UNIVERSIDAD DE BUENOS AIRES
FACULTAD DE INGENIERÍA

2do Cuatrimestre de 2025

[TA048] REDES - CURSO HAMELIN

Trabajo Práctico N°1

File Transfer

Integrantes:	*Padrón:*
Jorda, Santiago <sjorda@fi.uba.ar>	102924
Castellano Bogdan, Benjamín <bcastellano@fi.uba.ar>	111519
Embon, Joaquín Eduardo <jembon@fi.uba.ar>	111292
Sandroni, Morena <msandroni@fi.uba.ar>	110205
Taylor, Alan <ataylor@fi.uba.ar>	110034

2 de Octubre de 2025

Índice

1. Introducción

El objetivo de este trabajo es desarrollar una aplicación de red que permita la transferencia de archivos de forma segura, confiable y eficiente. La aplicación estará basada en una arquitectura cliente-servidor, en la cual los procesos se comunicarán utilizando UDP como protocolo de capa de transporte, seleccionando este protocolo por su simplicidad y flexibilidad, a pesar de no garantizar la entrega confiable de los datos.

Las operaciones principales a implementar son UPLOAD, que consiste en transferir un archivo desde el cliente hacia el servidor, y DOWNLOAD, que permite la transferencia de un archivo desde el servidor hacia el cliente. Para garantizar la integridad de los archivos y el correcto orden de los datos, se implementaron mecanismos de control de errores y retransmisión basados en los protocolos *Stop & Wait* y *Selective Repeat*, permitiendo gestionar acuses de recibo, retransmisiones y control de secuencias.

La aplicación será validada en una topología de red simulada con *Mininet*, en la cual se forzará la pérdida de paquetes para evaluar la robustez y eficiencia de la solución. Esto permitirá analizar cómo los protocolos implementados responden ante condiciones adversas de la red, asegurando la confiabilidad de la transferencia incluso en escenarios no ideales.

Además, se consideraron aspectos como la concurrencia en la atención de múltiples clientes, la partición adecuada de los archivos para evitar problemas de fragmentación y pérdida de paquetes, y la optimización de los tiempos de espera y retransmisión, buscando un equilibrio entre confiabilidad y eficiencia en la transferencia.

Con este trabajo se busca no solo desarrollar una aplicación funcional, sino también profundizar en los principios fundamentales de la transferencia de datos confiable, aplicando la teoría de redes a un caso práctico y evaluando su desempeño en un entorno controlado pero realista.

2. Hipótesis y Suposiciones Realizadas

Antes de comenzar la implementación, identificamos ciertos puntos que, a priori, sabíamos que serían clave para el éxito de nuestro trabajo y que determinarían la calidad de la aplicación, a saber:

- Manejo de múltiples clientes de manera concurrente

  Dado que UDP es un protocolo orientado a datagramas y no a conexiones persistentes como TCP, sabíamos que no contábamos con mecanismos intrínsecos para mantener una sesión dedicada con cada cliente. Esto requería un trabajo minucioso de identificación y gestión de los datos del lado del servidor, de forma que se pudiera garantizar la correcta entrega de los paquetes para cada cliente en particular. Asimismo, se asumió que los clientes podrían iniciar transferencias de manera simultánea, por lo que el servidor debía manejar la concurrencia de forma eficiente y segura.

- Implementación eficiente de los algoritmos de control de flujo

  La implementación de *Stop & Wait* y *Selective Repeat* planteaba el desafío de ofrecer la mayor eficiencia posible dentro de las limitaciones de cada protocolo. Para ello, era necesario seguir fielmente la teoría detrás de cada uno, optimizar el tiempo de los timeouts, gestionar correctamente los acuses de recibo (ACKs), y evitar bloqueos o retransmisiones innecesarias que pudieran afectar la velocidad global de transferencia.

- Partición y envío de información

  Sabiendo que, mediante la topología generada en *Mininet*, habría pérdida asegurada de paquetes, fue necesario discutir y definir cuidadosamente el tamaño de los paquetes a manejar. En archivos grandes, el riesgo de superar el MTU del enlace podría provocar fragmentación en la capa de red, generando paquetes fuera de nuestro control y aumentando la probabilidad de errores. Se asumió, además, que la aplicación debía manejar correctamente estas pérdidas y retransmisiones sin depender de la fragmentación automática de la red.

- Integridad y consistencia de los datos transferidos

  Se asumió que los archivos transferidos debían mantenerse íntegros, independientemente de la cantidad de retransmisiones necesarias. Esto implicó implementar mecanismos de verificación, como conteos de secuencia, que permitieran al receptor detectar y solicitar la retransmisión de paquetes perdidos o corruptos.

- Topología de red y condiciones adversas

  Se supuso que la topología simulada en *Mininet* reflejaría condiciones de red reales con pérdida de paquetes y retrasos variables. Esto permitió probar y validar que los protocolos implementados fueran robustos frente a condiciones no ideales, asegurando la confiabilidad de la transferencia de archivos.

3. Implementación

3.1. Paquetes

Como primer paso en el diseño de esta aplicación, fue necesario definir la forma que tendría la información intercambiada entre las entidades. Para ello, se estableció una estructura de paquete capaz de transportar tanto datos útiles como información de control, indispensable para garantizar el correcto flujo de transmisión.
Un paquete consta de dos partes:

1. **Header:** Contiene información de control necesaria para el funcionamiento del protocolo. En este caso, incluye los siguientes campos:
   - **Número de secuencia (SEQ):** Identifica de manera única a cada paquete.
   - **Número de acuse de recibo (ACK):** Indica el último paquete recibido de forma correcta.
   - **Tamaño del payload:** Indica el tamaño de la información que transporte el paquete.
   - **Flags** Campo que puede tener distintos valores dependiendo del tipo de paquete, ejemplos de este campo son el Acuse de Recibo (ACK), Finalización (FIN), Handshake o simplemente un paquete de datos.
   - **Checksum:** Campo que permite chequeo de errores sobre el paquete como perdida de información o corrupción de los datos.
2. **Payload:** Contiene la información útil que se desea transmitir.

3.2. Servidor

El servidor es responsable de aceptar las conexiones de los clientes y atender sus solicitudes de Upload y Download. Para ello, distribuye sus tareas en dos hilos principales:

3.2.1. Sender

Este hilo se encarga del envío de paquetes. Para ello mantiene una cola de paquetes pendientes de transmisión, la cual es alimentada por hilos dedicados a cada cliente. Dichos hilos son los responsables del procesamiento de los paquetes antes de colocarlos en la cola.

3.2.2. Receiver

Este hilo escucha en el socket del servidor y recibe los paquetes provenientes de los clientes. Una vez recibido un paquete, lo coloca en la cola correspondiente al cliente. En caso de tratarse de un nuevo cliente, reconoce el paquete de Handshake y lo incorpora a la estructura de clientes.
Adicionalmente, el servidor mantiene un hilo principal cuya función es monitorear la cola de clientes terminados y liberar los recursos asociados a ellos.

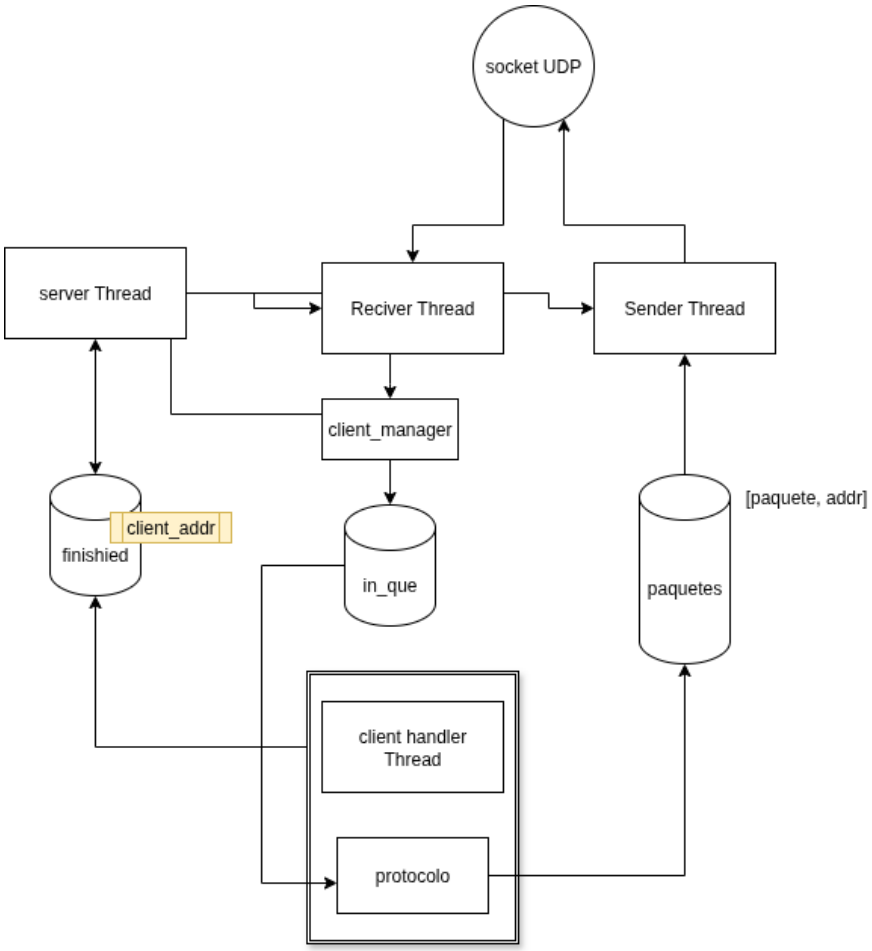
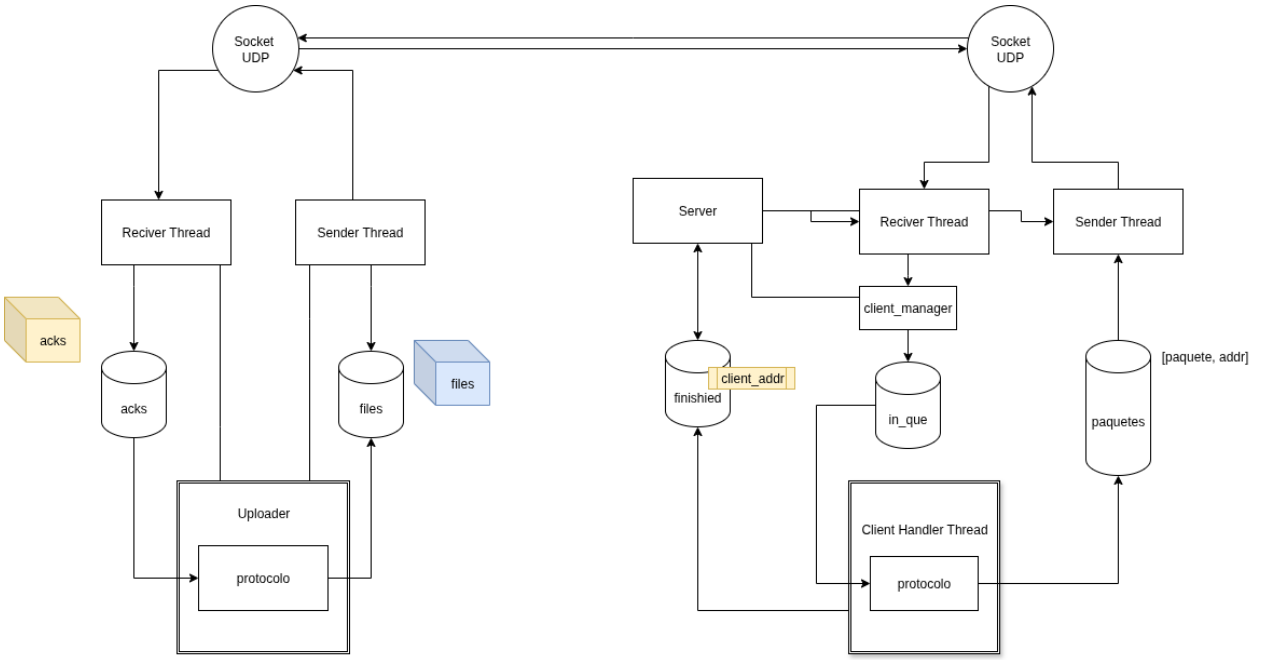


Figura 1: Diagrama Arquitectura del servidor, con sus hilos diferenciados

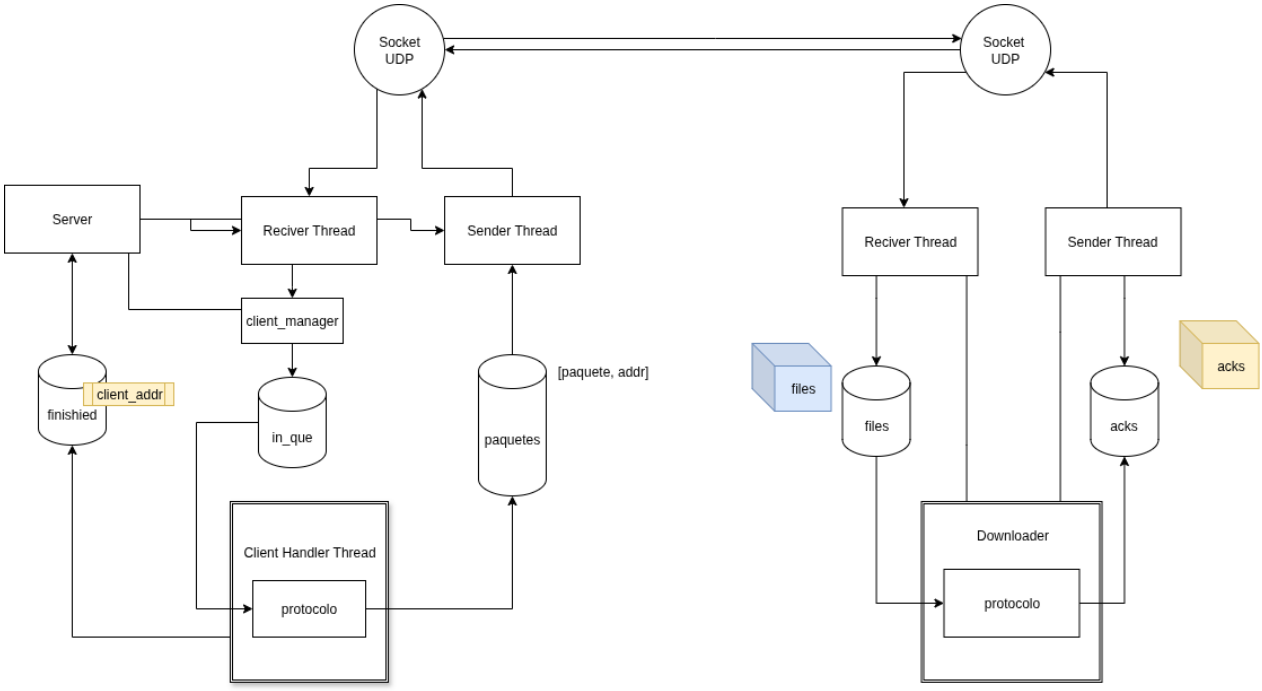
3.3. Upload y Download

Tanto la operación de Upload como la de Download comienzan con el establecimiento de una conexión con el servidor. El cliente inicia el proceso enviando un paquete con el bit de Handshake activado, al cual el servidor responde con un acuse de recibo.

La estructura del cliente es análoga a la del servidor, ya que cuenta con un hilo Sender encargado del envío de los paquetes en la cola de transmisión, y un hilo Receiver que escucha en el socket y deposita los paquetes recibidos en la cola principal.



(a) Arquitectura Upload.



(b) Arquitectura Download.

Figura 2: Diagrama de ambas operaciones.

3.4. Protocolos

Como uno de los atributos principales del cliente y el servidor en cada conexión, es el protocolo de transmisión que se va a aplicar durante la misma. El protocolo de cada entidad es el que se encarga de procesar los paquetes que están en la cola principal de ellos, previamente agregados por el hilo Receiver, y son los encargados de ir completando la cola del hilo Sender, para que envíe el paquete necesario, resultado

del procesamiento del paquete anterior.

Cada protocolo implementa una interfaz común que permite su utilización de manera transparente para el resto de la aplicación, permitiendo intercambiar responsables dependiendo de la operación a realizar. A continuación se describen los protocolos implementados:

3.4.1. Stop and Wait

Este protocolo es el más sencillo en cuanto a su implementación. Este consiste en chequear, por cada paquete enviado, que el mismo haya sido recibido correctamente por el receptor, mediante la recepción de un acuse de recibo (ACK). En caso de no recibir el ACK en un tiempo determinado (timeout), se retransmite el paquete. Este proceso se repite hasta que se recibe el ACK correspondiente.

```
    def send_file(
        self, filepath, filename, destination_address, in_queue, sender_queue
    ):
        seq_num = 0
        chunk_num = 0
        chunks = list(read_file_chunks(filepath, filename))

        for chunk_num, chunk in enumerate(chunks):
            packet = packageLib.create_data_packet(seq_num, chunk)
            retries = 0

            while retries < MAX_RETRIES:
                self._send(packet, destination_address, sender_queue)
                ack_received = self._wait_ack(seq_num, in_queue)

                if not ack_received:
                    retries += 1
                else:
                    break

            seq_num = 1 - seq_num

        fin_packet = packageLib.create_fin_packet()
        self._send(fin_packet, destination_address, sender_queue)
```

Es interesante contrastar que la implementación de este protocolo es muy sencilla, a cambio de una baja eficiencia en la utilización del canal de comunicación, ya que el emisor debe esperar el acuse de recibo por cada paquete enviado, lo que genera tiempos muertos en la transmisión.

3.4.2. Selective Repeat

A diferencia del protocolo Stop and Wait, el protocolo *Selective Repeat* permite el envío de múltiples paquetes sin esperar el ACK de cada uno de ellos. Esto se logra mediante la implementación de una ventana deslizante que controla los paquetes en tránsito, incrementando significativamente la eficiencia del canal de comunicación.

La implementación de este protocolo requirió un diseño más complejo para gestionar:

- **Ventana deslizante:** Mantiene un conjunto de paquetes en tránsito, permitiendo que el emisor envíe múltiples paquetes sin esperar confirmación.
- **Retransmisión selectiva:** Cada paquete tiene su propio temporizador, y solo se retransmiten aquellos que han expirado sin recibir confirmación.
- **Buffers de recepción:** El receptor almacena paquetes recibidos fuera de orden para entregar los datos en secuencia correcta.
- **Timeout adaptativo:** El tiempo de espera para la retransmisión se ajusta dinámicamente según las condiciones de la red.

```python
    def send_file(
        self, filepath, filename, destination_address, in_queue, sender_queue
    ):
        try:
            chunks = list(read_file_chunks(filepath, filename))
            total_chunks = len(chunks)

            while self.base < total_chunks:
                self._send_window_packets(chunks, destination_address, sender_queue)

                for _ in range(MAX_ACKS_TO_WAIT):
                    self._wait_for_acks(in_queue)
                    if self.base >= total_chunks:
                        break

                self._check_timeouts(chunks, destination_address, sender_queue)

            fin_packet = packageLib.create_fin_packet()
            self._send(fin_packet, destination_address, sender_queue)
```

Este fragmento muestra el método principal para el envío de un archivo utilizando Selective Repeat. El proceso se divide en tres etapas principales:

1. **Envío de paquetes:** Se envían paquetes hasta llenar la ventana deslizante.
2. **Espera de ACKs:** Se procesan los acuses de recibo, actualizando la ventana.
3. **Control de timeouts:** Se detectan y retransmiten los paquetes que hayan expirado sin recibir confirmación.

3.5. Parámetros de Ejecución

Uno de los puntos que a priori podrían no parecer tan importantes, pero que fueron de suma importancia para que la transferencia de archivos sea realmente eficiente, fue definir correctamente los parámetros de ejecución, de forma que optimicen tiempos, tamaños, y que la aplicación evite cuellos de botella innecesarios durante el envío y la recepción de datos. Entre los ajustes más importantes que se realizaron se encuentran:

3.5.1. Timeout

El cálculo del tiempo de espera antes de retransmitir un paquete (RTO, *Retransmission Timeout*) se implementó siguiendo el algoritmo adaptativo definido por TCP. La idea es ajustar dinámicamente el valor del timeout en función de las condiciones de la red, evitando tanto retransmisiones prematuras como demoras excesivas.

Para ello, cada vez que se envía un paquete se almacena el instante de envío. Cuando se recibe el `ACK` correspondiente, se mide el *SampleRTT*, es decir, el tiempo real que tardó ese paquete en ir y volver. Con esa muestra se actualizan dos estimadores mediante medias exponenciales:

$$\text{EstimatedRTT} = (1 - \alpha) \cdot \text{EstimatedRTT} + \alpha \cdot \text{SampleRTT}$$

$$\text{DevRTT} = (1 - \beta) \cdot \text{DevRTT} + \beta \cdot \left|\text{SampleRTT} - \text{EstimatedRTT}\right|$$

Finalmente, el RTO se fija como:

$$RTO = \text{máx}\left(0{,}05,\ \text{mín}(1{,}0,\ \text{EstimatedRTT} + 4 \cdot \text{DevRTT})\right)$$

De esta forma se asegura un valor mínimo de 50 ms y un máximo de 1 s, lo cual evita tanto retransmisiones innecesarias como tiempos de espera excesivos. En nuestra implementación utilizamos los parámetros estándar $\alpha = 0{,}125$ y $\beta = 0{,}25$.

Este mecanismo permitió que la aplicación se adaptara a condiciones variables de la red: en escenarios estables el DevRTT se mantiene bajo y el timeout se aproxima al RTT real, mientras que en escenarios inestables el RTO aumenta automáticamente para prevenir retransmisiones prematuras.

3.5.2. Tamaño de paquetes

Durante las primeras pruebas, utilizamos diferentes tamaños de paquete con el objetivo de encontrar un balance adecuado entre velocidad de transferencia y fiabilidad en la red. Inicialmente se probaron valores más pequeños, que si bien garantizaban la entrega correcta, generaban un overhead considerable debido a la gran cantidad de cabeceras transmitidas. Por el contrario, tamaños demasiado grandes resultaban en fragmentación de paquetes a nivel de red y un mayor costo en caso de retransmisión por pérdida.

Tras varios ajustes, se determinó que un payload de 1400 bytes ofrecía un rendimiento óptimo en nuestras condiciones de prueba. A este tamaño se le añadió un header de 9 bytes, necesario para incluir la información de control definida por nuestro protocolo. De esta manera, el tamaño total de cada paquete quedó en 1409 bytes, maximizando la utilización del canal sin incurrir en fragmentación ni sobrecarga excesiva.

4. Pruebas

4.1. Metodología de Pruebas

Para evaluar el desempeño de la aplicación desarrollada se definió una serie de pruebas controladas, en las que se midieron distintos aspectos del proceso de transferencia de archivos bajo condiciones normales y con pérdidas inducidas.

En primer lugar, se seleccionaron dos archivos de distinto tamaño con el objetivo de observar el comportamiento de ambos protocolos en escenarios de baja y alta carga de datos. El primero corresponde a un archivo pequeño (del orden de los kilobytes), mientras que el segundo es un archivo de 5 MB, lo cual permite diferenciar el rendimiento de la aplicación según el protocolo seleccionado para realizar la transferencia de información.

Adicionalmente, se realizaron pruebas en un entorno con un 10 % de pérdida de paquetes simulada, con el fin de evaluar la robustez del protocolo y comparar el impacto de dicha pérdida tanto en *Stop & Wait* como en *Selective Repeat*.

Durante cada ejecución se registraron:

- El tiempo total de transferencia del archivo.
- La correcta ejecución de las fases del protocolo: handshake inicial, envío de datos y finalización de la conexión.
- La cantidad de retransmisiones generadas como consecuencia de pérdidas.

De esta forma, fue posible analizar no solo la eficiencia temporal de cada estrategia de control de flujo, sino también la estabilidad y confiabilidad del proceso de transmisión en presencia de condiciones adversas.

4.2. Stop and Wait

Comenzamos con la ejecución de la aplicación aplicando este protocolo para un archivo pequeño.

```
joaquin@joaquin-HP-Laptop-15-dy2xxx:~/Desktop/Redes/TP1-Redes-Grupo5$ python3
src/upload.py -H 127.0.0.1 -p 9000 -s files -n file.txt -r stop-n-wait -v
Iniciando subida de archivo 'file.txt' al servidor 127.0.0.1:9000
[DEBUG] SENDER: - Hilo corriendo
[DEBUG] RECEIVER: - Hilo corriendo
[DEBUG] HANDSHAKE -

[DEBUG] HANDSHAKE - INICIO
[DEBUG] HANDSHAKE -
--- HANDSHAKE intento 1/3 ---
[DEBUG] HANDSHAKE - Se envia el bit de protocolo: b'\x00' del protocolo Stop &
 wait
[DEBUG] HANDSHAKE - Paquete de handshake enviado
[DEBUG] HANDSHAKE - Paquete de handshake recibido
[DEBUG] HANDSHAKE - EXITOSO

Datos del archivo a enviar
Nombre del archivo: file.txt
Ruta del a enviar: files/file.txt
size del archivo: 0.08 MB
Archivo dividido en 62 chunks de 1400 bytes cada uno

[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 0/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 1/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 2/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 3/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 4/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
```

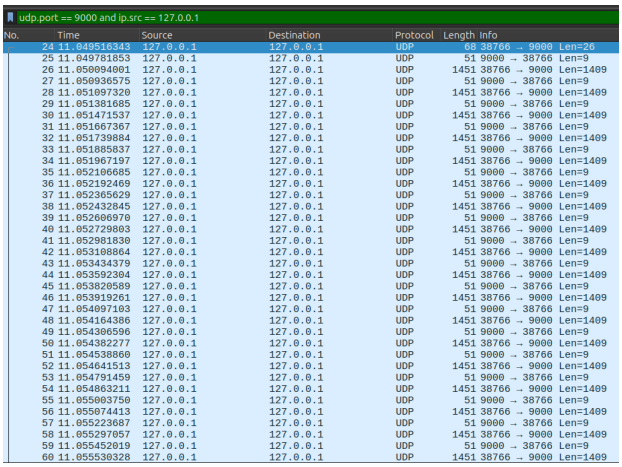
(a) Comienzo de la transferencia

```
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 52/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 53/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 54/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 55/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 56/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 57/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 58/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 59/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - Paquete enviado seq: 0 | chunk 60/62 | size: 1400 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 0
[DEBUG] STOPANDWAIT - ACK recibido seq: 0
[DEBUG] STOPANDWAIT - Paquete enviado seq: 1 | chunk 61/62 | size: 1311 bytes
[DEBUG] STOPANDWAIT - Esperando ACK para seq: 1
[DEBUG] STOPANDWAIT - ACK recibido seq: 1
[DEBUG] STOPANDWAIT - FIN enviado
Transferencia completada del archivo file.txt
Tiempo de transferencia: 0.01 segundos
[DEBUG] SENDER: - Hilo cerrado
[DEBUG] RECEIVER: - Hilo cerrado
joaquin@joaquin-HP-Laptop-15-dy2xxx:~/Desktop/Redes/TP1-Redes-Grupo5$
```

(b) Fin de la transferencia

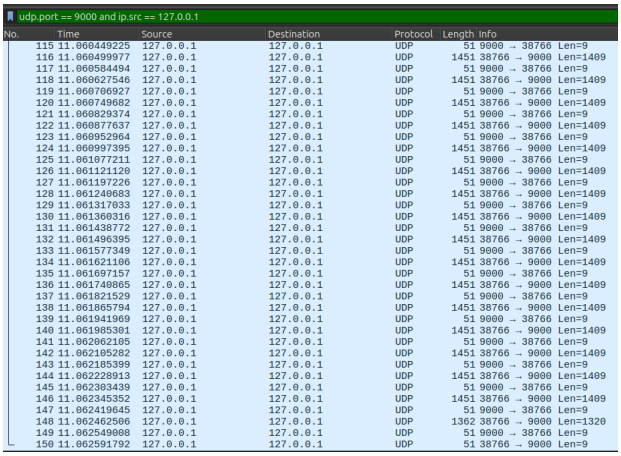
Figura 3: Transferencia de archivo pequeño con Stop and Wait.

Como podemos ver en los logs de la transferencia del archivo, parece ser que todo va en orden. Vamos a entrar más en detalle viendo la siguiente imagen.


udp.port == 9000 and ip.src == 127.0.0.1

No.	Time	Source	Destination	Protocol	Length	Info
24	11.049516343	127.0.0.1	127.0.0.1	UDP	68	38766 → 9000 Len=26
25	11.049781853	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
26	11.050094001	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
27	11.050936575	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
28	11.051097320	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
29	11.051381685	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
30	11.051471537	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
31	11.051667367	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
32	11.051739884	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
33	11.051885837	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
34	11.051967197	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
35	11.052106685	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
36	11.052192469	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
37	11.052365629	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
38	11.052432845	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
39	11.052606970	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
40	11.052729803	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
41	11.052981830	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
42	11.053108864	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
43	11.053434379	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
44	11.053592304	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
45	11.053820589	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
46	11.053919261	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
47	11.054097103	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
48	11.054164386	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
49	11.054306596	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
50	11.054382277	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
51	11.054538860	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
52	11.054641513	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
53	11.054791459	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
54	11.054863211	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
55	11.055003750	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
56	11.055074413	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
57	11.055223687	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
58	11.055297057	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
59	11.055452019	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
60	11.055530328	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409

(a) Comienzo de la transferencia


udp.port == 9000 and ip.src == 127.0.0.1

No.	Time	Source	Destination	Protocol	Length	Info
115	11.060449225	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
116	11.060499977	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
117	11.060584494	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
118	11.060627546	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
119	11.060706927	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
120	11.060749682	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
121	11.060829374	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
122	11.060877637	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
123	11.060952964	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
124	11.060997395	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
125	11.061077211	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
126	11.061121120	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
127	11.061197226	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
128	11.061240683	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
129	11.061317033	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
130	11.061360316	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
131	11.061438772	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
132	11.061496395	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
133	11.061577349	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
134	11.061621106	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
135	11.061697157	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
136	11.061740865	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
137	11.061821529	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
138	11.061865794	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
139	11.061941969	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
140	11.061985301	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
141	11.062062105	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
142	11.062105282	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
143	11.062185399	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
144	11.062228913	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
145	11.062303439	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
146	11.062345352	127.0.0.1	127.0.0.1	UDP	1451	38766 → 9000 Len=1409
147	11.062419645	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
148	11.062462506	127.0.0.1	127.0.0.1	UDP	1362	38766 → 9000 Len=1320
149	11.062549008	127.0.0.1	127.0.0.1	UDP	51	9000 → 38766 Len=9
150	11.062591792	127.0.0.1	127.0.0.1	UDP	51	38766 → 9000 Len=9

(b) Fin de la transferencia

Figura 4: Captura realizada con Wireshark de la transferencia completa

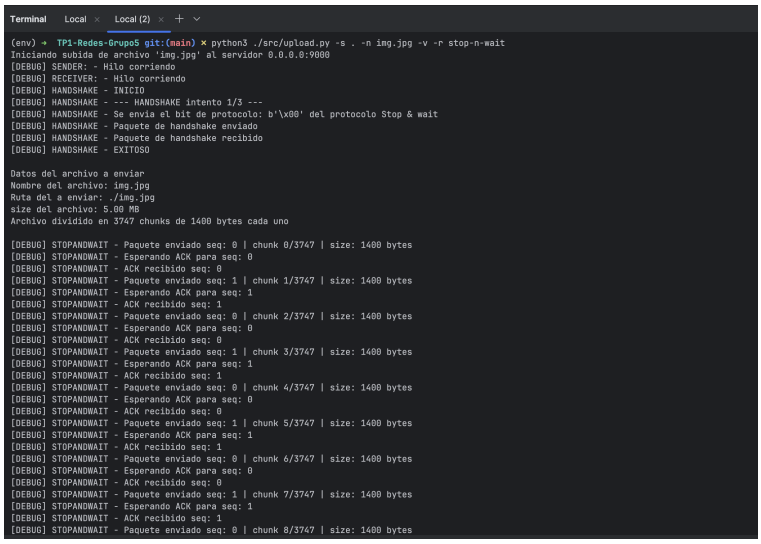
Con esta información a mano podemos identificar cada parte de la transferencia.

En las primeras dos líneas podemos ver como el cliente envía un paquete con un tamaño de 26 bytes, el cual corresponde al paquete de handshake, que tiene además del header, información adicional como el nombre del archivo, el tipo de operación a realizar, entre otros.

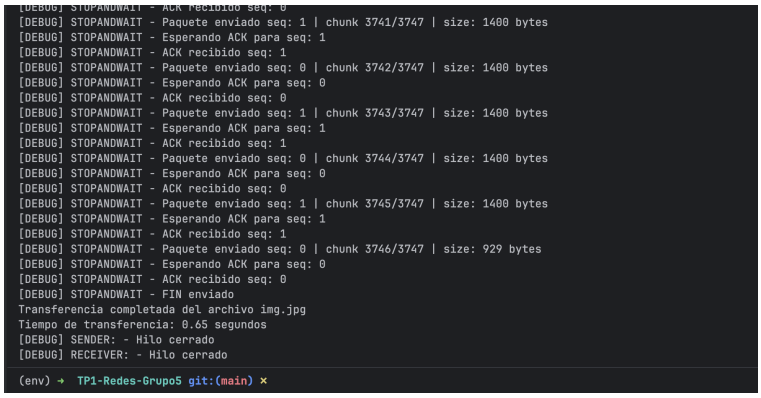
Luego, siguiendo con la teoría del protocolo *Stop & Wait*, comienza un ida y vuelta de paquetes, con su ACK correspondiente, podemos identificar los paquetes de información, los cuales son los que salen desde el cliente (la operación es upload), con un tamaño máximo de 1409 bytes (header + payload máximo). Los paquetes que vienen del servidor, son todos ACKS con un tamaño de 9 bytes, correspondientes solo al tamaño del header, que es el que trae esa información.

Al final, el cliente envía un paquete de 9 Bytes, pero que en este caso corresponde al flag de FIN, haciéndole saber al servidor que la transferencia se ha completado.

Continuaremos el análisis poniendo a prueba el protocolo para la transferencia de un archivo con un tamaño más considerable, de alrededor de 5 MB.



(a) Comienzo de la transferencia.



(b) Fin de la transferencia.

Figura 5: Transferencia de archivo grande con Stop and Wait.

Viendo la ejecución de la transferencia, podemos ver que la misma se llevó a cabo de manera satisfactoria. Se puede ver que duró 7 veces más que el primer ejemplo que vimos, pero de todas formas el tiempo tardado es menor al segundo sin perdidas de paquetes. Esto nos ayuda a concluir que la arquitectura diseñada y el intercambio de información se esta dando de manera muy eficiente, ofreciendo una buena velocidad incluso cuando los archivos comienzan a crecer.

Por último vamos a observar una captura correspondiente a la transferencia del mismo archivo, pero esta vez forzando una perdida de paquetes del 10 % en el enlace de red.

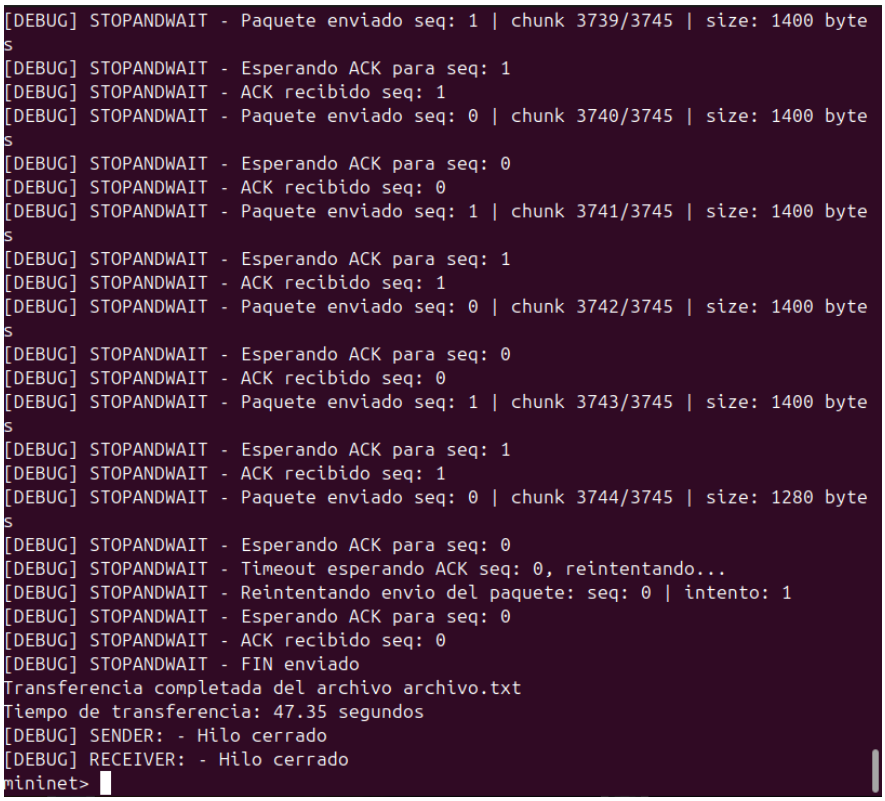
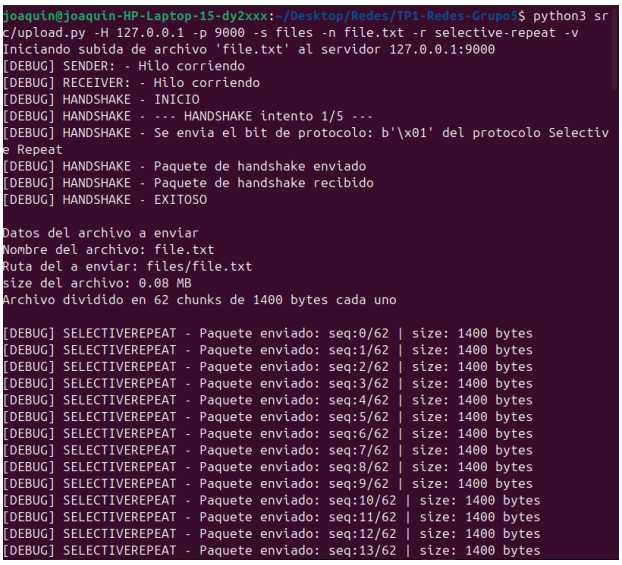


Figura 6: Transferencia de un archivo de 5MB con pérdida de 10 % para Stop and Wait

Acá se hace visible el cuello de botella de Stop and Wait, con las condiciones de pérdida que pusimos sobre el enlace, el protocolo tardo casi 50 segundos en transferir un archivo del mismo tamaño anterior, se puede ver a lo último un ejemplo de cómo se comporta el protocolo cuando se pierde un paquete. Si la entrega del ACK es mayor al timeout para ese paquete, se da como perdido y se retransmite.

4.3. Selective Repeat

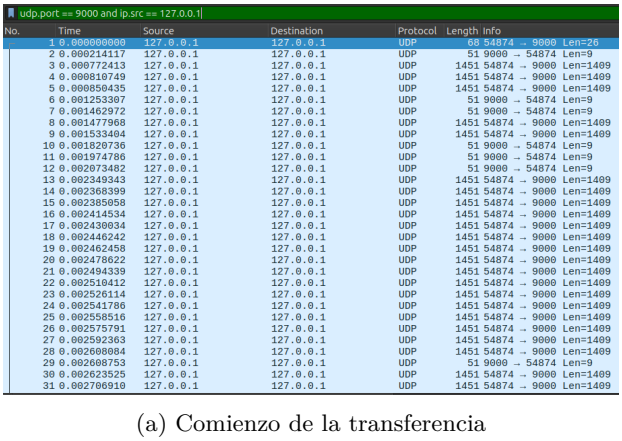
Comenzamos con la ejecución de la aplicación aplicando este protocolo para un archivo pequeño.



(a) Comienzo de la transferencia (b) Fin de la transferencia

Figura 7: Transferencia de archivo pequeño con *Selective Repeat*.

Al observar los logs de la transferencia, se aprecia que el protocolo funciona correctamente. Para profundizar, pasamos a analizar la captura de red.



(a) Comienzo de la transferencia (b) Fin de la transferencia

Figura 8: Captura realizada con Wireshark de la transferencia completa

En este caso, el paquete inicial de 26 bytes corresponde al handshake, con la misma información adicional (nombre del archivo, tipo de operación, etc.). Sin embargo, a diferencia de *Stop & Wait*, se observa claramente cómo el cliente comienza a enviar varios paquetes de datos consecutivos sin esperar el ACK inmediato de cada uno, lo que corresponde al ventaneo del protocolo *Selective Repeat*.

Los ACKs del servidor también aparecen en la captura, pero se encuentran intercalados, y en algunos casos llegan después de varios paquetes de datos. Esto es esperable, ya que el receptor confirma bloques de paquetes y no necesariamente uno por uno en orden estricto. El tamaño máximo de los paquetes de datos sigue siendo de 1409 bytes, mientras que los ACKs son de 9 bytes, como en el protocolo anterior.

Finalmente, el cliente envía el paquete con el flag de FIN (9 bytes), indicando el cierre de la transferencia.

A continuación, probamos con un archivo de mayor tamaño, alrededor de 5 MB.

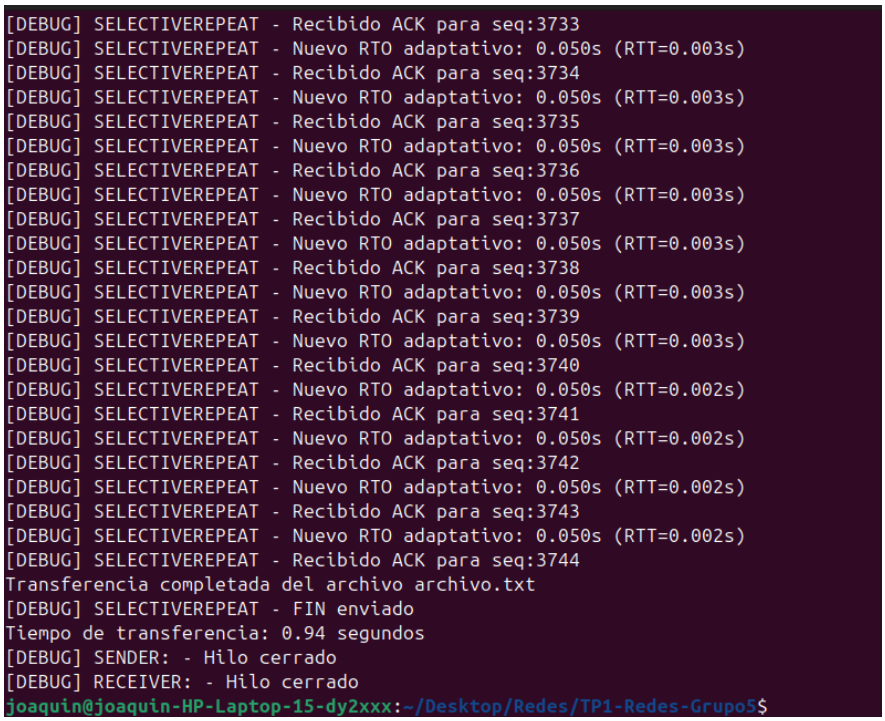


Figura 9: Transferencia de un archivo de 5MB *Selective Repeat*

En este escenario se aprecia que la transferencia se completa en un tiempo de un orden similar al de *Stop & Wait*, pero manteniendo un flujo constante de paquetes. En la captura puede verse claramente cómo el receptor envía múltiples ACKs de forma consecutiva, correspondientes a la ventana de datos recibida. Esto refleja la lógica del protocolo, que permite confirmar bloques de paquetes en lugar de responder inmediatamente a cada uno.

Además, se observa el cálculo dinámico del Retransmission Timeout (RTO), que es fundamental en *Selective Repeat* para ajustar los tiempos de espera en función de las condiciones de la red. Este mecanismo introduce una pequeña sobrecarga en entornos sin pérdidas, lo cual explica que la transferencia de 5 MB haya resultado ligeramente más lenta que con *Stop & Wait*.

Por último, realizamos la misma transferencia con un 10 % de pérdida de paquetes en el enlace de red.

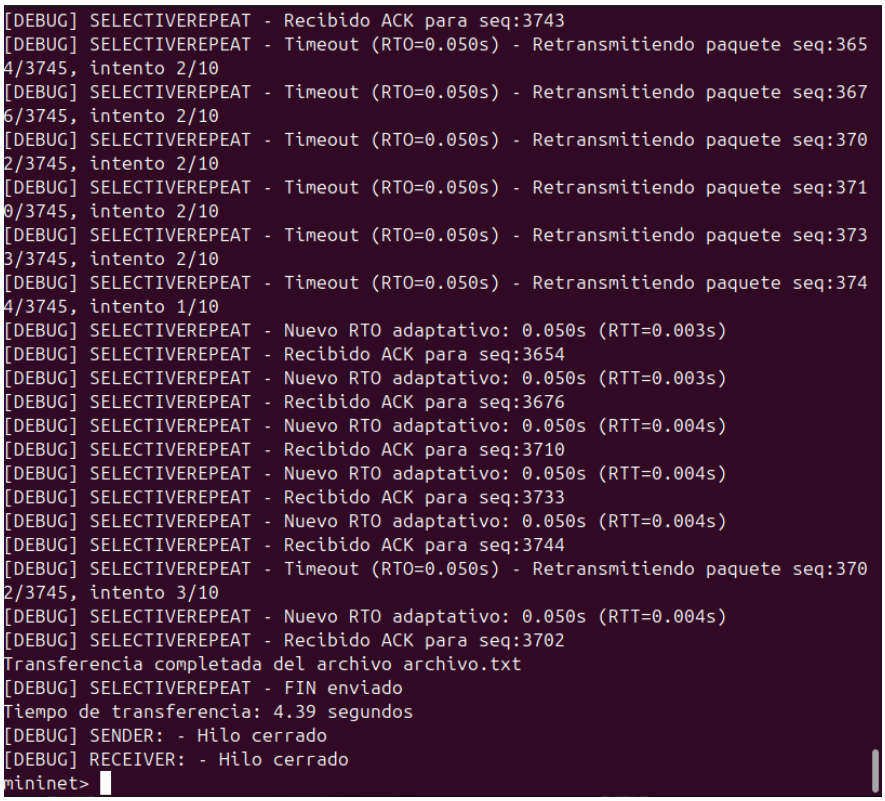


Figura 10: Transferencia de un archivo de 5MB con pérdida de 10 % para *Selective Repeat*

En estas condiciones adversas, se observa que el tiempo de transferencia se ve afectado, pero mucho menos que en *Stop & Wait*. La retransmisión selectiva permite que, cuando un paquete se pierde, solo sea necesario volver a enviar ese paquete en particular y no bloquear el flujo completo de la ventana.

Gracias a esto, el protocolo logra completar la transferencia en un tiempo mucho más veloz, menor a 5 segundos, demostrando su mayor robustez frente a pérdidas de paquetes y su capacidad de mantener un buen desempeño incluso en enlaces poco confiables.

4.4. Comparativa y Discusión

A partir de las pruebas realizadas, se puede establecer una comparación directa entre ambos protocolos en distintos escenarios. En la Tabla 1 se resumen los tiempos de transferencia para un archivo de 5 MB, tanto en condiciones ideales como bajo un 10 % de pérdidas de paquetes.

Protocolo	**Sin pérdidas (s)**	**Con 10 % de pérdidas (s)**
Stop and Wait	0.65	47.35
Selective Repeat	0.90	4.39

Cuadro 1: Comparación de tiempos de transferencia para archivo de 5 MB

En condiciones ideales (0 % de pérdidas), el protocolo *Stop and Wait* logra un tiempo de transferencia ligeramente menor que *Selective Repeat*. Esto se explica porque, en nuestro caso, *Selective Repeat* incluye un cálculo dinámico del *Retransmission Timeout (RTO)* y la gestión de la ventana, lo cual introduce una sobrecarga adicional que no se ve compensada cuando no existen pérdidas en la red.

No obstante, al incorporar un 10 % de pérdidas de paquetes en el enlace, las diferencias se vuelven notorias. Mientras que *Stop and Wait* necesita casi 50 segundos para completar la transferencia, debido a que cada pérdida implica esperar al vencimiento del timeout para retransmitir, *Selective Repeat* logra finalizar

en apenas 4 segundos. Esto se debe a que el protocolo retransmite únicamente los paquetes efectivamente perdidos y permite que el resto de la ventana siga avanzando, evitando bloqueos innecesarios.

En conclusión, aunque *Stop and Wait* puede resultar competitivo en escenarios perfectos sin pérdidas, el protocolo *Selective Repeat* demuestra una clara superioridad en entornos realistas y adversos, ya que su capacidad de ventaneo y retransmisión selectiva asegura una mayor eficiencia, estabilidad y robustez frente a la pérdida de paquetes.

5. Preguntas a Responder

A continuación, luego del desarrollar el detalle técnico de nuestra implementación y el mejor entendimiento que logramos alcanzar, vamos a responder la preguntas planteadas en el enunciado del trabajo práctico.

5.1. Describir el Protocolo Cliente - Servidor

El protocolo Cliente - Servidor es una arquitectura de aplicación que consiste en el intercambio de mensajes entre dos entidades: el cliente, que inicia la comunicación y solicita servicios, y el servidor, que espera las solicitudes y proporciona los servicios requeridos. En nuestro caso, el cliente, en caso de la operación de UPLOAD, envía un archivo al servidor, el cual se ocupa de enviar acuses de recibo de cada paquete recibido, y en caso de la operación de DOWNLOAD, el cliente solicita un archivo al servidor, el cual se ocupa de enviarlo en paquetes, esperando los acuses de recibo correspondientes.

5.2. ¿Cuál es la función de un protocolo de capa de aplicación?

La función de un protocolo de capa de aplicación es definir cómo las aplicaciones en distintos hosts se comunican entre sí a través de la red. Entre sus funciones principales, podemos destacar:

- Especificar el formato y la estructura de los mensajes que se envían entre aplicaciones.
- Definir la secuencia de interacción, por ejemplo, quién habla primero, cómo se confirma la recepción, qué pasa si hay un error.
- Dar semántica a los datos transmitidos, es decir, que ambas aplicaciones entiendan lo mismo al intercambiar información.
- Ofrecer servicios específicos para el usuario final o la aplicación (transferencia de archivos, correo electrónico, navegación web, chat, etc.).

5.3. Detalle el protocolo de aplicación desarrollado en este trabajo

El protocolo de aplicación desarrollado en este trabajo define la comunicación entre cliente y servidor para la transferencia de archivos. Se diseñó sobre la capa de transporte, proporcionando reglas específicas de interacción, formato de mensajes y manejo de confiabilidad.

Entidades participantes

- **Cliente:** inicia la transferencia de archivos, enviando y recibiendo datos.
- **Servidor:** recibe los pedidos de los clientes, administra la concurrencia y distribuye los datos a través de hilos dedicados.

Tipos de mensajes

- **HANDSHAKE:** utilizado al inicio de la comunicación para establecer la conexión.
- **DATOS:** contienen bloques de información del archivo a transferir.
- **ACK:** confirman la recepción correcta de un bloque de datos.
- **FIN:** utilizado para finalizar la transferencia y cerrar la comunicación.

Flujo de comunicación

La comunicación se organiza en secuencias alternadas de tipo:

- `HANDSHAKE` → `ACK` para establecer la conexión.
- `DATOS` → `ACK` durante la transferencia.
- `FIN` → `ACK` para cerrar la conexión de forma ordenada.

El receptor siempre utiliza ACK para confirmar la recepción correcta de cada mensaje, asegurando la fiabilidad de la transferencia.

Características principales

- **Fiabilidad:** mediante el uso de ACK explícitos.
- **Concurrencia:** múltiples clientes pueden transferir archivos simultáneamente sin interferencias.
- **Simplicidad:** formato de mensajes reducido (HANDSHAKE, DATOS, ACK y FIN).

5.4. TCP vs UDP ¿Qué servicios proveen? ¿Cuáles son sus características? ¿Cuándo usar uno u otro?

Servicios Proveídos

Mientras que UDP es un protocolo de capa de transporte que ofrece un servicio orientado a datos, con servicios básicos de multiplexación, demultiplexación y detección de errores, TCP es un protocolo de capa de transporte orientado a conexiones, que ofrece servicios más avanzados, como control de flujo, control de congestión y manejo de retransmisiones ante pérdidas de paquetes.

Características

UDP se caracteriza por ser un protocolo que ofrece rapidez y baja latencia, a cambio de un intercambio de información menos confiable. No garantiza la entrega de paquetes, ni el orden en que llegan, ni la integridad de los datos. Por otro lado, TCP es un protocolo más robusto y confiable, que asegura la entrega de datos en el orden correcto y sin pérdidas, pero a costa de operaciones adicionales, como el establecimiento de conexiones, que pueden introducir latencia.

Cuándo Usar Uno u Otro

La elección entre TCP y UDP depende de los requisitos específicos de la aplicación. Si la aplicación requiere una transferencia de datos segura y ordenada, como en la transferencia de archivos o la navegación web, TCP es la opción preferida. En cambio, si la aplicación prioriza la velocidad y puede tolerar cierta pérdida de datos, como en transmisiones de video en tiempo real o juegos en línea, UDP es más adecuado.

6. Dificultades

A lo largo del desarrollo de este proyecto, nos hemos enfrentado a varias dificultades técnicas y conceptuales. La resolución de de estos desafíos requirió una combinación de profunda investigación, distintas implementaciones que de forma iterativa nos llevaron a la solución final, y la colaboración efectiva entre los miembros del equipo.

6.1. Flujo de los Datos

Probablemente el mayor desafío que encontramos, y la parte del trabajo que más cambió fue definir un flujo de datos eficiente y robusto. Inicialmente, el cliente y el servidor se diferenciaban en la forma en la que manejaban los datos.

En el caso del cliente, la comunicación se organizaba en secuencias `ACK` $\rightarrow$ `Datos` o `Datos` $\rightarrow$ `ACK`, manteniendo siempre una correspondencia clara entre confirmación y transmisión.

Por su parte, el servidor contaba con un hilo principal encargado de recibir los paquetes. Dependiendo del par `(dirección, puerto)` de destino, los derivaba a un hilo dedicado a cada cliente. Estos hilos mantenían colas de mensajes con los paquetes que debían enviarse. Posteriormente, el hilo principal era el responsable de tomar esos mensajes y enviarlos efectivamente a través del socket.

Con el objetivo de mejorar la eficiencia y la claridad del flujo de datos, optamos finalmente por utilizar la implementación que describimos en las secciones anteriores. Este enfoque nos permitió una eficiencia mucho mayor, y una estructura de código más clara y mantenible y reutilizable.

6.2. Concurrencia y Sincronización

Siguiendo con la estructura concurrente descrita en la sección anterior, la sincronización de los hilos y la gestión de la cola de mensajes como recurso de acceso compartido, requirió de un minucioso diseño e implementación. Mediante una gran tarea de debugging y pruebas, logramos definir y separar las tareas críticas para los hilos principales durante la ejecución de cada operación, lo cual alcanzó y nos permitió evitar condiciones de carrera y garantizar el acceso seguro a la cola de mensajes.

6.3. Optimización de Parámetros

Los parámetros que existen en toda la aplicación, como el tamaño de la ventana en el protocolo de *Selective Repeat*, el tamaño máximo de los paquetes, los tiempos de espera para la retransmisión de paquetes, entre otros, fueron optimizados a través de un proceso iterativo de pruebas y ajustes. Este proceso implicó la realización de múltiples experimentos para evaluar el rendimiento bajo diferentes configuraciones, lo que nos permitió conocer mejor el comportamiento y estructura de la red, y así ajustar los parámetros para lograr un equilibrio óptimo entre eficiencia y confiabilidad.

Es interesante remarcar que además del trabajo que conllevó terminar de definir este aspecto de la implementación, por momentos también fue difícil terminar de entender el impacto que provocaba una mala elección de estos valores, creyendo que la calidad de la transferencia de archivos estaba mayormente determinada por los algoritmos en si.

7. Conclusiones

Durante el desarrollo de este trabajo, hemos estudiado los principios básicos de la transferencia de datos confiable a través de redes, hemos implementado una aplicación de red, con arquitectura cliente - servidor, que soporta dos operaciones principales, UPLOAD para subir archivos aplicación servidor, y DOWNLOAD para descargar archivos desde el mismo. Como objetivo principal, se implementó un protocolo de aplicación que utiliza el protocolo UDP como protocolo de capa de transporte, y que mediante la utilización de protocolos que manejan la pérdida de paquetes, *Stop & Wait* y *Selective Repeat*, logra una transferencia de datos confiable.

La performance del protocolo implementado fue evaluada mediante el despliegue de una topología de red virtual, utilizando la herramienta *Mininet*, que permitió simular condiciones de red adversas, y medir el rendimiento del protocolo.

Finalmente, habiendo adquirido los conocimientos necesarios, respondimos a las preguntas teóricas planteadas en la consigna, y comparamos las dificultades reales a las que nos enfrentamos durante la implementación del protocolo, con las suposiciones iniciales de los desafíos que nos podíamos encontrar durante el desarrollo del proyecto.

Este trabajo no solo nos permitió afianzar nuestros conocimientos teóricos y entender los desafíos prácticos de la transferencia de datos en redes, sino que también es interesante haber podido experimentar el valor de la colaboración y el diseño iterativo de la solución final y eficiente.