710e9df8be653f9d9422f7412d7768433824670
```

Por otro lado, el servidor informa por medio de logs acerca de todas las operaciones ejecutadas, entre ellas se encuentran: el inicio del servidor; información de puertos y dirección de listening; apertura de nuevas conexiones realizadas ya sea de un cliente principal o de conexiones secundarias; estado de conexiones activas; cambios en la configuración del servidor; desconexiones de clientes; información y estado final de la transferencia de archivos; y finalización del servidor.

```
→ docker run --platform linux/amd64 -i ghcr.io/communications-infrastructure/simple-file-transfer-system-server:main
2022-10-01 17:37:52 INFO    TCPServer [STARTING] server is starting...
2022-10-01 17:37:52 INFO    TCPServer [MAIN THREAD] Press Enter to exit...
2022-10-01 17:37:52 INFO    TCPServer [PROCESS] Server process ID: 8
2022-10-01 17:37:52 INFO    TCPServer [LISTENING] Server is listening on 172.17.0.6:6969
2022-10-01 17:38:04 INFO    TCPServer [NEW CONNECTION] ('172.17.0.5', 34852) connected.
2022-10-01 17:38:04 INFO    TCPServer [ACTIVE CONNECTIONS] 1
2022-10-01 17:38:04 INFO    TCPServer [MAIN CONNECTION] ('172.17.0.5', 34852) connected.
2022-10-01 17:39:17 ERROR   TCPServer An exception has occurred while running the server:
2022-10-01 17:39:17 ERROR   TCPServer Traceback (most recent call last):
2022-10-01 17:39:17 ERROR   TCPServer   File "/home/server/classes/client_handling.py", line 42, in handle_client
2022-10-01 17:39:17 ERROR   TCPServer     self.main_client()
2022-10-01 17:39:17 ERROR   TCPServer   File "/home/server/classes/client_handling.py", line 90, in main_client
2022-10-01 17:39:17 ERROR   TCPServer     self.num_clients = int(commands[2])
2022-10-01 17:39:17 ERROR   TCPServer IndexError: list index out of range
2022-10-01 17:41:25 INFO    TCPServer [FILE CONFIG] File to be sent: 100mb.mp4
2022-10-01 17:41:25 INFO    TCPServer [FILE CONFIG] File size: 114283931 bytes
2022-10-01 17:41:55 INFO    TCPServer [DISCONNECTED] ('172.17.0.5', 34852) disconnected.
2022-10-01 17:42:07 INFO    TCPServer [NEW CONNECTION] ('172.17.0.5', 47340) connected.
2022-10-01 17:42:07 INFO    TCPServer [ACTIVE CONNECTIONS] 1
2022-10-01 17:42:07 INFO    TCPServer [MAIN CONNECTION] ('172.17.0.5', 47340) connected.
2022-10-01 17:42:34 INFO    TCPServer [FILE CONFIG] File to be sent: 100mb.mp4
2022-10-01 17:42:34 INFO    TCPServer [FILE CONFIG] File size: 114283931 bytes
2022-10-01 17:42:43 INFO    TCPServer [FILE TRANSFER] - File Transfer Command Started
2022-10-01 17:42:43 INFO    TCPServer [NEW CONNECTION] ('172.17.0.5', 54690) connected.
2022-10-01 17:42:43 INFO    TCPServer [ACTIVE CONNECTIONS] 2
2022-10-01 17:42:49 INFO    TCPServer [FILE TRANSFER] - File 100mb.mp4 sent to ('172.17.0.5', 54690) [CLIENT 3] in 1.380197286605835 seconds
2022-10-01 17:42:49 INFO    TCPServer File successfully sent to Client 3
```

Para generar las imágenes de Docker, utilizamos un workflow con GitHub Actions que crea, actualiza, y publica paquetes automáticamente con cada push al repositorio, tanto para el contenedor del servidor como para el del cliente.

simple-file-transfer-system-server

Simple file transfer system using TCP protocol between client and server

Install from the command line: [Learn more](#)

```
$ docker pull ghcr.io/communications-infrastructure/simple-file-transfer-system-server:main
```

Recent tagged image versions

main	↓ 2
Published about 1 hour ago · Digest	...

[View and manage all versions](#)

README.md

Simple File Transfer System

Simple file transfer system using TCP protocol between client and server.

Details

communications-infrastructure
simple-file-transfer-system
MIT License
Readme

Last published 1 hour ago Issues 0

Total downloads 10

Collaborators 2

 [zejiran](#) Juan Alegría
 [ElReyZero](#)

[Open an issue](#)

De este modo, siempre se mantendrán consistentes los comandos de ejecución de cada uno de los componentes. Además, utilizando el comando ‘docker export’ podemos acceder al sistema de archivos de los contenedores, para analizar las operaciones exitosas y el comportamiento del sistema de transferencia, luego de todo el proceso de ejecución.

For executing the server in your system, use the following commands:

```
# Download Docker image
docker pull ghcr.io/communications-infrastructure/simple-file-transfer-system-server:main

# Start container with server
docker run --platform linux/amd64 -i ghcr.io/communications-infrastructure/simple-file-transfer-sys
```

For executing the client in your system, use the following commands:

```
# Download Docker image
docker pull ghcr.io/communications-infrastructure/simple-file-transfer-system-client:main

# Start container with client
docker run --platform linux/amd64 -i ghcr.io/communications-infrastructure/simple-file-transfer-sys
```

For verifying Docker image contents, export container filesystem with:

```
docker export $(docker ps -lq) -o out.tar
```

Enlace al repositorio de GitHub

- <https://github.com/communications-infrastructure/simple-file-transfer-system>

Enlace al video de explicación de desarrollo, aplicaciones, y resultados

- <https://youtu.be/zAG1iy0ky94>

Enlace de descarga a capturas de tráfico realizadas durante la práctica

- [Archivos Wireshark](#)

Análisis de las pruebas de desempeño obtenidas del servicio de transferencia

Prueba	1C/100MB	1C/250MB	5C/100MB	5C/250MB	10C/100MB	10C/250MB
Transferencia exitosa	Sí	Sí	Sí	Sí	Sí	Sí
Valor total de bytes recibidos por cada cliente	114283931	266789347	114283931	266789347	114283931	266789347
Tiempo de la transferencia para cada cliente	3.952 segundos	11.5935 segundos	Cliente1: 20.5304 s Cliente2: 27.2989 s Cliente3: 23.4189 s Cliente4: 26.1505 s Cliente5: 26.3159 s	Cliente1: 119.725 s Cliente2: 117.3087 s Cliente3: 114.9332 s Cliente4: 118.8057 s Cliente5: 105.2537 s	Cliente1: 39.789 s Cliente2: 50.701 s Cliente3: 57.235 s Cliente4: 41.542 s Cliente5: 52.643 s Cliente6: 56.804 s Cliente7: 22.659 s Cliente8: 60.067 s Cliente9: 51.455 s Cliente10: 51.425 s	Cliente1: 167.853 s Cliente2: 370.992 s Cliente3: 335.194 s Cliente4: 401.341 s Cliente5: 166.526 s Cliente6: 311.296 s Cliente7: 175.829 s Cliente8: 382.437 s Cliente9: 494.991 s Cliente10: 468.499 s
Tasa de transferencia a cada cliente	28.8 MB/s	23 MB/s	Cliente1: 5.57 MB/s Cliente2: 4.19 MB/s Cliente3: 4.88 MB/s Cliente4: 4.37 MB/s Cliente5: 4.34 MB/s	Cliente1: 2.23 MB/s Cliente2: 3.01 MB/s Cliente3: 2.32 MB/s Cliente4: 2.25 MB/s Cliente5: 2.53 MB/s	Cliente1: 2.87 MB/s Cliente2: 2.25 MB/s Cliente3: 1.99 MB/s Cliente4: 2.75 MB/s Cliente5: 2.17 MB/s Cliente6: 2.01 MB/s Cliente7: 5.044 MB/s Cliente8: 1.9026 MB/s Cliente9: 2.22 MB/s Cliente10: 2.22 MB/s	Cliente1: 1.58 MB/s Cliente2: 0.72 MB/s Cliente3: 0.80 MB/s Cliente4: 0.67 MB/s Cliente5: 1.60 MB/s Cliente6: 0.86 MB/s Cliente7: 1.52 MB/s Cliente8: 0.70 MB/s Cliente9: 0.54 MB/s Cliente10: 0.57 MB/s
Numero Puerto utilizado para la conexión de cada cliente (Aplicación Cliente)	62279	56889	57043, 57045, 57046, 57045, 57047,	57095, 57096, 57097, 57098, 57099,	51369, 51371, 51370, 51372, 51373, 51374, 51375, 51376, 51377, 51378	51456, 51457, 51458, 51459, 51460, 51461, 51462, 51463, 51465, 51466
Valor total de bytes transmitidos por el servidor a cada cliente	114283931	266789347	114283931	266789347	114283931	266789347
Tiempo de transferencia a cada uno de los clientes medido desde el servidor	3.9419 segundos	11.2437 segundos	Cliente1: 19.8701 s Cliente2: 27.237 s Cliente3: 23.1789 s Cliente4: 25.9013 s Cliente5: 26.0205 s	Cliente1: 119.9112 s Cliente2: 116.5220 s Cliente3: 113.1263 s Cliente4: 118.3554 s Cliente5: 102.9829 s	Cliente1: 38.9426 s Cliente2: 49.3071 s Cliente3: 57.0001 s Cliente4: 40.2048 s Cliente5: 52.3012 s Cliente6: 40.2048 s	Cliente1: 122.2840 s Cliente2: 370.5014 s Cliente3: 333.7360 s Cliente4: 399.3070 s Cliente5: 124.6701 s Cliente6: 307.9421 s

					Cliente7: 22.1892 s Cliente8: 59.9598 s Cliente9: 50.3936 s Cliente10: 50.1636 s	Cliente7: 97.1465 s Cliente8: 381.7627 s Cliente9: 494.8216 s Cliente10: 467.8046 s
Número Puerto utilizado para la conexión con cada cliente (Aplicación Servidor)	6969	6969	6969	6969	6969	6969

Prueba 1:

Se puede ver que en esta prueba se le está dando el ancho de banda completo al único usuario que está conectado, de esta manera, es capaz de aprovechar al máximo la red y la descarga del archivo.

Prueba 2:

Como en esta prueba sigue siendo una conexión, se le da el ancho de banda completo a la única conexión, la diferencia en la tasa de transferencia puede deberse a fluctuaciones en la red o congestión en ella debido al encolamiento de paquetes (ya que el tamaño recibido es mucho más grande que en la prueba 1).

Prueba 3:

Al comparar esta prueba en la anterior, se puede ver que las tasas de transferencia disminuyen bastante, esto se debe a que ahora hay 5 conexiones concurrentes recibiendo un archivo al mismo tiempo, con lo cual el ancho de banda de la red es compartido.

Prueba 4:

En este caso se notó una disminución de la tasa de transferencia aun mayor que en la prueba anterior a pesar de que el número de conexiones se haya mantenido igual, lo más posible es que se deba a un encolamiento de paquetes mucho más grande debido a que se está transmitiendo un poco más del doble de paquetes que en la prueba anterior.

Prueba 5:

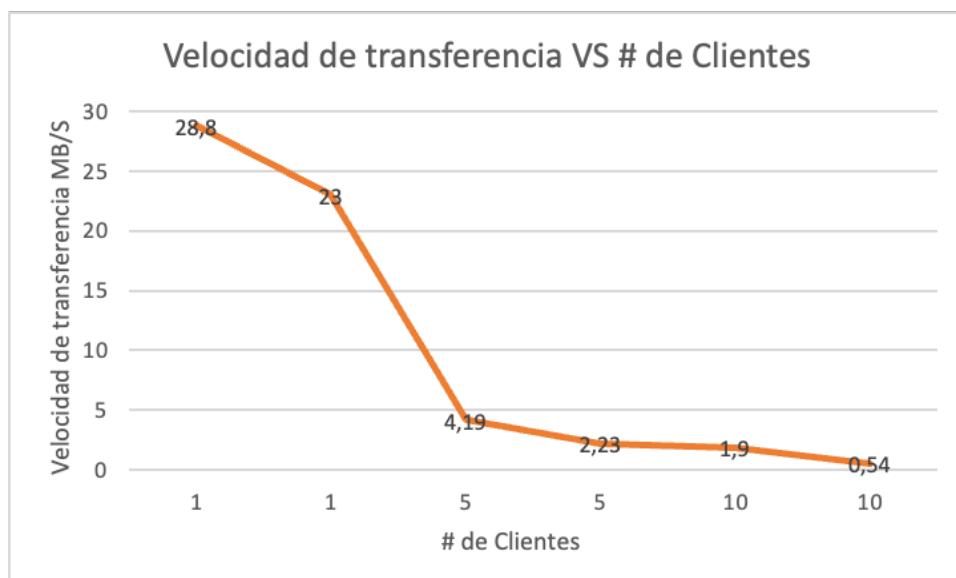
En la prueba 5, se puede evidenciar un cambio en la tasa de transferencia similar debido a la cantidad de conexiones recibiendo paquetes al mismo tiempo. Curiosamente, se demoró mucho menos que la prueba 4, esto puede ser causado por menor tráfico y congestión en la red al momento de realizar la prueba

Prueba 6:

En esta prueba se notó una disminución muchísimo mayor en la tasa de transferencia comparado con el resto de las pruebas. A pesar de que se haya mantenido la misma cantidad de clientes que en la prueba 5, al revisar el tráfico de la red se pudo evidenciar que existió un número elevado de reenvío de paquetes debido a posibles pérdidas de ellos. Igualmente, al tener que enviar más del doble de paquetes (aunque 20 veces más que en la prueba 1), tiene sentido que haya habido una disminución sustancial en la velocidad de transferencia.

Análisis general de las pruebas

En la anterior tabla se puede ver que a medida que aumenta la cantidad de las conexiones y el tamaño del archivo a enviar, existe una mayor cantidad de tráfico, encolamiento y congestión en la red, esto causa que las velocidades de transferencia simultáneas disminuyan sustancialmente y que el tiempo de transferencia de archivos sea mucho mayor que si se tuvieran conexiones secuenciales.



Gráfica 1. Comparación de velocidad de transferencia respecto a clientes activos.