

REDES: INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS
(TA048) CURSO: ALVAREZ HAMELIN

Trabajo Práctico 1 Reliable Data Transfer



Mayo 2025

Agustín Altamirano	110237
Cristhian David Noriega	109164
Ian von der Heyde	107638
Juan Martín de la Cruz	109588
Santiago Tomás Fassio	109463

Índice

1. Introducción	4
2. Hipótesis y suposiciones realizadas	4
3. Implementación	5
3.1. Arquitectura general	5
3.2. Stop and Wait	5
3.3. Selective Repeat	5
3.4. Módulos y clases clave	5
3.5. Flujo de control para UPLOAD	6
3.5.1. Casos particulares con StopAndWait	7
3.5.2. Casos particulares con SelectiveRepeat	8
3.6. Flujo de control para DOWNLOAD	9
3.6.1. Casos particulares con StopAndWait	11
3.6.2. Casos particulares con SelectiveRepeat	12
3.7. Decisiones sobre las constantes de configuración	12
4. Pruebas	13
4.1. Metodología de Evaluación	13
4.2. Resultados sin Pérdida de Paquetes	14
4.2.1. Subir un archivo de 5MB usando el protocolo Stop & Wait	14
4.2.2. Descargar un archivo de 5MB usando el protocolo Stop & Wait	14
4.2.3. Subir un archivo de 5MB usando el protocolo Selective Repeat	15
4.2.4. Descargar un archivo de 5MB usando el protocolo Selective Repeat	15
4.3. Resultados con Pérdida de Paquetes	15
4.3.1. Subir un archivo de 5MB usando el protocolo Stop & Wait	15
4.3.2. Descargar un archivo de 5MB usando el protocolo Stop & Wait	15
4.3.3. Subir un archivo de 5MB usando el protocolo Selective Repeat	16
4.3.4. Descargar un archivo de 5MB usando el protocolo Selective Repeat	16
4.3.5. Subir un archivo de 10MB usando el protocolo Stop & Wait	16
4.3.6. Descargar un archivo de 10MB usando el protocolo Stop & Wait	17
4.3.7. Subir un archivo de 10MB usando el protocolo Selective Repeat	17
4.3.8. Descargar un archivo de 10MB usando el protocolo Selective Repeat	17
5. Análisis	17
5.1. Tabla sobre los resultados sin Pérdida de Paquetes	17
5.2. Tabla sobre los resultados con Pérdida de Paquetes	18
5.3. Interpretación de los resultados obtenidos	18
6. Preguntas	19
6.1. Arquitectura Cliente-Servidor	19

6.2. ¿Cuál es la función de un protocolo de capa de aplicación?	20
6.3. Detalle el protocolo de aplicación desarrollado en este trabajo.	20
6.4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuándo es apropiado utilizar cada uno?	22
6.4.1. TCP (Transmission Control Protocol)	22
6.4.2. UDP (User Datagram Protocol)	23
7. Dificultades encontradas	23
8. Conclusion	24
9. Anexo: Fragmentación IPv4	26
9.1. Topología de red virtual	26
9.2. Generación y análisis de tráfico UDP por la red	26
9.3. Generación y análisis de tráfico TCP por la red	28

1. Introducción

En el ámbito de las redes de computadoras, garantizar la entrega confiable de datos a través de canales inherentemente no confiables constituye un desafío central. Para enfrentar esta problemática, se han diseñado diversos protocolos de control de errores, cuya finalidad es asegurar la transmisión íntegra y ordenada de la información entre sistemas distribuidos, incluso en presencia de pérdidas, duplicaciones o retrasos.

Este trabajo práctico aborda dicha problemática mediante la implementación de una arquitectura cliente-servidor desarrollada en el lenguaje de programación **Python**, utilizando la biblioteca estándar **socket** para establecer comunicación entre procesos sobre el protocolo UDP en la capa de transporte. Al tratarse de un protocolo no orientado a la conexión y sin mecanismos de confiabilidad incorporados, se implementaron dos soluciones para lograr una transferencia de archivos robusta: el protocolo *Stop-and-Wait*, como enfoque simple y secuencial, y *Selective Repeat*, como mecanismo más avanzado y eficiente que permite la retransmisión selectiva de paquetes perdidos o corruptos.

La implementación se diseñó para operar sobre una red no confiable simulada en **mininet**, permitiendo la inyección artificial de condiciones adversas como pérdidas, duplicaciones y retrasos en los paquetes, con el fin de evaluar el comportamiento y la eficacia de cada protocolo en contextos realistas. El sistema desarrollado cuenta con una estructura modular que separa claramente las responsabilidades de envío y recepción, favoreciendo la legibilidad, el mantenimiento y la extensibilidad del código.

A lo largo del informe se describen las decisiones de diseño adoptadas, los desafíos técnicos enfrentados durante el desarrollo, las estrategias implementadas para garantizar la confiabilidad de la comunicación y los resultados obtenidos en distintas pruebas de funcionamiento y estrés.

2. Hipótesis y suposiciones realizadas

Para guiar el desarrollo y la evaluación de los protocolos *Stop-and-Wait* y *Selective Repeat*, se plantean las siguientes hipótesis y suposiciones:

1. Hipótesis de confiabilidad:

- Bajo una tasa de pérdida de paquetes, ambos protocolos lograrán la transferencia completa de archivos sin errores de integridad.

2. Hipótesis de rendimiento:

- En condiciones de alta latencia o retrasos variables, *Selective Repeat* mostrará mayor eficiencia (throughput efectivo) que *Stop-and-Wait*, gracias al uso de ventanas deslizantes que permiten mantener el canal ocupado.

3. Hipótesis de escalabilidad:

- El servidor podrá atender de forma concurrente a múltiples clientes simultáneos sin degradación significativa del rendimiento individual de cada transferencia.
- La modularidad de la arquitectura cliente-servidor facilitará la extensión futura a otros mecanismos de recuperación de errores (por ejemplo, *Go-Back-N*) sin necesidad de refactorizaciones profundas.

4. Suposiciones de red y modelo de error:

- Las pérdidas, duplicaciones y retrasos de paquetes se distribuyen de manera aleatoria e independiente, sin patrones de ráfagas.
- En caso de que los paquetes lleguen desordenados, se garantizará su reordenamiento.

- No se considera la corrupción de datos en tránsito (errores de bit), asumiendo que un paquete corrupto se comporta como paquete perdido o duplicado.
- El modelo de retransmisión y temporización (timeouts) se basa en estimaciones estáticas del RTT, sin algoritmos de adaptación dinámicos.

3. Implementación

3.1. Arquitectura general

La solución se organiza en dos componentes principales:

- **Cliente:** instancia de la clase `Client`, que expone dos métodos para las operaciones `UPLOAD` y `DOWNLOAD`, respectivamente. Según el parámetro `protocol.type`, el cliente delega toda la lógica de transmisión fiable al objeto `StopAndWait` o `SelectiveRepeat`.
- **Servidor:** instancia de la clase `Server`, que arranca dos hilos:
 1. `ServerReceiver.run()`: recibe todos los paquetes UDP entrantes, los desempaquetta en objetos `TransportProtocolSegment` y los despacha hacia un manejador `UserManager` dedicado a cada cliente.
 2. `ServerSender.run()`: extrae de una cola (`send_queue`) los segmentos que deben enviarse (ACKs, datos, FIN, etc.) y los envía al socket UDP.

Cada nuevo cliente genera un hilo `UserManager`, que coordina el protocolo concreto (Stop-and-Wait o Selective Repeat) en su rol de servidor.

3.2. Stop and Wait

Implementa una versión simple de un protocolo de transmisión confiable. El emisor envía un segmento y espera su confirmación (ACK) antes de continuar con el siguiente. Si el ACK no llega en un tiempo determinado (timeout), retransmite el segmento. Este mecanismo garantiza orden y fiabilidad, pero puede desaprovechar el canal si la latencia es alta, ya que solo se permite un paquete 'en vuelo' a la vez.

3.3. Selective Repeat

Implementa un protocolo más avanzado que mejora el rendimiento usando una ventana deslizante. Permite que múltiples segmentos estén 'en vuelo' simultáneamente, sin esperar un ACK por cada uno antes de enviar el siguiente. Cada segmento se numera, y el receptor puede confirmar (ACK) paquetes fuera de orden. El emisor solo retransmite aquellos que no fueron reconocidos tras un timeout, no todos. Esto permite mayor eficiencia en enlaces con alta latencia o pérdida.

$$(1) \quad \text{next_seq_num} < \text{send_base} + \text{window_size} \quad (1)$$

$$(2) \quad \text{send_base}_{\text{nuevo}} = \min \{ s \in \mathbb{N} \mid s \geq \text{send_base} \wedge s \notin \text{ACK}_{\text{recibidos}} \} \quad (2)$$

3.4. Módulos y clases clave

- **Client:** Inicializa el socket UDP, selecciona el protocolo y expone métodos `upload_file()` / `download_file()`.
- **Server:** Gestiona el socket UDP, arranca `ServerReceiver` y `ServerSender`, y coordina la finalización al recibir "exit".

- **ServerReceiver:**
 - `_receive_segment()`: deserializa `TransportProtocolSegment`.
 - `_dispatch_segment()`: crea o enruta el segmento al `UserManager`.
 - **ServerSender:** Lee de `send_queue` los segmentos a enviar y los envía con `socket.sendto()`.
 - **UserManager:** Punto de entrada para cada cliente; encapsula la lógica de alto nivel de `start_upload()`, `receive_file_from_client()`, `send_file_to_client()`, etc.
 - **StopAndWait / SelectiveRepeat:** Implementan las funciones de control de flujo, ventanas, timeouts y retransmisiones respectivas a cada protocolo.
 - **TransportProtocolSegment:** Representa la unidad básica de comunicación entre cliente y servidor. Cada segmento encapsula información de control (número de secuencia y flags como FIN, ACK) y un posible contenido de datos (payload). Está diseñada para serializar y deserializar los segmentos a una estructura binaria adecuada para ser transmitida por UDP, usando la biblioteca `struct`. También provee métodos auxiliares para crear segmentos especiales como ACK o FIN, y para identificar fácilmente el tipo de segmento recibido.

3.5. Flujo de control para UPLOAD

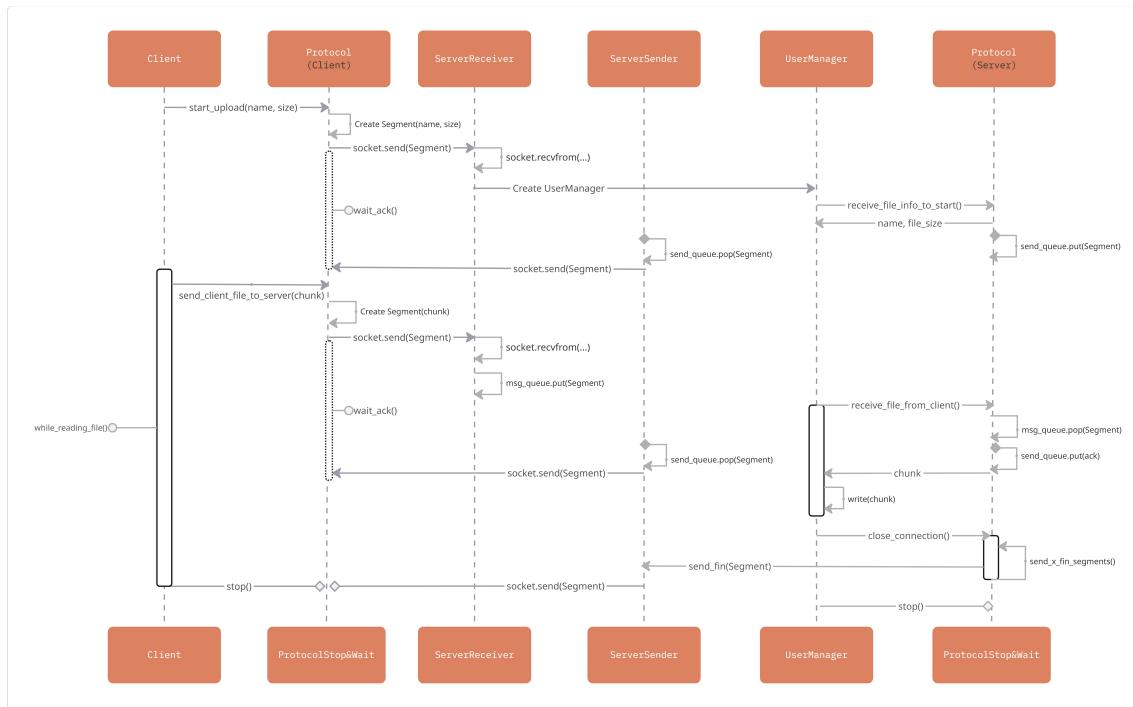


Figura 1: Secuencia del proceso de subida de archivos (UPLOAD)

1. Invoca `start_upload(nombre, tamaño)` del protocolo seleccionado (`StopAndWait` o `SelectiveRepeat`), que envía al servidor un segmento de inicio con la operación deseada y la metadata del archivo (nombre y tamaño).
 2. El `ServerReceiver` recibe este segmento inicial, crea un hilo `UserManager` asociado al cliente y este invoca `receive_file_info_to_start()`, que interpreta la metadata y confirma la operación.

3. El cliente abre el archivo local y lo parte en bloques (chunks). Para cada parte invoca `send_client_file_to_server(chunk)`, que:
 - Parte el chunk en segmentos con número de secuencia.
 - Envía cada segmento por UDP al servidor y, según el protocolo elegido:
 - En Stop-and-Wait: espera el ACK correspondiente antes de continuar.
 - En Selective Repeat: continúa enviando segmentos hasta llenar la ventana de envío.
 - Retransmite automáticamente si ocurre un timeout o llega un ACK inválido.
4. El servidor, desde el método `receive_file_from_client()` en `UserManager`, valida cada segmento recibido, extrae el contenido y lo escribe en un archivo dentro del directorio de almacenamiento. Además, responde con el ACK correspondiente.
5. Finalmente, cuando el servidor ha recibido todos los fragmentos correctamente y ha escrito el archivo completo, envía un conjunto de segmentos de tipo FIN al cliente para confirmar el cierre definitivo del flujo.
6. Ambas partes ejecutan `stop()` para liberar los recursos del protocolo y cerrar definitivamente la conexión.

3.5.1. Casos particulares con StopAndWait

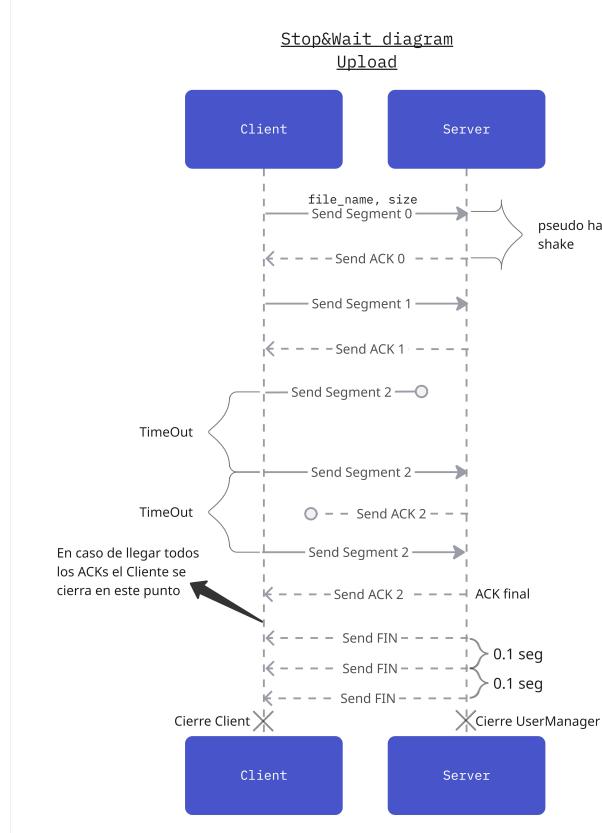


Figura 2: Posibles secuencias de envíos de segmentos

- **Pérdida de Segmento:** Cuando un segmento se pierde, el emisor no recibe el ACK correspondiente dentro del tiempo de espera (timeout), por lo que retransmite el segmento.

- **Pérdida de ACK:** Cuando un ACK se pierde, el emisor tampoco recibe confirmación dentro del timeout, así que también retransmite el mismo segmento, aunque el receptor ya lo haya recibido. Este lo detecta como duplicado y lo descarta
- **Final de conexión:** Cuando el emisor recibe el último ACK, entiende que el archivo terminó de enviarse, por lo que procede a cerrarse. Para el caso particular en el que el receptor envía el último ACK pero el mismo se pierda, el receptor, luego de enviar ese último ACK, envía 3 segmentos FIN espaciados temporalmente, de modo que si el emisor no recibió el ACK pero sí alguno de los FIN, entienda que el archivo terminó de enviarse correctamente y procede a cerrarse.

3.5.2. Casos particulares con SelectiveRepeat

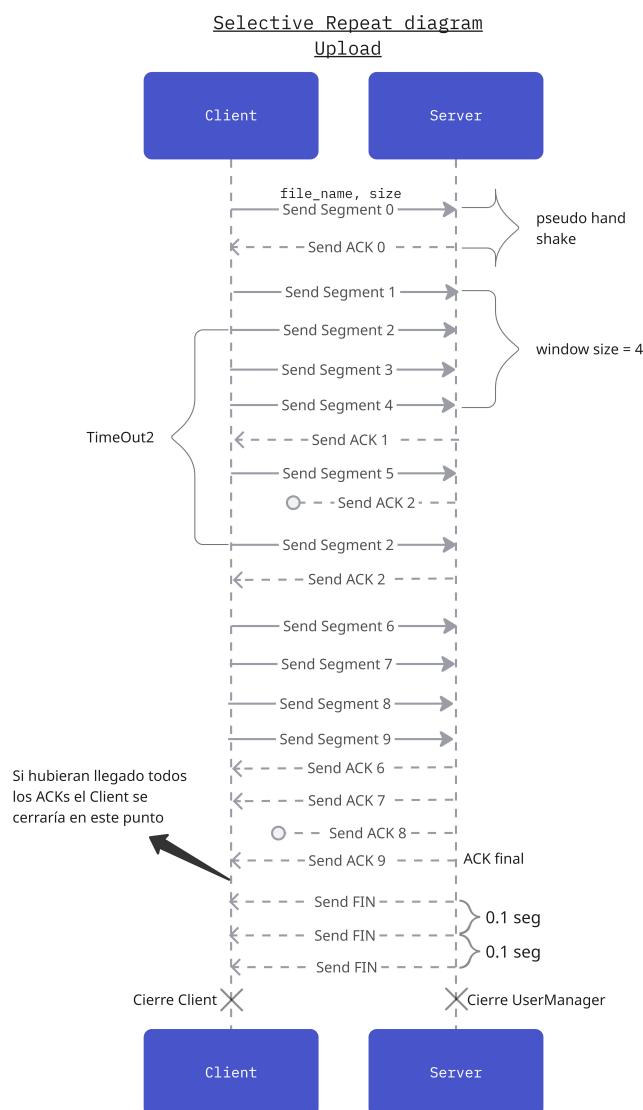


Figura 3: Posibles secuencias de envíos de segmentos

- **Pérdida de Segmento:** Al igual que en StopAndWait, cuando un segmento se pierde, el receptor no puede confirmar su recepción, por lo que no envía el ACK correspondiente. Al vencer el tiempo de espera (timeout) para ese segmento en particular, el emisor lo retransmite.

Sin embargo, a diferencia de StopAndWait, en Selective Repeat el emisor puede continuar enviando múltiples segmentos hasta alcanzar el tamaño máximo de su ventana de envío, sin necesidad de esperar un ACK por cada uno antes de continuar.

- **Pérdida de ACK:** La pérdida del ACK funciona de forma similar a StopAndWait. Si un ACK se pierde, el emisor no recibe confirmación de recepción dentro del timeout asignado al segmento correspondiente, por lo que lo retransmite. El receptor, al recibir un segmento duplicado, lo reconoce como tal y simplemente vuelve a enviar el ACK.
- **Avance de ventana:** Si el segmento o ACK perdido corresponde al número de secuencia más bajo dentro de la ventana (la base), entonces la ventana no podrá avanzar hasta que ese segmento sea correctamente retransmitido y reconocido mediante su ACK. Esto puede provocar un estancamiento temporal en el avance de la ventana, aún cuando se hayan recibido correctamente otros segmentos posteriores.
- **Final de conexión:** Cuando el emisor recibe todos los ACKs correspondientes, entiende que el archivo fue transferido correctamente y procede a cerrarse. Para el caso particular en el que el receptor envía el último ACK pero el mismo se pierde, el receptor, luego de enviar ese último ACK, envía 3 segmentos FIN espaciados temporalmente (cada 0.1 segundos), de modo que si el emisor no recibió el ACK pero sí alguno de los FIN, entienda que la transferencia finalizó correctamente y proceda a cerrarse.

3.6. Flujo de control para DOWNLOAD

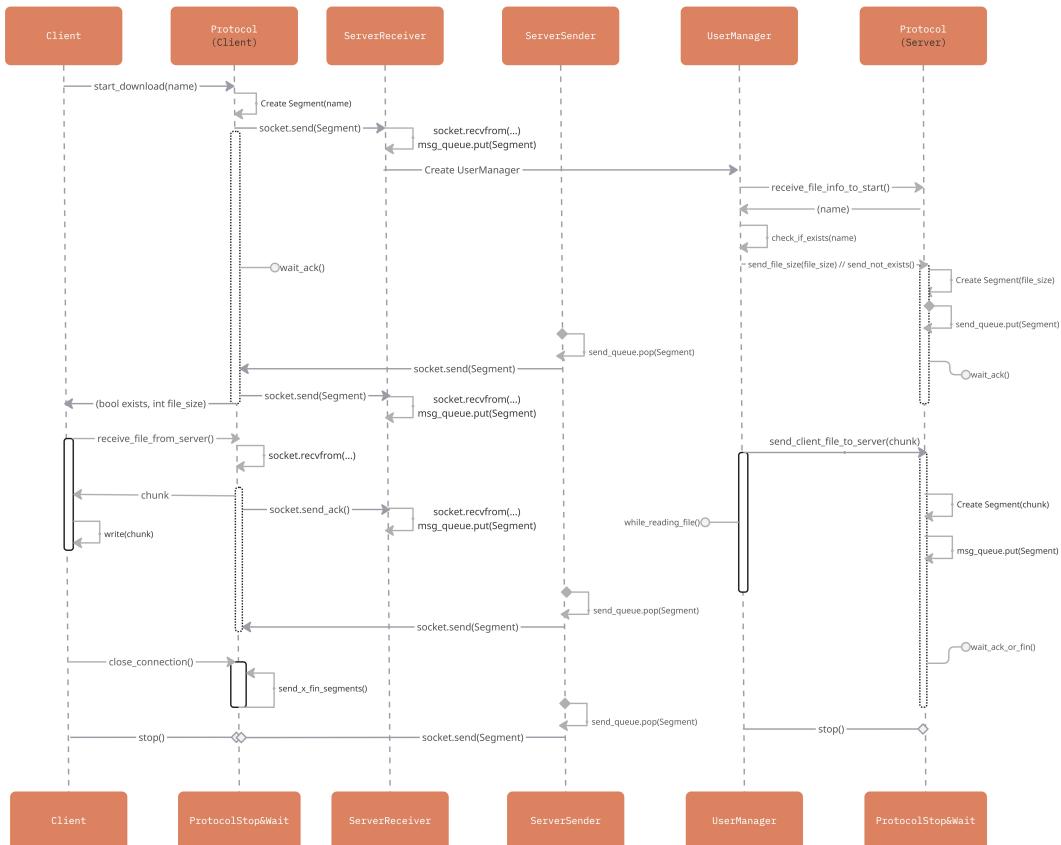


Figura 4: Secuencia del proceso de bajada de archivos (DOWNLOAD)

1. El cliente inicia la descarga invocando `start_download(nombre)` en el protocolo seleccionado, donde `nombre` es el archivo que desea obtener del servidor.
2. El protocolo construye un segmento de solicitud con tipo `DOWNLOAD` e incluye el nombre del archivo. Este segmento se envía al servidor por UDP.
3. El `ServerReceiver` recibe este segmento, crea un nuevo hilo `UserManager` y este llama a `receive_file_info_to_start()`, que verifica la existencia del archivo en el sistema. Si no existe, se envía un segmento `FIN` al respectivo cliente.
4. Si el archivo existe, el servidor obtiene su tamaño y responde con un segmento de tipo `ACK`, indicando al cliente que comenzará la transmisión e incluye en su payload el tamaño del archivo.
5. En caso que el cliente reciba el segmento `ACK`, crea el archivo local destino y queda a la espera de los datos. Caso contrario, si recibe el segmento `FIN` cierra la conexión.
6. El `UserManager` del servidor comienza la transmisión del archivo, partiendo en *chunks*. Luego usando alguno de los protocolos el chunk se segmenta y numera para ser enviado.
 - En **Stop-and-Wait**: el servidor envía un segmento y espera el `ACK` antes de continuar.
 - En **Selective Repeat**: el servidor envía múltiples segmentos hasta llenar la ventana, gestionando ACKs y retransmisiones según corresponda.
7. El cliente, en `receive_server_file()`, valida cada segmento recibido:
 - Extrae el número de secuencia y contenido.
 - Envía el `ACK` correspondiente.
 - Guarda el segmento en el archivo destino.
8. Una vez escrito todo el archivo, el cliente envía un conjunto de segmentos `FIN` al servidor para confirmar el cierre definitivo del flujo. En caso de recibirla, se termina la ejecución del `UserManager` correspondiente.
9. Ambas partes ejecutan `stop()` para liberar los recursos del protocolo y cerrar definitivamente la conexión.

3.6.1. Casos particulares con StopAndWait

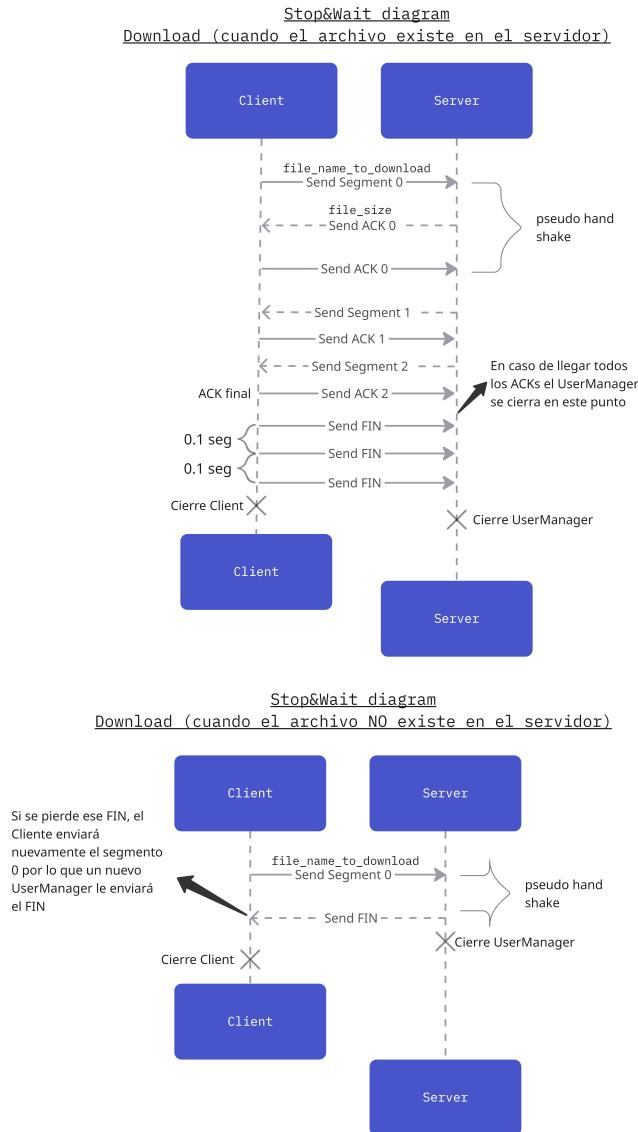


Figura 5: Posibles secuencias de descarga de segmentos (Misma Lógica que UPLOAD pero con un *pseudo-handshake* distinto)

- **Pseudo-handshake:** El cliente envía un Segmento 0 que contiene el nombre del archivo a descargar en su *payload* (`file_name_to_download`). Esto actúa como una especie de 'solicitud' al servidor. El servidor, al recibir esta solicitud, busca el archivo. Si lo encuentra, responde también con el Segmento 0, que contiene el tamaño del archivo en su *payload* (`file_size`), indicando que está disponible y se iniciará la transferencia. El cliente, al recibir este Segmento 0 de respuesta, envía un ACK 0, confirmando que recibió correctamente la información del tamaño y está listo para recibir los datos del archivo. Si el archivo no existe en el servidor, el mismo envía un Segmento FIN, indicando que el archivo no está disponible y finalizando la comunicación.
- **Reenvío de ACKs y Segmentos:** la lógica es la misma que la explicada en Upload de Stop And Wait
- **Final de conexión:** Cuando el emisor recibe todos los ACKs correspondientes, entiende que

el archivo fue transferido correctamente y procede a cerrarse. Para el caso particular en el que el receptor envía el último ACK pero el mismo se pierde, el receptor, luego de enviar ese último ACK, envía 3 segmentos FIN espaciados temporalmente (cada 0.1 segundos), de modo que si el emisor no recibió el ACK pero sí alguno de los FIN, entienda que la transferencia finalizó correctamente y proceda a cerrarse.

3.6.2. Casos particulares con SelectiveRepeat

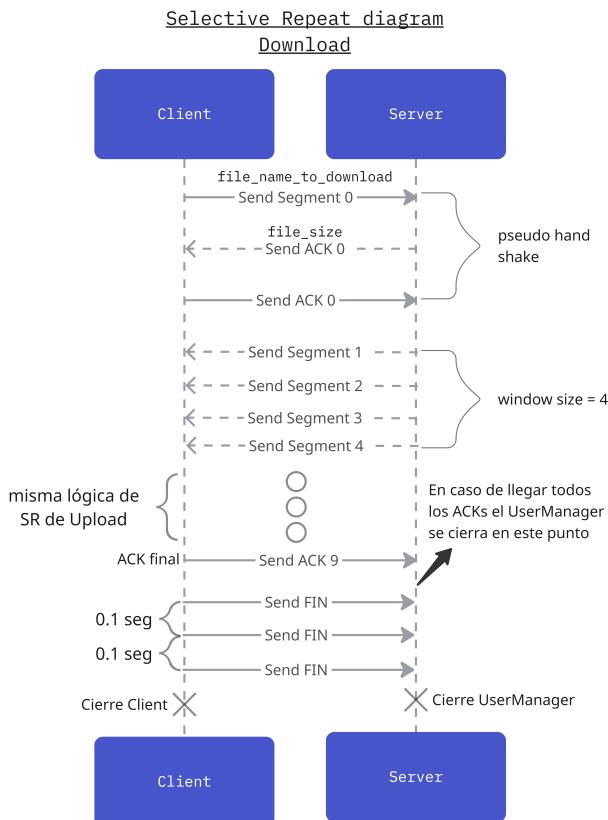


Figura 6: Posibles secuencias de descarga de segmentos (el *pseudo-handshake* es idéntico al de DOWNLOAD de StopAndWait)

- Tanto la logica de *handshake* como la de cierre se mantienen. Pero al igual que en Upload la logica de envio y reenvio de ACKs corresponde a la de Selective Repeat.

3.7. Decisiones sobre las constantes de configuración

Durante la implementación del protocolo, tomamos decisiones específicas respecto a las constantes utilizadas, con el objetivo de balancear rendimiento, confiabilidad y simplicidad. A continuación, se detallan los valores elegidos y las razones detrás de cada uno:

- WINDOW_SIZE = 16:** Se eligió un tamaño de ventana moderado que permite un buen nivel de paralelismo sin sobrecargar la red o la memoria. Un tamaño mayor podría incrementar la eficiencia en redes de alta latencia.
- START_SEQUENCE_NUMBER = 0:** Comenzamos la numeración de secuencias en 0 por simplicidad. Nuestro trabajo práctico no implementa aritmética modular para el número

de secuencia, ya que está pensado para la transmisión de archivos pequeños. Sin embargo, sería una mejora sencilla de incorporar.

- **HEADER_SIZE = 8 bytes:** Este tamaño fijo permite incluir los campos esenciales (número de secuencia, flags, etc.) sin agregar sobrecarga innecesaria.
- **MAX_SEGMENT_SIZE = 4096 bytes:** Se escogió un valor de 4KB para maximizar la cantidad de datos transmitidos por segmento, buscando eficiencia en redes locales de forma empírica. Sin embargo, somos conscientes de que este tamaño **provoca fragmentación a nivel IP** cuando se usa UDP sobre redes con MTU estándar de 1500 bytes, como Ethernet. Para evitar toda fragmentación IP, el tamaño máximo seguro hubiera sido de **1472 bytes**, calculado como 1500 (MTU) – 20 (IP header) – 8 (UDP header). En el contexto de este trabajo, que se ejecuta en un entorno controlado de red local (Mininet), aceptamos esta fragmentación ya que no afecta el funcionamiento del protocolo. Sin embargo, en un diseño destinado a redes heterogéneas o Internet, hubiera sido preferible fijar **MAX_SEGMENT_SIZE = 1472** para garantizar compatibilidad y evitar la pérdida de datagramas por fragmentación.
- **MAX_PAYLOAD_SIZE = MAX_SEGMENT_SIZE - HEADER_SIZE:** Definimos esta constante para ser utilizada como referencia en **CHUNK_SIZE** y para determinar la cantidad de segmentos en los que se debe dividir cada chunk.
- **CHUNK_SIZE = MAX_PAYLOAD_SIZE × 8:** Definir **CHUNK_SIZE** con este valor permite aprovechar el tamaño de payload máximo disponible de la mejor manera, evitando enviar segmentos con poca información y lograr así una división eficiente de los archivos.
- **TIMEOUT_SECONDS = 0.08 segundos:** Un timeout bajo (80 ms) permite reaccionar rápidamente ante pérdidas en redes locales, optimizando la retransmisión sin esperar demasiado.
- **RETRANSMIT_TIMER = 0.05 segundos:** Configurado ligeramente menor que el timeout, para que las retransmisiones sean ágiles y se minimice el tiempo de recuperación.
- **NUM_FIN_SEGMENTS_TO_SEND = 3:** Para cerrar la conexión de forma robusta, enviamos el segmento FIN tres veces, reduciendo el riesgo de que se pierda en la red.
- **BYTES_FILE_SIZE = 4, BYTES_FILE_NAME_SIZE = 2, BYTES_IS_UPLOAD = 1:** Definimos estos tamaños fijos para codificar el tamaño del archivo, el nombre y el flag de operación, permitiendo una estructura compacta y fácil de parsear.

Estas decisiones fueron tomadas considerando tanto la naturaleza del trabajo práctico como las características típicas de redes LAN, donde la latencia es baja y las tasas de pérdida son mínimas, pero donde igualmente es importante asegurar confiabilidad y eficiencia.

4. Pruebas

4.1. Metodología de Evaluación

Con el objetivo de evaluar el rendimiento de los protocolos **Stop-and-Wait (SW)** y **Selective Repeat (SR)** implementados en este trabajo, se diseñaron pruebas de transferencia de archivos en ambas direcciones (UPLOAD y DOWNLOAD). Las pruebas se ejecutaron bajo dos escenarios principales:

- **Sin pérdidas de paquetes**, en un entorno controlado.
- **Con pérdidas de paquetes**, mediante la introducción artificial de condiciones adversas.

Se utilizaron archivos de **5 MB** y **10 MB** para observar el comportamiento en volúmenes pequeños y moderados. La configuración base de los parámetros fue la siguiente:

El protocolo fue ajustado para maximizar la eficiencia de transmisión sin comprometer la estabilidad, buscando un equilibrio entre la ocupación del canal y el uso de memoria. Asimismo, los tiempos de espera y retransmisión fueron calibrados para asegurar una respuesta ágil ante pérdidas, manteniendo una buena relación entre confiabilidad y velocidad.

4.2. Resultados sin Pérdida de Paquetes

4.2.1. Subir un archivo de 5MB usando el protocolo Stop & Wait

```
ian@ian-VirtualBox:~/Escritorio/Redes/FIUBA-REDES-TP1$ python3 src/upload.py -H localhost -p 8080 -s 1.jpg -n hola2.jpg -r sw
[2025/05/08 00:05:45] - [upload INFO] - Client initialized with protocol sw.
[2025/05/08 00:05:45] - [upload INFO] - Uploading file hola2.jpg to the server
.
[2025/05/08 00:05:45] - [upload INFO] - Starting upload with hola2.jpg and size 5245329 bytes.
[2025/05/08 00:05:45] - [upload INFO] - File 1.jpg uploaded successfully.
[2025/05/08 00:05:45] - [upload INFO] - Client closed.
Upload completed in 0.21 seconds.
```

Figura 7: Subida de un archivo usando Stop & Wait (Archivo de 5 MB)

Como se observa en la imagen, el protocolo Stop & Wait logra subir el archivo de 5 MB de forma correcta y sin retransmisiones, completando la transferencia en aproximadamente 0.21 segundos.

4.2.2. Descargar un archivo de 5MB usando el protocolo Stop & Wait

```
ian@ian-VirtualBox:~/Escritorio/Redes/FIUBA-REDES-TP1$ python3 src/download.py -H localhost -p 8080 -d hola -n hola2.jpg -r sw
[2025/05/08 00:07:16] - [download INFO] - Client initialized with protocol sw.
[2025/05/08 00:07:16] - [download INFO] - Starting download with hola2.jpg.
[2025/05/08 00:07:16] - [download INFO] - File hola2.jpg downloaded successfully.
[2025/05/08 00:07:17] - [download INFO] - Client closed.
Download completed in 0.44 seconds.
```

Figura 8: Descarga de un archivo usando Stop & Wait (Archivo de 5 MB)

En la descarga, también sin pérdidas, se observa una transferencia fluida que se completa en aproximadamente 0.44 segundos, ligeramente más lenta que la subida.

4.2.3. Subir un archivo de 5MB usando el protocolo Selective Repeat

```
ian@ian-VirtualBox:~/Escritorio/Redes/FIUBA-REDES-TP1$ python3 src/upload.py -H localhost -p 8080 -s 1.jpg -n hola2.jpg -r sr
[2025/05/08 00:07:52] - [upload INFO] - Client initialized with protocol sr.
[2025/05/08 00:07:52] - [upload INFO] - Uploading file hola2.jpg to the server
.
[2025/05/08 00:07:52] - [upload INFO] - Starting upload with hola2.jpg and size 5245329 bytes.
[2025/05/08 00:07:52] - [upload INFO] - File 1.jpg uploaded successfully.
[2025/05/08 00:07:52] - [upload INFO] - Client closed.
Upload completed in 0.20 seconds.
```

Figura 9: Subida de un archivo usando Selective Repeat (Archivo de 5 MB)

La subida usando Selective Repeat muestra una performance similar a Stop & Wait, con una duración total cercana a los 0.20 segundos.

4.2.4. Descargar un archivo de 5MB usando el protocolo Selective Repeat

```
ian@ian-VirtualBox:~/Escritorio/Redes/FIUBA-REDES-TP1$ python3 src/download.py -H localhost -p 8080 -d hola -n hola2.jpg -r sr
[2025/05/08 00:08:12] - [download INFO] - Client initialized with protocol sr.
[2025/05/08 00:08:12] - [download INFO] - Starting download with hola2.jpg.
[2025/05/08 00:08:12] - [download INFO] - File hola2.jpg downloaded successfully.
[2025/05/08 00:08:12] - [download INFO] - Client closed.
Download completed in 0.36 seconds.
```

Figura 10: Descarga de un archivo usando Selective Repeat (Archivo de 5 MB)

La descarga con Selective Repeat tuvo un tiempo total de aproximadamente 0.36 segundos.

4.3. Resultados con Pérdida de Paquetes

4.3.1. Subir un archivo de 5MB usando el protocolo Stop & Wait

```
root@agustin-pc:/home/agustin/Documentos/redes/FIUBA-REDES-TP1# python3 src/upload.py -H 192.168.1.2 -p 8080 -r sw -s cinco.jpg -n c.jpg
[2025/05/06 01:18:45] - [upload INFO] - Client initialized with protocol sw.
[2025/05/06 01:18:45] - [upload INFO] - Uploading file c.jpg to the server.
[2025/05/06 01:18:45] - [upload INFO] - Starting upload with c.jpg and size 5245329 bytes.
[2025/05/06 01:19:40] - [upload INFO] - File cinco.jpg uploaded successfully.
[2025/05/06 01:19:40] - [upload INFO] - Client closed.
Upload completed in 54.33 seconds.
```

Figura 11: Subida de un archivo usando Stop & Wait

Como se puede ver en la imagen, subir un archivo de 5MB utilizando el protocolo Stop and Wait tardó 54.33 segundos.

4.3.2. Descargar un archivo de 5MB usando el protocolo Stop & Wait

Ahora veamos cuánto tarda la descarga de un archivo de 5MB utilizando el protocolo Stop and Wait.

```
root@agustin-pc:/home/agustín/Documentos/redes/FIUBA-REDES-TP1# python3 src/download.py -H 192.168.1.2 -p 8080 -r sw -d salida -n c.jpg
[2025/05/06 01:20:51] - [download INFO] - Client initialized with protocol sw.
[2025/05/06 01:20:51] - [download INFO] - Starting download with c.jpg.
[2025/05/06 01:21:46] - [download INFO] - File c.jpg downloaded successfully.
[2025/05/06 01:21:46] - [download INFO] - Client closed.
Download completed in 55.61 seconds.
```

Figura 12: Descarga de un archivo usando Stop & Wait

La descarga de un archivo de 5MB usando Stop and Wait tardó 55.61 segundos.

4.3.3. Subir un archivo de 5MB usando el protocolo Selective Repeat

Ahora procedemos a probar el protocolo Selective Repeat subiendo un archivo de 5MB.

```
root@agustin-pc:/home/agustín/Documentos/redes/FIUBA-REDES-TP1# python3 src/upload.py -H 192.168.1.2 -p 8080 -r sr -s cinco.jpg -n c.jpg
[2025/05/06 01:12:48] - [upload INFO] - Client initialized with protocol sr.
[2025/05/06 01:12:48] - [upload INFO] - Uploading file c.jpg to the server.
[2025/05/06 01:12:48] - [upload INFO] - Starting upload with c.jpg and size 5245
329 bytes.
[2025/05/06 01:13:23] - [upload INFO] - File cinco.jpg uploaded successfully.
[2025/05/06 01:13:23] - [upload INFO] - Client closed.
Upload completed in 34.80 seconds.
```

Figura 13: Subida de un archivo usando Selective Repeat

En este caso el tiempo requerido para completar la tarea fue de 34.80 segundos. Notar que el tiempo es menor que el protocolo Stop and Wait.

4.3.4. Descargar un archivo de 5MB usando el protocolo Selective Repeat

Finalmente probamos descargar un archivo de 5MB usando Selective Repeat.

```
root@agustin-pc:/home/agustín/Documentos/redes/FIUBA-REDES-TP1# python3 src/download.py -H 192.168.1.2 -p 8080 -r sr -d salida -n c.jpg
[2025/05/06 01:15:06] - [download INFO] - Client initialized with protocol sr.
[2025/05/06 01:15:06] - [download INFO] - Starting download with c.jpg.
[2025/05/06 01:15:45] - [download INFO] - File c.jpg downloaded successfully.
[2025/05/06 01:15:46] - [download INFO] - Client closed.
Download completed in 39.19 seconds.
```

Figura 14: Subida de un archivo usando Selective Repeat

Obtuvimos un tiempo de 39.19 segundos descargando un archivo de 5MB con Selective Repeat.

4.3.5. Subir un archivo de 10MB usando el protocolo Stop & Wait

```
root@agustin-pc:/home/agustín/Documentos/redes/FIUBA-REDES-TP1# python3 src/upload.py -H 192.168.1.2 -p 8080 -r sw -s pesada.jpg -n d.jpg
[2025/05/06 01:32:09] - [upload INFO] - Client initialized with protocol sw.
[2025/05/06 01:32:09] - [upload INFO] - Uploading file d.jpg to the server.
[2025/05/06 01:32:09] - [upload INFO] - Starting upload with d.jpg and size 1017
4706 bytes.
[2025/05/06 01:33:57] - [upload INFO] - File pesada.jpg uploaded successfully.
[2025/05/06 01:33:57] - [upload INFO] - Client closed.
Upload completed in 108.23 seconds.
```

Figura 15: Subida de un archivo usando Stop & Wait

4.3.6. Descargar un archivo de 10MB usando el protocolo Stop & Wait

```
root@agustin-pc:/home/agustin/Documentos/redes/FIUBA-REDES-TP1# python3 src/download.py -H 192.168.1.2 -p 8080 -r sw -d salida -n d.jpg
[2025/05/06 01:34:21] - [download INFO] - Client initialized with protocol sw.
[2025/05/06 01:34:21] - [download INFO] - Starting download with d.jpg.
[2025/05/06 01:36:02] - [download INFO] - File d.jpg downloaded successfully.
[2025/05/06 01:36:02] - [download INFO] - Client closed.
Download completed in 101.33 seconds.
```

Figura 16: Descargar un archivo usando Stop & Wait

4.3.7. Subir un archivo de 10MB usando el protocolo Selective Repeat

```
root@agustin-pc:/home/agustin/Documentos/redes/FIUBA-REDES-TP1# python3 src/upload.py -H 192.168.1.2 -p 8080 -r sr -s pesada.jpg -n d.jpg
[2025/05/06 01:28:00] - [upload INFO] - Client initialized with protocol sr.
[2025/05/06 01:28:00] - [upload INFO] - Uploading file d.jpg to the server.
[2025/05/06 01:28:00] - [upload INFO] - Starting upload with d.jpg and size 1017
4706 bytes.
[2025/05/06 01:29:10] - [upload INFO] - File pesada.jpg uploaded successfully.
[2025/05/06 01:29:10] - [upload INFO] - Client closed.
Upload completed in 69.46 seconds.
```

Figura 17: Subir un archivo usando Selective Repeat

4.3.8. Descargar un archivo de 10MB usando el protocolo Selective Repeat

```
root@agustin-pc:/home/agustin/Documentos/redes/FIUBA-REDES-TP1# python3 src/download.py -H 192.168.1.2 -p 8080 -r sr -d salida -n d.jpg
[2025/05/06 01:30:22] - [download INFO] - Client initialized with protocol sr.
[2025/05/06 01:30:22] - [download INFO] - Starting download with d.jpg.
[2025/05/06 01:31:31] - [download INFO] - File d.jpg downloaded successfully.
[2025/05/06 01:31:31] - [download INFO] - Client closed.
Download completed in 69.02 seconds.
```

Figura 18: Subir un archivo usando Selective Repeat

5. Análisis

5.1. Tabla sobre los resultados sin Pérdida de Paquetes

Protocolo	Dirección	Tamaño Archivo	Tiempo Total
Stop-and-Wait	UPLOAD	5 MB	~0.21 segundos
Stop-and-Wait	DOWNLOAD	5 MB	~0.44 segundos
Selective Repeat	UPLOAD	5 MB	~0.20 segundos
Selective Repeat	DOWNLOAD	5 MB	~0.36 segundos

Cuadro 1: Tiempos de transferencia sin pérdida de paquetes (Archivo de 5 MB)

5.2. Tabla sobre los resultados con Pérdida de Paquetes

Protocolo	Dirección	Tamaño Archivo	Tiempo Total
Stop-and-Wait	UPLOAD	5 MB	~54.33 segundos
Stop-and-Wait	DOWNLOAD	5 MB	~55.61 segundos
Stop-and-Wait	UPLOAD	10 MB	~108.23 segundos
Stop-and-Wait	DOWNLOAD	10 MB	~101.33 segundos
Selective Repeat	UPLOAD	5 MB	~34.80 segundos
Selective Repeat	DOWNLOAD	5 MB	~39.19 segundos
Selective Repeat	UPLOAD	10 MB	~69.46 segundos
Selective Repeat	DOWNLOAD	10 MB	~69.02 segundos

Cuadro 2: Tiempos de transferencia sin pérdida de paquetes (Archivos de 5 y 10 MB)

5.3. Interpretación de los resultados obtenidos

En un escenario sin pérdidas de paquetes, ambos protocolos —Stop-and-Wait y Selective Repeat— exhiben un rendimiento prácticamente idéntico, con tiempos de transferencia cercanos a los tres segundos para un archivo de 5 MB. Esta similitud se debe a que, en condiciones ideales, la simplicidad de Stop-and-Wait no representa una desventaja significativa, ya que cada segmento enviado es rápidamente reconocido, y no se produce ningún tipo de estancamiento en la comunicación.

Selective Repeat, por su parte, posee una arquitectura más compleja, basada en el manejo de ventanas y buffers, pero en ausencia de pérdida esta ventaja estructural no se ve reflejada en mejoras concretas de performance. Al no tener que gestionar retransmisiones, ambos protocolos pueden operar de forma continua y mantener el canal ocupado de manera eficiente.

De esta forma, los resultados obtenidos bajo esas condiciones permiten validar las hipótesis planteadas, en particular la de rendimiento y confiabilidad. Para archivos de 5 y 10 MB, ambos protocolos logran completar exitosamente la transferencia, confirmando la hipótesis de confiabilidad: no se detectaron errores de integridad, y la transmisión concluyó correctamente en todos los casos.

En cuanto al rendimiento, se observa que en un entorno ideal, Stop-and-Wait (SW) alcanza tiempos competitivos, especialmente para archivos de tamaño moderado. Esto se debe a que la ausencia de pérdidas y la baja latencia permiten que la simple mecánica de espera y confirmación no introduzca penalizaciones significativas. El canal se mantiene ocupado de forma razonablemente eficiente, ya que cada ACK llega a tiempo para habilitar el envío del siguiente segmento sin pausas innecesarias.

Sin embargo, Selective Repeat demuestra una mayor eficiencia, particularmente en las transferencias de 10 MB, donde la diferencia de tiempos se vuelve más notoria (por ejemplo, ~108s vs. ~69s en subida). Esta mejora valida la hipótesis de rendimiento, ya que la arquitectura de ventana deslizante permite explotar mejor el ancho de banda disponible, aún sin necesidad de recurrir a retransmisiones. Aunque la diferencia no es drástica en archivos pequeños, se amplifica con el tamaño del archivo, anticipando que Selective Repeat escalaría mejor frente a volúmenes más grandes o redes con mayores latencias.

Además, el comportamiento observado sugiere que la eficiencia relativa de Selective Repeat aumenta con el tamaño del archivo, lo que respalda también la hipótesis de escalabilidad. Si bien el costo computacional y estructural de Selective Repeat es mayor (buffers, temporizadores por paquete, reordenamiento), este se justifica cuando el volumen de datos o las condiciones de red lo requieren.

6. Preguntas

6.1. Arquitectura Cliente-Servidor

La arquitectura Cliente-Servidor es un modelo de diseño en el que las funciones se dividen entre proveedores de recursos o servicios, denominados *servidores*, y consumidores de estos recursos, denominados *clientes*. Esta interacción se realiza a través de una red, como puede ser Internet o una red local. En este esquema, los clientes inician la comunicación enviando solicitudes, y los servidores responden procesando dichas solicitudes y devolviendo los resultados correspondientes. Esta conexión posibilita una comunicación continua y bidireccional, permitiendo que tanto el cliente como el servidor puedan enviar y recibir datos entre sí.

Resulta evidente que en este modelo el cliente depende completamente de la disponibilidad del servidor; sin él, no podría funcionar. A su vez, el servidor carecería de sentido si no existieran clientes que hicieran uso de sus servicios. En este contexto, ambas partes son interdependientes: una no tiene razón de ser sin la otra.

Uno de los puntos más característicos de la arquitectura Cliente-Servidor es la centralización de los datos, pues el server recibe, procesa y almacena todos los datos provenientes de todos los clientes.

En el contexto de nuestra aplicación de transferencia de archivos (operaciones de subida y descarga), este modelo se implementa de la siguiente manera:

Servidor

El servidor opera como un nodo central encargado de recibir y procesar las solicitudes provenientes de múltiples clientes. En nuestra implementación, el servidor utiliza sockets basados en el protocolo UDP para la comunicación. Para permitir el manejo concurrente de múltiples clientes, cada solicitud es gestionada en un hilo independiente, representado por la clase `ClientHandler`.

Esta estructura multihilo permite realizar múltiples transferencias de archivos de forma simultánea, sin bloquear la atención de nuevos clientes. Según el tipo de solicitud recibida, el servidor puede ejecutar dos operaciones principales: enviar un archivo solicitado (descarga) o recibir un archivo enviado por el cliente (subida).

Cliente

El cliente es el componente que inicia la comunicación con el servidor para solicitar o enviar archivos. Nuestra aplicación permite que el cliente ejecute dos tipos de operación:

- **Subida de archivos (upload):** Antes de iniciar la transferencia, el cliente valida que el archivo exista y cumpla con las restricciones de tamaño establecidas. Para garantizar la entrega confiable de los paquetes sobre UDP, se utiliza un protocolo de control de errores como Stop-and-Wait o Selective Repeat, los cuales aseguran la correcta recepción de los datos por parte del servidor.
- **Descarga de archivos (download):** El cliente solicita al servidor un archivo determinado. El servidor responde enviando el archivo en fragmentos o bloques de datos. El cliente, por su parte, reconstruye el archivo a medida que va recibiendo y confirmando cada paquete, asegurando así la integridad del contenido descargado.

Esta arquitectura permite escalar el sistema para soportar múltiples transferencias simultáneas, manteniendo un diseño modular y fácilmente extensible.

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

Un protocolo de capa de aplicación es un conjunto de reglas y estándares que permiten a las aplicaciones en diferentes dispositivos comunicarse de manera efectiva a través de una red. Este tipo de protocolo define cómo las aplicaciones deben intercambiar datos y coordinar sus interacciones, asegurando que ambas partes entiendan y procesen la información correctamente.

Sus funciones principales incluyen:

- **Definición del Formato de Datos:** Especifica cómo los datos deben estructurarse, empaquetarse y transmitirse para que las aplicaciones puedan interpretarlos correctamente.
- **Control de Sesiones:** Gestiona el inicio, mantenimiento y finalización de las sesiones de comunicación entre aplicaciones, asegurando que las interacciones sean coherentes y ordenadas.
- **Gestión de Seguridad:** Proporciona mecanismos para implementar medidas de seguridad como autenticación, cifrado y control de acceso.

6.3. Detalle el protocolo de aplicación desarrollado en este trabajo.

El protocolo de aplicación desarrollado en este trabajo tiene como objetivo permitir la transferencia confiable de archivos entre un cliente y un servidor, utilizando el protocolo de transporte UDP. Dado que UDP no garantiza la entrega ni el orden de los paquetes, se implementaron mecanismos adicionales para asegurar la integridad de los datos y la sincronización entre las partes involucradas.

Operaciones Soportadas

- **Subida de Archivos (Upload):** El cliente envía un archivo al servidor, que lo almacena en un directorio predefinido.
- **Descarga de Archivos (Download):** El cliente solicita un archivo al servidor, que lo fragmenta y envía en partes.

Protocolos de Transferencia

Se implementan dos protocolos confiables para manejar la transferencia de archivos:

- **Stop-and-Wait (SW):** Implementa un protocolo simple donde cada segmento debe ser confirmado antes de enviar el siguiente. Incluye retransmisión automática en caso de timeout.
- **Selective Repeat (SR):** Utiliza una ventana deslizante de tamaño fijo (WINDOW_SIZE) para permitir la transmisión de múltiples segmentos simultáneamente. Mantiene buffers separados para segmentos enviados y recibidos, y utiliza un hilo dedicado para las retransmisiones.

Estructura de los Segmentos

Cada segmento transmitido contiene los siguientes campos:

- **Header (8 bytes):**
 - Número de secuencia (4 bytes)
 - Flags (1 byte) - bits para FIN y ACK
 - Padding (3 bytes)
- **Payload:** Datos del archivo o metainformación (tamaño variable)

Arquitectura del Servidor

El servidor implementa una arquitectura multi-hilo con los siguientes componentes:

- **ServerReceiver:** Maneja la recepción inicial de conexiones y crea un UserManager para cada cliente.
- **ServerSender:** Gestiona el envío de segmentos a los clientes.
- **UserManager:** Coordina la transferencia de archivos para cada cliente individual.
- **Colas de Mensajes:** Facilitan la comunicación entre componentes.

Manejo de Errores y Confiabilidad

- Retransmisión automática de segmentos perdidos
- Temporizadores (TIMEOUT_SECONDS) para detectar pérdidas
- Buffers para manejar segmentos fuera de orden
- Manejo de desconexiones y limpieza de recursos
- Logging detallado de eventos y errores

Flujo del Protocolo

1. Subida de Archivos (Upload)

Cliente:

- Inicia enviando información del archivo (tamaño, nombre)
- parte el archivo en chunks de tamaño CHUNK_SIZE
- Envía los segmentos según el protocolo seleccionado (SW o SR), que a su vez parte el chunk en segmentos de tamaño MAX_SEGMENT_SIZE
- Espera confirmaciones y maneja retransmisiones

Servidor:

- Recibe la información inicial del archivo
- Crea un UserManager para manejar la transferencia
- Recibe y almacena los segmentos
- Envía ACKs por cada segmento recibido
- Maneja la reconstrucción del archivo
- Finaliza con segmentos FIN

2. Descarga de Archivos (Download)

Cliente:

- Envía solicitud con nombre del archivo
- Recibe el tamaño del archivo o indicación de que no existe (segmento con flag FIN)
- Recibe y reconstruye el archivo
- Maneja la recepción de segmentos fuera de orden
- finaliza con segmentos FIN

Servidor:

- Verifica la existencia del archivo
- Envía tamaño del archivo o segmento FIN si no existe
- Parte y envía el archivo
- Maneja retransmisiones y confirmaciones

Mecanismos de Confiabilidad

Stop-and-Wait:

- Envío secuencial de segmentos
- Espera de ACK antes de enviar el siguiente
- Retransmisión tras timeout

Selective Repeat:

- Ventana deslizante de tamaño fijo
- Buffers separados para envío y recepción
- Hilo dedicado para retransmisiones
- Manejo de segmentos fuera de orden
- ACKs selectivos para cada segmento

6.4. La capa de transporte del stack TCP/IP ofrece dos protocolos: TCP y UDP. ¿Qué servicios proveen dichos protocolos? ¿Cuáles son sus características? ¿Cuándo es apropiado utilizar cada uno?

6.4.1. TCP (Transmission Control Protocol)

El protocolo TCP proporciona un servicio de comunicación **orientado a la conexión**, es decir, se establece una conexión confiable entre emisor y receptor antes de transmitir los datos.

Servicios y características principales

- Entrega de datos confiable y en orden.
- Control de flujo y control de congestión.
- Retransmisión de paquetes perdidos o corruptos.
- Detección de errores.

Cuándo utilizar TCP

TCP es adecuado cuando la integridad y confiabilidad de los datos son críticas. Se utiliza en aplicaciones como:

- Transferencia de archivos (ej. FTP).
- Correos electrónicos (ej. SMTP).
- Aplicaciones web (HTTP/HTTPS).

6.4.2. UDP (User Datagram Protocol)

UDP ofrece un servicio **no orientado a la conexión**, el cual no garantiza la entrega ni el orden de los datos y no incluye mecanismos de control de flujo o retransmisión.

Servicios y características principales

- Baja sobrecarga, mayor velocidad.
- No garantiza la entrega ni el orden.
- Sin control de congestión ni flujo.

Cuándo utilizar UDP

UDP es útil en aplicaciones donde la velocidad es prioritaria sobre la fiabilidad, como:

- Transmisiones en vivo (streaming).
- Videollamadas.
- Gaming.

7. Dificultades encontradas

Durante el desarrollo del proyecto, nos enfrentamos a diversas dificultades que requirieron iteraciones constantes y replanteos en nuestra solución. A continuación, detallamos los principales desafíos:

- **Diseño inicial de la aplicación:** Una de las primeras dificultades fue definir una arquitectura modular y escalable que permitiera implementar y comparar distintos protocolos de transferencia de archivos. Determinar cómo estructurar las entidades principales (cliente, servidor, lógica del protocolo, manejo de archivos y comunicación) y establecer responsabilidades claras para cada componente requirió tiempo y pruebas.

- **Sincronización entre cliente y servidor:** Uno de los problemas más desafiantes fue lograr una correcta sincronización entre ambas partes durante la transferencia de datos. Resultó complejo coordinar los números de secuencia y los ACKs, ya que cualquier desfasaje entre cliente y servidor podía generar duplicación de datos, pérdidas o incluso dejar la comunicación en un estado inconsistente. Fue necesario revisar en profundidad cómo se enviaban y recibían las confirmaciones, y ajustar varios detalles para evitar situaciones de bloqueo o retransmisiones incorrectas.
- **Soporte para pérdida de paquetes:** Inicialmente, la aplicación fue desarrollada bajo condiciones ideales, sin considerar explícitamente la pérdida de paquetes. Al introducir este factor para pruebas más realistas, surgieron múltiples fallos, desde retransmisiones ineficientes hasta problemas de bloqueo en la comunicación. Fue necesario ajustar cuidadosamente el manejo de temporizadores (*timeouts*), asegurar la correcta retransmisión de paquetes perdidos y validar que el receptor pudiera identificar duplicados y responder apropiadamente.
- **Implementación de Selective Repeat:** A diferencia de Stop-and-Wait, Selective Repeat fue mucho más complicado al requerir el manejo de una ventana deslizante, múltiples temporizadores, buffer de recepción fuera de orden y mecanismos para aceptar y reordenar paquetes. Más difícil aún fue la depuración de errores en este protocolo, ya que los fallos no siempre se manifestaban de forma inmediata y dependían de condiciones específicas de pérdida o timeout.
- **Pruebas en condiciones adversas:** Otro de los desafíos fue aprender a utilizar Mininet para simular una red con pérdidas. Al principio, entender cómo configurar correctamente estos escenarios no fue trivial, ya que requería familiarizarse con comandos específicos y con la lógica del entorno virtualizado. Una vez superada esa curva de aprendizaje, pudimos reproducir las distintas situaciones y testear cómo se comportaban nuestros protocolos bajo condiciones no ideales. Esto nos ayudó a identificar varias fallas que no aparecían en los casos felices y a mejorar la robustez general de la aplicación.

8. Conclusion

El desarrollo de esta aplicación cliente-servidor para la transferencia de archivos a través de la red ha sido un proyecto desafiante, que nos permitió profundizar en los fundamentos de redes y protocolos de comunicación, tanto desde un enfoque teórico como práctico.

A lo largo del proyecto, exploramos las principales características necesarias para lograr una transferencia de archivos confiable. El protocolo Stop-and-Wait se destacó por su simplicidad y facilidad de implementación, siendo adecuado para escenarios con bajo tráfico o condiciones de red ideales.

Por otro lado, Selective Repeat demostró ser una solución más robusta y eficiente. Gracias al uso de ventanas deslizantes y retransmisiones selectivas, logró mantener el canal ocupado de forma continua, lo que se traduce en una mejora del rendimiento principalmente bajo condiciones no ideales. Esta eficiencia adicional, sin embargo, vino acompañada de una mayor complejidad en su diseño e implementación, debido a la gestión de múltiples temporizadores, buffers y reordenamiento de paquetes.

Durante el desarrollo de la aplicación enfrentamos diversos desafíos técnicos, los cuales fueron fundamentales para consolidar nuestra comprensión sobre la teoría aprendida en clase. Si bien ya contábamos con conocimientos previos sobre el manejo de sockets y la programación concurrente con hilos, este proyecto nos permitió aplicar esas habilidades en un contexto más exigente, integrándolas con el diseño e implementación de protocolos personalizados.

Los resultados obtenidos no solo confirmaron la confiabilidad de ambos protocolos, sino que también ofrecieron evidencia concreta sobre sus diferencias en cuanto a rendimiento y escalabilidad. En definitiva, si se espera una red con condiciones estables, con baja latencia y poca pérdida de paquetes, el protocolo Stop-and-Wait puede ser una opción válida por su simplicidad y facilidad

de implementación. Sin embargo, ante escenarios con alta tasa de pérdida, latencia o mayores exigencias de rendimiento, el protocolo Selective Repeat demuestra claras ventajas gracias a su eficiencia y capacidad de mantener el canal más ocupado de forma continua. En contextos reales, donde las condiciones de red pueden variar significativamente, Selective Repeat se posiciona como una solución más robusta y escalable, aunque Stop-and-Wait no debe ser descartado en situaciones controladas o de baja complejidad.

Este trabajo deja una base firme para futuras evaluaciones en escenarios más complejos, como redes con pérdida de paquetes o condiciones adversas, donde se espera que la ventaja de Selective Repeat se acentúe aún más.

9. Anexo: Fragmentación IPv4

Con el objetivo de comprender y asimilar el proceso de fragmentación en IPv4, creamos una red virtual utilizando la herramienta *mininet* con una determinada topología, y generamos tráfico entre los hosts de la misma para observar el comportamiento del protocolo de capa de red.

9.1. Topología de red virtual

Utilizamos una topología lineal, la cual podemos observar en la siguiente figura:

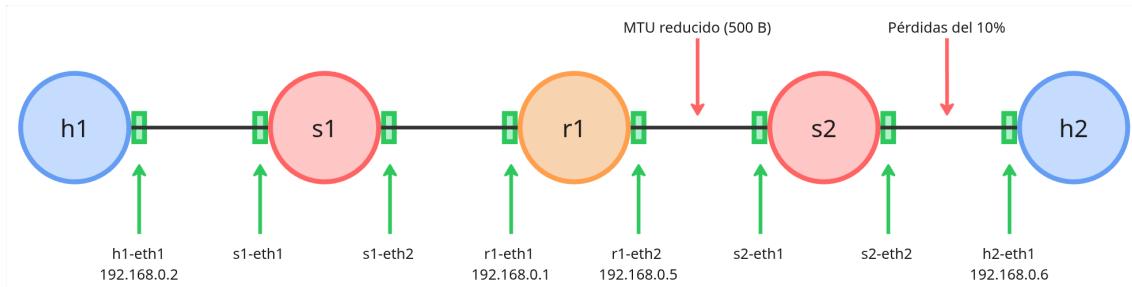


Figura 19: Topología de red utilizada

donde:

- *h1* y *h2* son *hosts* (en azul)
- *s1* y *s2* son *link-layer switches* (en rojo)
- *r1* es un *router* (en naranja)
- En verde, las interfaces de cada nodo

Como característica particular, cabe destacar que el enlace que conecta *r1* con *s2* posee un MTU limitado a 500 bytes, a diferencia del resto de enlaces los cuales tienen un MTU de 1500 bytes. El enlace que conecta *s2* con *h2* tiene una pérdida de paquetes del 10%. El MTU reducido del enlace es una condición necesaria para que se produzca la fragmentación.

9.2. Generación y análisis de tráfico UDP por la red

Para producir tráfico UDP, utilizamos la herramienta *iperf*. Elegimos *h2* como servidor y en él ejecutamos:

```
iperf -s -u
```

Seleccionamos a *h1* para actuar como cliente, y en él ejecutamos:

```
iperf -c 192.168.0.6 -n 144K -u
```

Es decir, *h1* le envía 144 kilobytes de datos a *h2*.

Al mismo tiempo que ejecutamos estos dos comandos, hicimos dos capturas de paquetes con Wireshark, una en la interfaz *s1-eth1* (antes de la fragmentación y pérdida de paquetes) y la otra en la interfaz *s2-eth2* (luego de la fragmentación y pérdida de paquetes). (Nota: las capturas obtenidas están adjuntadas en el mismo archivo .zip donde se encuentra este informe)

En el caso de *s1 – eth1*, obtuvimos una captura que muestra lo siguiente:

100	1.099...	192.168.0.2	192.168.0.6	UDP	1512 50723 → 5001 Len=1470
101	1.110...	192.168.0.2	192.168.0.6	UDP	498 50723 → 5001 Len=456
102	1.121...	192.168.0.2	192.168.0.6	UDP	1512 50723 → 5001 Len=1470
103	1.134...	192.168.0.2	192.168.0.6	UDP	1512 50723 → 5001 Len=1470
104	1.144...	192.168.0.2	192.168.0.6	UDP	1512 50723 → 5001 Len=1470
105	1.154...	192.168.0.2	192.168.0.6	UDP	1512 50723 → 5001 Len=1470
106	1.158...	192.168.0.6	192.168.0.2	UDP	170 5001 → 50723 Len=128
107	1.158...	192.168.0.6	192.168.0.2	UDP	170 5001 → 50723 Len=128
108	1.158...	192.168.0.2	192.168.0.6	ICMP	198 Destination unreachable (Port unreachable)

Figura 20: Recorte de la captura de segmentos UDP en *s1 – eth1*

Observamos que *h1* envió 105 segmentos UDP de 1470 bytes a *h2*. No hay fragmentación de datagramas todavía, ya que el MTU no se encuentra limitado en el enlace que conecta *h1* con *s1*.

Por su parte, la captura en $s2 - eth2$ permite observar lo siguiente:

355	1.075...	192.168.0.2	192.168.0.6	IPv4	72	Fragmented IP protocol (proto=UDP 17, off=1440, ID=051c)
356	1.086...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=0, ID=051d)
357	1.086...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=480, ID=051d)
358	1.086...	192.168.0.2	192.168.0.6	IPv4	72	Fragmented IP protocol (proto=UDP 17, off=1440, ID=051d)
359	1.098...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=0, ID=051e) [Reassembled in #362]
360	1.098...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=480, ID=051e) [Reassembled in #362]
361	1.098...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=960, ID=051e) [Reassembled in #362]
362	1.098...	192.168.0.2	192.168.0.6	UDP	72	50723 → 5001 Len=1470
363	1.109...	192.168.0.2	192.168.0.6	UDP	498	50723 → 5001 Len=456
364	1.120...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=480, ID=0520)
365	1.120...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=960, ID=0520)
366	1.120...	192.168.0.2	192.168.0.6	IPv4	72	Fragmented IP protocol (proto=UDP 17, off=1440, ID=0520)
367	1.133...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=0, ID=0521)
368	1.133...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=480, ID=0521)
369	1.133...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=960, ID=0521)
370	1.143...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=0, ID=0522) [Reassembled in #373]
371	1.143...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=480, ID=0522) [Reassembled in #373]
372	1.143...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=960, ID=0522) [Reassembled in #373]
373	1.143...	192.168.0.2	192.168.0.6	UDP	72	50723 → 5001 Len=1470
374	1.153...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=0, ID=0523) [Reassembled in #377]
375	1.153...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=480, ID=0523) [Reassembled in #377]
376	1.153...	192.168.0.2	192.168.0.6	IPv4	514	Fragmented IP protocol (proto=UDP 17, off=960, ID=0523) [Reassembled in #377]
377	1.153...	192.168.0.2	192.168.0.6	UDP	72	50723 → 5001 Len=1470
378	1.156...	192.168.0.2	192.168.0.6	UDP	170	5001 → 50723 Len=128
379	1.156...	192.168.0.2	192.168.0.6	UDP	170	5001 → 50723 Len=128
380	1.158...	192.168.0.2	192.168.0.6	ICMP	198	Destination unreachable (Port unreachable)

Figura 21: Recorte de la captura de segmentos UDP en *s2 – eth2*

En esta captura se pueden apreciar varios detalles de la recepción de segmentos en *h2*:

- Se observan datagramas IPv4 (en negro) y segmentos UDP (en celeste). Este hecho delata la **fragmentación** realizada por el router *r1*. Los datagramas IPv4 corresponden a los fragmentos de los segmentos originales, mientras que los segmentos UDP que vemos son el resultado del rearmado de los fragmentos (solo en aquellos casos donde se hayan recibido todos los fragmentos).
 - Analicemos el caso en el que se pudo rearmar un segmento UDP correctamente. A modo de ejemplo, tomemos el segmento con número 373 de la imagen anterior. Este fue reensamblado a partir de los fragmentos 370 al 372, tal como indica el comentario `[Reassembled in #373]`. Es decir, fue rearmado a partir de cuatro fragmentos (el último no se muestra en Wireshark, está “incluido” en el segmento 373).

La cantidad de fragmentos coincide con el valor esperado de una fragmentación de segmentos de 1470 bytes para adaptarse a un MTU de 500 bytes: si a los 1470 bytes le sumamos los 20 bytes de los headers IP de cada fragmento, obtenemos $1470B + 20B * 4 = 1550B$, los cuales deben ser separados en cuatro datagramas IPv4 de 500 bytes, cada uno conteniendo 480 bytes de datos (excepto el último, que solo contiene 30 bytes). Esto último también se puede observar teniendo en cuenta el número de *offset* de cada fragmento.

- En el caso de que se pierda al menos un fragmento de un segmento, no se puede rearmar el segmento original. Esto lo observamos, por ejemplo, en los fragmentos 364, 365 y 366: estos corresponden al segundo, tercer y cuarto fragmento de un segmento. El segmento no se reensabló ya que se perdió el primer fragmento.

- Si consideramos que cada uno de los 105 segmentos originales se dividió en 4 fragmentos, y a su vez la red cuenta con un 10% de pérdida de paquetes en todo el camino, entonces la cantidad esperada de datagramas recibidos en el receptor es de $105 * 4 - 10\% = 378$. La estimación es muy similar al valor observado: en la captura de *s2 – eth2* hay 377 datagramas IP recibidos.
- Los dos ítems anteriores evidencian que el protocolo UDP **no cuenta con un mecanismo de recuperación de errores de red**. El emisor UDP no se entera en caso de producirse una pérdida de un segmento, y tampoco realiza nada para reenviarlo.
- Una consecuencia negativa (aunque necesaria) de la fragmentación realizada por la limitación del MTU es el **aumento del tráfico de la red**: Al principio del camino viajaron 105 datagramas IP (uno por segmento), mientras que luego de la fragmentación se recibieron 377 datagramas (incluyendo las pérdidas).

9.3. Generación y análisis de tráfico TCP por la red

De manera casi idéntica a lo realizado para UDP, generamos tráfico TCP con *iperf*. Elegimos *h2* como servidor y en él ejecutamos:

```
iperf -s
```

Seleccionamos a *h1* para actuar como cliente, y en él ejecutamos:

```
iperf -c 192.168.0.6 -n 144K
```

Nuevamente, *h1* le envía 144 kilobytes de datos a *h2*. Al sacar la opción **-u**, seleccionamos TCP como protocolo de transporte en vez de UDP.

Realizamos dos capturas con Wireshark, en las mismas interfaces capturadas previamente. En el caso de *s1 – eth1*, obtuvimos una captura que muestra lo siguiente:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000	192.168.0.2	192.168.0.6	TCP	74	55299 - 5001 [SYN] Seq=0 Win=1468 SACK_PERM Tsv=1939370748 TSerr=0 WS=512
3	0.001	192.168.0.2	192.168.0.6	TCP	66	55299 - 5001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 Tsv=1939370750 TSerr=2065616974
4	0.001	192.168.0.2	192.168.0.6	TCP	126	55299 - 5001 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=60 Tsv=1939370750 TSerr=2065616974
5	0.001	192.168.0.2	192.168.0.6	TCP	7306	55299 - 5001 [PSH, ACK] Seq=61 Ack=1 Win=42496 Len=7240 Tsv=1939370750 TSerr=2065616974
6	0.001	192.168.0.2	192.168.0.6	TCP	5858	55299 - 5001 [PSH, ACK] Seq=7301 Ack=1 Win=42496 Len=5792 Tsv=1939370750 TSerr=2065616974
9	0.002	192.168.0.2	192.168.0.6	TCP	66	55299 - 5001 [ACK] Seq=139370752 Ack=29 Win=42496 Len=0 Tsv=1939370752 TSerr=2065616974
12	0.002	192.168.0.2	192.168.0.6	TCP	5907	55299 - 5001 [ACK] Seq=139370752 Ack=29 Win=42496 Len=1448 Tsv=1939370752 TSerr=2065616975
14	0.002	192.168.0.2	192.168.0.6	TCP	1514	[TCP Fast Retransmission] 55299 - 5001 [ACK] Seq=61 Ack=29 Win=42496 Len=1448 Tsv=L-1939370751 TSerr=2065616975
17	0.003	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission] 55299 - 5001 [ACK] Seq=4405 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616976
18	0.003	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission] 55299 - 5001 [ACK] Seq=10197 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616976
19	0.003	192.168.0.2	192.168.0.6	TCP	1514	55299 - 5001 [ACK] Seq=18885 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616976
21	0.003	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission] 65299 - 5001 [ACK] Seq=14541 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616976
22	0.003	192.168.0.2	192.168.0.6	TCP	4410	55299 - 5001 [ACK] Seq=23333 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616976
24	0.003	192.168.0.2	192.168.0.6	TCP	1514	55299 - 5001 [ACK] Seq=27777 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616976
26	0.003	192.168.0.2	192.168.0.6	TCP	1514	[TCP Fast Retransmission] 55299 - 5001 [ACK] Seq=4405 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616977
28	0.004	192.168.0.2	192.168.0.6	TCP	4410	55299 - 5001 [PSH, ACK] Seq=26125 Ack=29 Win=42496 Len=4344 Tsv=L-1939370752 TSerr=2065616977
38	0.004	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission] 55299 - 5001 [ACK] Seq=23229 Ack=29 Win=42496 Len=1448 Tsv=L-1939370752 TSerr=2065616977
32	0.004	192.168.0.2	192.168.0.6	TCP	1514	55299 - 5001 [ACK] Seq=30469 Ack=29 Win=42496 Len=1448 Tsv=L-1939370753 TSerr=2065616978
34	0.004	192.168.0.2	192.168.0.6	TCP	1514	55299 - 5001 [ACK] Seq=31917 Ack=29 Win=42496 Len=1448 Tsv=L-1939370753 TSerr=2065616978
36	0.004	192.168.0.2	192.168.0.6	TCP	2065616978	[TCP Retransmission] 55299 - 5001 [ACK] Seq=33355 Ack=29 Win=42496 Len=1448 Tsv=L-1939370754 TSerr=2065616978
38	0.005	192.168.0.2	192.168.0.6	TCP	1514	55299 - 5001 [PSH, ACK] Seq=43335 Ack=29 Win=42496 Len=1448 Tsv=L-1939370754 TSerr=2065616979
40	0.005	192.168.0.2	192.168.0.6	TCP	1514	[TCP Fast Retransmission] 55299 - 5001 [ACK] Seq=27573 Ack=29 Win=42496 Len=1039370754 TSerr=2065616979
42	0.006	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission] 55299 - 5001 [ACK] Seq=31917 Ack=29 Win=42496 Len=1448 Tsv=L-1939370755 TSerr=2065616979
43	0.006	192.168.0.2	192.168.0.6	TCP	1514	55299 - 5001 [ACK] Seq=34813 Ack=29 Win=42496 Len=1448 Tsv=L-1939370755 TSerr=2065616979
45	0.006	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission] 55299 - 5001 [ACK] Seq=31917 Ack=29 Win=42496 Len=1448 Tsv=L-1939370755 TSerr=2065616980

Figura 22: Recorte de la captura de segmentos TCP en *s1 – eth1* (solo los primeros segmentos enviados)

172 3. 825..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=107213 Ack=29 Win=42496 Len=1448 TSval=1939374574 TSecr=2065620795
173 4. 029..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=107213 Ack=29 Win=42496 Len=1448 TSval=1939374778 TSecr=2065620795
174 4. 437..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=107213 Ack=29 Win=42496 Len=1448 TSval=1939375186 TSecr=2065620795
175 5. 245..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=107213 Ack=29 Win=42496 Len=1448 TSval=1939375994 TSecr=2065620795
177 5. 245..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=111557 Ack=29 Win=42496 Len=1448 TSval=1939375994 TSecr=2065622219	
178 5. 245..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=111595 Ack=29 Win=42496 Len=1448 TSval=1939375994 TSecr=2065622219	
179 5. 245..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=111595 Ack=29 Win=42496 Len=1448 TSval=1939375994 TSecr=2065622219	
181 5. 245..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=114453 Ack=29 Win=42496 Len=1448 TSval=1939375994 TSecr=2065622219	
184 6. 845..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=1110109 Ack=29 Win=42496 Len=1448 TSval=1939377594 TSecr=2065622219
186 6. 845..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=114453 Ack=29 Win=42496 Len=1448 TSval=1939377594 TSecr=2065623819
187 6. 845..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=115901 Ack=29 Win=42496 Len=1448 TSval=1939377594 TSecr=2065623819	
189 6. 845..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=117349 Ack=29 Win=42496 Len=1448 TSval=1939377594 TSecr=2065623819	
199 7. 093..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=121693 Ack=29 Win=42496 Len=1448 TSval=1939377842 TSecr=2065623819	
201 7. 093..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=123141 Ack=29 Win=42496 Len=1448 TSval=1939377842 TSecr=2065624067	
203 7. 093..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=115901 Ack=29 Win=42496 Len=1448 TSval=1939377842 TSecr=2065624067
204 7. 093..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=121693 Ack=29 Win=42496 Len=1448 TSval=1939377842 TSecr=2065624067
206 7. 093..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=123141 Ack=29 Win=42496 Len=1448 TSval=1939381435 TSecr=2065624067
206 15. 54..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=123141 Ack=29 Win=42496 Len=1448 TSval=1939381435 TSecr=2065624067
208 15. 54..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=126037 Ack=29 Win=42496 Len=1448 TSval=1939386298 TSecr=2065632523	
209 15. 54..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=127485 Ack=29 Win=42496 Len=1448 TSval=1939386298 TSecr=2065632523	
211 15. 54..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=128933 Ack=29 Win=42496 Len=1448 TSval=1939386298 TSecr=2065632523	
212 15. 54..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=130381 Ack=29 Win=42496 Len=1448 TSval=1939386298 TSecr=2065632523	
214 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=127485 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611
215 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=128933 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611
217 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=131829 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
219 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=133277 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
221 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=131829 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611
223 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=134725 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
224 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=136173 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
227 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=137621 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
228 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=138989 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
231 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=140517 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
232 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=141965 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
235 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=143413 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
236 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	55290 → 5001 [ACK] Seq=144861 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611	
238 15. 63..	192.168.0.2	192.168.0.6	TCP	1214	55290 → 5001 [FIN, PSH]	Seq=146309 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611
240 15. 63..	192.168.0.2	192.168.0.6	TCP	1514	[TCP Retransmission]	55290 → 5001 [ACK] Seq=144861 Ack=29 Win=42496 Len=1448 TSval=1939386386 TSecr=2065632611
243 15. 64..	192.168.0.2	192.168.0.6	TCP	66	55290 → 5001 [ACK] Seq=147458 Ack=39 Win=42496 Len=0 TSval=1939386397 TSecr=2065632622	

Figura 23: Recorte de la captura de segmentos TCP en s1 – eth1 (solo los últimos segmentos enviados)

2.0.001..	192.168.0.6	192.168.0.2	TCP	74	55290 [ENV, ACK]	Seq=0	Ack=1	Wnn=43469	Lenn=43469	TSval=1939374748	TSecr=1939374748 w3=512
7.0.002..	192.168.0.2	192.168.0.6	TCP	66	55290 [ACK]	Seq=1	Ack=f1	Wnn=43529	Lenn=0	TSval=1939374675	TSecr=1939370759
8.0.002..	192.168.0.2	192.168.0.6	TCP	64	55291 [PSH]	Seq=1	Ack=f1	Wnn=43529	Lenn=28	TSval=1939370759	TSecr=1939370759
10.0.002..	192.168.0.2	192.168.0.6	TCP	78	TCP Dup ACK 7/1	55291	→ 55290 [ACK]	Seq=29	Ack=e1	Wnn=43529	Len=1448 TSval=1939370759 SLE=15098 SRE=14405
11.0.002..	192.168.0.2	192.168.0.6	TCP	78	TCP Dup ACK 7/2	55291	→ 55290 [ACK]	Seq=29	Ack=e1	Wnn=43529	Len=1448 TSval=1939370759 SLE=15098 SRE=14405
13.0.002..	192.168.0.2	192.168.0.6	TCP	78	TCP Dup ACK 7/3	55291	→ 55290 [ACK]	Seq=29	Ack=e1	Wnn=43529	Len=1448 TSval=1939370759 SLE=15098 SRE=14405
15.0.002..	192.168.0.2	192.168.0.6	TCP	80	TCP Dup ACK 7/4	55291	→ 55290 [ACK]	Seq=29	Ack=e1	Wnn=43529	Len=1448 TSval=1939370759 SLE=15098 SRE=14405
17.0.002..	192.168.0.2	192.168.0.6	TCP	80	TCP Dup ACK 7/5	55291	→ 55290 [ACK]	Seq=29	Ack=e1	Wnn=43529	Len=1448 TSval=1939370759 SLE=15098 SRE=14405
20.0.003..	192.168.0.6	192.168.0.2	TCP	94	55291 → 55290 [ACK]	Seq=29	Ack=4495	Win=39424	Len=1448 TSval=1939386386 TSecr=1939370759 SLE=14541 SRE=5853 SRE=10197		
23.0.003..	192.168.0.6	192.168.0.2	TCP	95	TCP Dup ACK 20/1	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=39424	Len=1448 TSval=1939386386 TSecr=1939370759 SLE=14541 SRE=10197
24.0.003..	192.168.0.6	192.168.0.2	TCP	95	TCP Dup ACK 20/2	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=39424	Len=1448 TSval=1939386386 TSecr=1939370759 SLE=14541 SRE=10197
27.0.004..	192.168.0.6	192.168.0.2	TCP	78	TCP Dup ACK 20/3	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=39424	Len=1448 TSval=1939386386 TSecr=1939370759 SLE=14541 SRE=10197
29.0.004..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=20992	Len=8 TSval=1939370759 SLE=24077 SRE=8125
31.0.004..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=20992	Len=8 TSval=1939370759 SLE=24077 SRE=8125
35.0.005..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=25560	Len=8 TSval=1939370759 SLE=30469 SRE=31917
37.0.005..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=25560	Len=8 TSval=1939370759 SLE=27673 SRE=31917
39.0.005..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4495	Win=25560	Len=8 TSval=1939370759 SLE=33345 SRE=34813 SLE=29021 SRE=31917
41.0.006..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4497	Win=21594	Len=8 TSval=1939370759 SLE=33345 SRE=34813
44.0.006..	192.168.0.6	192.168.0.2	TCP	78	TCP Window Update	55291	→ 55290 [ACK]	Seq=29	Ack=4497	Win=150016	Len=8 TSval=1939370759 SLE=33345 SRE=34813
46.0.006..	192.168.0.6	192.168.0.2	TCP	66	55291 → 55290 [ACK]	Seq=29	Ack=4497	Win=4820	Len=8 TSval=1939370759 SLE=33345 SRE=34813		
50.0.011..	192.168.0.6	192.168.0.2	TCP	78	TCP Dup ACK 21/1	55291	→ 55290 [ACK]	Seq=29	Ack=47743	Win=150016	Len=8 TSval=1939370759 SLE=39157 SRE=40605

Figura 24: Recorte de la captura de segmentos TCP en s1 – eth1 (solo los primeros segmentos recibidos)

170 5. 245..	192.168.0.6	192.168.0.2	TCP	78	TCP Dup ACK 21/1	55291	→ 55290 [ACK]	Seq=29	Ack=111557	Win=151652	Len=1448 TSval=1939374574 SLE=111557 SRE=114453
185 6. 845..	192.168.0.6	192.168.0.2	TCP	66	55291 → 55290 [ACK]	Seq=29	Ack=114453	Win=147496	Len=1448 TSval=1939374574 SLE=2065620795		
188 6. 845..	192.168.0.6	192.168.0.2	TCP	66	55291 → 55290 [ACK]	Seq=29	Ack=114453	Win=147496	Len=1448 TSval=1939374574 SLE=2065620795		
191 7. 093..	192.168.0.6	192.168.0.2	TCP	78	[TCP Dup ACK 188/1]	55291	→ 55290 [ACK]	Seq=29	Ack=15901	Win=150016	Len=8 TSval=1939377594 SLE=118797 SRE=120245
193 7. 093..	192.168.0.6	192.168.0.2	TCP	78	55291 → 55290 [ACK]	Seq=29	Ack=117349	Win=148992	Len=0	TSval=1939377594 SLE=118797 SRE=120245	
196 7. 093..	192.168.0.6	192.168.0.2	TCP	78	55291 → 55290 [ACK]	Seq=29	Ack=120245	Win=148992	Len=0	TSval=1939377594 SLE=118	

- Establecimiento de conexión (*Three-Way Handshake*), con los segmentos [SYN], [SYN, ACK] y el primer [ACK].
- Acuse de recibo (*acknowledgment*) de segmentos.
- Retransmisión de segmentos por *timeout* y *Fast Retransmission*.
- Control de flujo (segmentos [TCP Window Update]).
- Cierre de conexión (segmentos [FIN]).

Del otro lado, tenemos la captura realizada en *s2 – eth2*:

No.	Time	Source	Destination	Protocol	Length	Info
2	0.000	192.168.0.6	192.168.0.2	TCP	74	5901 → 55290 [SYN] Seq=0 Win=4440 Len=0 MSS=1468 SACK_PERM TSeq=193537074 TSerr=193537074 WS=512
5	0.001	192.168.0.6	192.168.0.2	TCP	66	5901 → 55290 [ACK] Seq=1 Ack=61 Win=43520 Len=0 TSeq=2065616975 TSerr=1939370750
6	0.001	192.168.0.6	192.168.0.2	TCP	94	5901 → 55290 [PSH, ACK] Seq=1 Ack=61 Win=43520 Len=28 TSeq=2065616975 TSerr=1939370750
14	0.001	192.168.0.6	192.168.0.2	TCP	78	TSeq=2065616975 ACK=62 Win=43520 Len=0 TSeq=2065616975 TSerr=1939370750 SLE=1509 SRE=2057
15	0.001	192.168.0.6	192.168.0.2	TCP	5081	→ 55290 [ACK] Seq=29 Ack=61 Win=43520 Len=0 TSeq=2065616975 TSerr=1939370750 SLE=1509 SRE=4405
26	0.001	192.168.0.6	192.168.0.2	TCP	87	TSeq=2065616975 ACK=62 Win=43520 Len=0 TSeq=2065616975 TSerr=1939370750 SLE=1509 SRE=4405
47	0.001	192.168.0.6	192.168.0.2	TCP	78	TSeq=2065616975 ACK=62 Win=43520 Len=0 TSeq=2065616975 TSerr=1939370750 SLE=1509 SRE=4405
59	0.001	192.168.0.6	192.168.0.2	TCP	5081	→ 55290 [ACK] Seq=29 Ack=61 Win=43520 Len=0 TSeq=2065616975 TSerr=1939370750 SLE=1645 SRE=14541 SLE=5853 SRE=10197 SLE=1509 SRE=4405
63	0.002	192.168.0.6	192.168.0.2	TCP	94	5901 → 55290 [ACK] Seq=29 Ack=4465 Win=39424 Len=0 TSeq=1939370751 SLE=1509 SRE=18885 SLE=11645 SRE=14541 SLE=5853 SRE=18197
64	0.002	192.168.0.6	192.168.0.2	TCP	78	TSeq=1939370751 ACK=4465 Win=39424 Len=0 TSeq=1939370751 SLE=1509 SRE=18885 SLE=11645 SRE=14541 SLE=5853 SRE=18197
75	0.002	192.168.0.6	192.168.0.2	TCP	5081	→ 55290 [ACK] Seq=29 Ack=4465 Win=39424 Len=0 TSeq=1939370751 SLE=1509 SRE=20333 SLE=5853 SRE=14541
92	0.002	192.168.0.6	192.168.0.2	TCP	78	TSeq=1939370751 ACK=4465 Win=39424 Len=0 TSeq=1939370751 SLE=1509 SRE=23239
103	0.002	192.168.0.6	192.168.0.2	TCP	78	[TCP Window Update] 5081 → 55290 [ACK] Seq=29 Ack=4465 Win=39424 Len=0 TSeq=1939370751 SLE=1509 SRE=23239
112	0.003	192.168.0.6	192.168.0.2	TCP	78	[TCP Window Update] 5081 → 55290 [ACK] Seq=29 Ack=23229 Win=24684 Len=0 TSeq=2065616978 TSerr=1939370752 SLE=24677 SRE=26125
117	0.003	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=27573 Win=22528 Len=0 TSeq=1939370753 TSerr=1939370753 SLE=3365 SRE=31917
122	0.003	192.168.0.6	192.168.0.2	TCP	78	[TCP Window Update] 5081 → 55290 [ACK] Seq=29 Ack=27573 Win=22528 Len=0 TSeq=1939370753 TSerr=1939370753 SLE=24677 SRE=27573
127	0.003	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=27573 Win=22528 Len=0 TSeq=1939370753 TSerr=1939370753 SLE=3365 SRE=31917
138	0.004	192.168.0.6	192.168.0.2	TCP	94	[TCP Dup ACK 117[2]] 5081 → 55290 [ACK] Seq=29 Ack=27573 Win=22528 Len=0 TSeq=1939370753 TSerr=1939370753 SLE=3365 SRE=31917
143	0.004	192.168.0.6	192.168.0.2	TCP	78	5901 → 55290 [ACK] Seq=29 Ack=11917 Win=21504 Len=0 TSeq=2065616979 TSerr=1939370754 SLE=3365 SRE=34813
148	0.004	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=11917 Win=21504 Len=0 TSeq=2065616979 TSerr=1939370754 SLE=3365 SRE=34813
150	0.005	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=58261 Win=48128 Len=0 TSeq=1939370755 TSerr=1939370755 SLE=336261
168	0.010	192.168.0.6	192.168.0.2	TCP	78	5901 → 55290 [ACK] Seq=29 Ack=57709 Win=54272 Len=0 TSeq=1939370755 TSerr=1939370755 SLE=31917 SRE=49695
176	0.010	192.168.0.6	192.168.0.2	TCP	78	[TCP Window Update] 5081 → 55290 [ACK] Seq=29 Ack=57709 Win=54272 Len=0 TSeq=1939370755 TSerr=1939370755 SLE=31917 SRE=42053

Figura 26: Recorte de la captura de segmentos TCP en *s2 – eth2* (solo los primeros segmentos enviados)

No.	Time	Source	Destination	Protocol	Length	Info
582	15.54	192.168.0.6	192.168.0.2	TCP	66	5901 → 55290 [ACK] Seq=29 Ack=126037 Win=148400 Len=0 TSeq=1939386298 TSerr=1939386298
590	15.54	192.168.0.6	192.168.0.2	TCP	66	5901 → 55290 [ACK] Seq=29 Ack=127485 Win=150803 Len=0 TSeq=2065622523 TSerr=1939386298
601	15.54	192.168.0.6	192.168.0.2	TCP	66	5901 → 55290 [ACK] Seq=29 Ack=127485 Win=150803 Len=0 TSeq=2065622523 TSerr=1939386298
602	15.63	192.168.0.6	192.168.0.2	TCP	78	5901 → 55290 [ACK] Seq=29 Ack=128993 Win=148992 Len=0 TSeq=2065622611 TSerr=1939386380 SLE=131829
609	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=131361 Win=148400 Len=0 TSeq=2065622611 TSerr=1939386380
614	15.63	192.168.0.6	192.168.0.2	TCP	78	[TCP Window Update] 5081 → 55290 [ACK] Seq=29 Ack=131361 Win=148400 Len=0 TSeq=2065622611 TSerr=1939386380
628	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=134725 Win=148400 Len=0 TSeq=2065622611 TSerr=1939386380
619	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=134725 Win=148400 Len=0 TSeq=2065622611 TSerr=1939386380
628	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=134725 Win=148400 Len=0 TSeq=2065622611 TSerr=1939386380
630	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=134725 Win=148400 Len=0 TSeq=2065622611 TSerr=1939386380
638	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=139969 Win=150803 Len=0 TSeq=2065622611 TSerr=1939386380
639	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=139969 Win=150803 Len=0 TSeq=2065622611 TSerr=1939386380
648	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=140017 Win=150803 Len=0 TSeq=2065622611 TSerr=1939386380
649	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=143413 Win=150803 Len=0 TSeq=2065622611 TSerr=1939386380
651	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=144801 Win=150803 Len=0 TSeq=2065622611 TSerr=1939386380
661	15.63	192.168.0.6	192.168.0.2	TCP	78	[TCP Window Update] 5081 → 55290 [ACK] Seq=29 Ack=144801 Win=150803 Len=0 TSeq=2065622611 TSerr=1939386380
666	15.63	192.168.0.6	192.168.0.2	TCP	68	5901 → 55290 [ACK] Seq=29 Ack=147458 Win=149092 Len=0 TSeq=2065622611 TSerr=1939386380
667	39.39	192.168.0.6	192.168.0.2	ICMP	64	Time-to-live exceeded [Fragment reassembly time exceeded]
670	39.39	192.168.0.6	192.168.0.2	ICMP	542	Time-to-live exceeded [Fragment reassembly time exceeded]
671	39.39	192.168.0.6	192.168.0.2	ICMP	542	Time-to-live exceeded [Fragment reassembly time exceeded]
672	39.39	192.168.0.6	192.168.0.2	ICMP	542	Time-to-live exceeded [Fragment reassembly time exceeded]
673	39.39	192.168.0.6	192.168.0.2	ICMP	542	Time-to-live exceeded [Fragment reassembly time exceeded]
674	39.39	192.168.0.6	192.168.0.2	ICMP	542	Time-to-live exceeded [Fragment reassembly time exceeded]
678	39.39	192.168.0.6	192.168.0.2	ICMP	542	Time-to-live exceeded [Fragment reassembly time exceeded]

Figura 27: Recorte de la captura de segmentos TCP en *s2 – eth2* (solo los últimos segmentos enviados)

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000	192.168.0.2	192.168.0.6	TCP	74	55299 → 5001 [SYN] Seq=0 Win=42340 Len=0 MSS=1468 SACK_PERM TSeq=1939370748 TSerr=0 WS=512
3	0.000	192.168.0.2	192.168.0.6	TCP	66	55299 → 5001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSeq=1939370750 TSerr=2065616974
4	0.001	192.168.0.2	192.168.0.6	TCP	66	55299 → 5001 [PSH, ACK] Seq=1 Ack=1 Win=42496 Len=0 TSeq=1939370750 TSerr=2065616974
7	0.001	192.168.0.2	192.168.0.6	TCP	66	55299 → 5001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSeq=1939370750 TSerr=2065616974
23	0.001	192.168.0.2	192.168.0.6	TCP	66	55299 → 5001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSeq=1939370750 TSerr=2065616974
24	0.001	192.168.0.2	192.168.0.6	TCP	66	55299 → 5001 [ACK] Seq=1 Ack=1 Win=42496 Len=0 TSeq=1939370750 TSerr=2065616974
25	0.001	192.168.0.2	192.168.0.6	TCP	74	[TCP Previous segment not captured] 55299 → 5001 [PSH, ACK] Seq=5853 Ack=1 Win=42496 Len=1448 TSeq=1939370750 TSerr=2065616974
27	0.001	192.168.0.2	192.168.0.6	TCP	514	Fragmented IP protocol (proto=TCP 6, off=488, ID=5e63) [Reassembled in #30]
28	0.001	192.168.0.2	192.168.0.6	TCP	514	Fragmented IP protocol (proto=TCP 6, off=488, ID=5e63) [Reassembled in #30]
29	0.001	192.168.0.2	192.168.0.6	TCP	514	Fragmented IP protocol (proto=TCP 6, off=488, ID=5e63) [Reassembled in #30]
30	0.001	192.168.0.2	192.168.0.6	TCP	74	55299 → 5001 [ACK] Seq=7301 Ack=1 Win=42496 Len=1448 TSeq=1939370750 TSerr=2065616974
31	0.001	192.168.0.2	192.168.0.6	TCP	514	Fragmented IP protocol (proto=TCP 6, off=0, ID=5e64) [Reassembled in #34]
32	0.001	192.168.0.2	192.168.0.6	TCP	514	Fragmented IP protocol (proto=TCP 6, off=488, ID=5e64) [Reassembled in #34]
33	0.001	192.168.0.2	192.168.0.6	TCP	514	Fragmented IP protocol (proto=TCP 6, off=488, ID=5e64) [Reassembled in #34]
34	0.001	192.168.0.2	192.168.0.6	TCP	74	55299 → 5001 [ACK] Seq=5749 Ack=1 Win=42496 Len=1448 TSeq=1939370750 TSerr=2065616974

Figura 28: Recorte de la captura de segmentos TCP en *s2 – eth2* (solo los primeros segmentos recibidos)

Source IP	Source Port	Destination IP	Destination Port	Protocol	Sequence Number	Length	Flags	Information
659 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=0, ID=5ef7] [Reassembled in #653]				
651 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=480, ID=5ef7] [Reassembled in #653]				
652 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=960, ID=5ef7] [Reassembled in #653]				
653 15 .63	192 .168 .0 .2	192 .168 .0 .6	TCP	74 55290 - 5081 [ACK] Seq=143413 Ack=29 Win=42496 Len=1448 TStamp=1939386386 TSrcr=2065632611				
654 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=0, ID=5ef8]				
655 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=480, ID=5ef8]				
656 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=960, ID=5ef8]				
657 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=0, ID=5ef9] [Reassembled in #650]				
658 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=480, ID=5ef9] [Reassembled in #650]				
659 15 .63	192 .168 .0 .2	192 .168 .0 .6	TCP	254 [TCP Previous segment not captured] 55290 - 5081 [STW] DSr ACK Seq=1436309 Ack=29 Win=42496 Len=1148 TStamp=1939386386 TSsrc=2065632611				
660 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=0, ID=5ef9] [Reassembled in #651]				
661 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=480, ID=5ef9] [Reassembled in #651]				
662 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=960, ID=5ef9] [Reassembled in #651]				
663 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=0, ID=5ef9] [Reassembled in #652]				
664 15 .63	192 .168 .0 .2	192 .168 .0 .6	IPV4	514 Fragmented IP protocol [proto=TCP 6, offf=480, ID=5ef9] [Reassembled in #652]				
665 15 .63	192 .168 .0 .2	192 .168 .0 .6	TCP	74 [TCP retransmission of Order] 55290 - 5081 [ACK] Seq=144081 Ack=29 Win=42496 Len=1448 TStamp=1939386386 TSsrc=2065632611				
666 15 .64	192 .168 .0 .2	192 .168 .0 .6	TCP	66 55290 - 5081 [ACK] Seq=147458 Ack=30 Win=42496 Len=8 TStamp=1939386397 TSsrc=2065632622				
669 30 .39	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				
670 30 .39	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				
671 30 .39	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				
672 30 .39	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				
673 30 .39	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				
674 30 .39	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				
675 31 .93	192 .168 .0 .6	192 .168 .0 .2	ICMP	542 Time-to-live exceeded (Fragment reassembly time exceeded)				

Figura 29: Recorte de la captura de segmentos TCP en *s2 – eth2* (solo los últimos segmentos recibidos)

En esta captura, se puede observar el mismo proceso de reensamble de fragmentos que con las capturas de UDP, además del visible **aumento del tráfico en la red** producido por la fragmentación.

En caso de perderse un fragmento (y por lo tanto un segmento), el host receptor no pierde la posibilidad de recibir la información que contenía (a diferencia de UDP), gracias al mecanismo de recuperación de errores de red de TCP mencionado anteriormente. Podemos comprobar que toda la información transmitida llegó correctamente observando la salida del comando *iperf* en el servidor (receptor):

```
^Croot@agustin-pc:/home/agustin/Documentos/redes/FIUBA-REDES-TP1# iperf -s
-----
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 1] local 192.168.0.6 port 5001 connected with 192.168.0.2 port 55290 (icwnd/mss/irtt=14/1448/671)
[ ID] Interval      Transfer     Bandwidth
[ 1] 0.0000-15.6362 sec   144 KBytes  75.4 Kbits/sec
```

Figura 30: Salida del comando *iperf* en el servidor

Como se puede observar, el servidor recibió correctamente los 144 KB enviados originalmente por el cliente.